System Support for Adaptive Pervasive Applications

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Marcus Handte

aus Nürtingen

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. phil. nat. Christian Becker
Tag der mündlichen Prüfung:	09.07.2009

Institut für Parallele und Verteilte Systeme (IPVS)

Universität Stuttgart

2009

Table of Contents

Abstract

Driven by the ongoing miniaturization of computer technology as well as the proliferation of wireless communication technology, Pervasive Computing envisions seamless and distraction-free task support by distributed applications that are executed on computers embedded in everyday objects. As such, this vision is equally appealing to the computer industry and the user. Induced by various factors such as invisible integration, user mobility and computer failures, the resulting computer systems are heterogeneous, highly dynamic and evolving. As a consequence, applications that are executed in these systems need to adapt continuously to their ever-changing execution environment. Without further precautions, the need for adaptation can complicate application development and utilization which hinders the realization of the basic vision.

As solution to this dilemma, this dissertation describes the design of system software for Pervasive Computing that simplifies the development of adaptive applications. As opposed to shifting the responsibility for adapting an application to the user or the application developer, the system software introduces a component-based application model that can be configured and adapted automatically. To enable automation at the system level, the application developer specifies the dependencies on components and resources in an abstract manner using contracts. Upon application startup, the system uses the contractual descriptions to compute and execute valid configurations. At runtime, it detects changes to the configuration that require adaptation and it reconfigures the application.

To compute valid configurations upon application startup, the dissertation identifies the requirements for configuration algorithms. Based on an analysis of the problem complexity, the dissertation classifies possible algorithmic solutions and it presents an integrated approach for configuration based on a parallel backtracking algorithm. Besides from scenario specific modifications, retrofitting the backtracking algorithm requires a problem mapping from configuration to constraint satisfaction which can be computed on-the-fly at runtime. The resulting approach for configuration is then extended to support the optimization of a cost function that captures the most relevant cost factors during adaptation. This enables the use of the approach for configuration upon startup and reconfiguration during runtime adaptation.

As basis for the evaluation of the system software and the algorithm, the dissertation outlines a prototypical implementation. The prototypical implementation is used for a thorough evaluation of the presented concepts and algorithms by means of real world measurements and a number of simulations. The evaluation results suggest that the presented system software can indeed simplify the development of distributed applications that compensate the heterogeneity, dynamics and evolution of the underlying system. Furthermore, they indicate that the algorithm for configuration and the extensions for adaptation provide a sufficiently high performance in typical applications scenarios. Moreover, the results also suggest that they are preferable over of alternative solutions.

To position the presented solution within the space of possible and existing solutions, the dissertation discusses major representatives of existing systems and it proposes a classification of the relevant aspects. The relevant aspects are the underlying conceptual model of the system and the distribution of the responsibility for configuration and adaptation. The classification underlines that in contrast to other solutions, the presented solution provides a higher degree of automation without relying on the availability of a powerful computer. Thus, it simplifies the task of the application developer without distracting the user while being applicable to a broader range of scenarios. After discussing the related approaches and clarifying similarities and differences, the dissertation concludes with a short summary and an outlook on future work.

Zusammenfassung

Das vorliegende Kapitel fasst den Inhalt der Arbeit in deutscher Sprache zusammen. Entsprechend folgt der Aufbau dieses Kapitels dem Aufbau der detaillierten Ausarbeitung, die in englischer Sprache verfasst ist. Kapitel 1 skizziert zunächst das Themengebiet und motiviert die grundlegende Problemstellung. Die Kapitel 2 bis 4 beschreiben die Ansätze für die resultierenden Teilprobleme auf konzeptioneller Ebene. Kapitel 5 skizziert eine prototypische Implementierung der zuvor entwickelten Konzepte. Kapitel 6 diskutiert die Evaluation der in Kapitel 2 bis 4 beschriebenen Ansätze auf Basis der Implementierung aus Kapitel 5. Kapitel 7 ordnet die Arbeit in den Rahmen existierender Ansätze ein. Kapitel 8 schließt mit einer kurzen Zusammenfassung und einem Ausblick auf einige weitergehende Forschungsfragen.

1 Einleitung

Angetrieben von der voranschreitender Miniaturisierung und der anhaltenden exponentiellen Leistungssteigerung, waren Rechnersysteme über die letzten Jahrzehnte in der Lage, sich ständig neue Einsatzgebiete zu erschließen. Bereits heute wird der überwiegende Teil an Rechnern nicht mehr als Allzweckrechner eingesetzt, sondern er findet seine Bestimmung als eingebettetes System zur Überwachung und Steuerung von physikalischen Prozessen. Mit Hilfe von drahtloser Vernetzungstechnik, wird es zukünftigen Generationen von eingebetteten Systemen in verstärktem Maße möglich sein, miteinander zu kommunizieren.

Aufbauend auf diesen Trends beschreibt die Vision des Pervasive Computings – also die Vision des alles durchdringenden Rechnens – einen grundlegenden Paradigmenwechsel hinsichtlich des Einsatzes und der Nutzung von Rechnersystemen. Durch die Integration werden Rechner immer weniger als solche zu erkennen sein und durch die drahtlose Vernetzung können sie bei Bedarf spontan miteinander interagieren. Dadurch sind sie in der Lage, selbständig komplizierte Aufgaben zu lösen, die nicht oder nur bedingt von einem einzelnen eingebetteten System gelöst werden können. Die Anwendungen, die gemeinsam von diesen eingebetteten Systemen ausgeführt werden, werden ihre Nutzer nahtlos in ihrem täglichen Leben unterstützen, ohne sie von ihren Aufgaben abzulenken.

Die aus dieser Vision resultierenden vernetzten Systeme unterscheiden sich von herkömmlichen. Insbesondere werden sie, bedingt durch die Spezialisierung einzelner Rechner, hochgradig heterogen sein. Weiterhin werden sie durch die Eigenschaften der drahtlosen Kommunikationstechnik und durch die Mobilität von Benutzern und Rechnern zeitweise hoch dynamisch sein. Darüber hinaus wird sich die eingesetzte Rechnertechnik schneller entwickeln als sie von ihren Eigentümern ersetzt wird. Dies führt letztlich dazu, dass die vernetzten Systeme aus Rechnern unterschiedlicher Generationen bestehen werden.

Aus Sicht der Anwendungen führen diese Eigenschaften zu zusätzlichen Anforderungen, da sie sich an die Heterogenität und Dynamik anpassen müssen um dem Anwender eine möglichst nahtlose Unterstützung zu bieten. Allerdings können Anwendungen zur Steuerung der Anpassung nicht auf den Nutzer zurückgreifen, da dieser durch die zusätzliche Interaktion von der zu erledigenden Aufgabe abgelenkt wird. Dies wiederum erschwert die Aufgabe des Anwendungsentwicklers, da dieser neben der eigentlichen Anwendungslogik zusätzlich noch robuste Anpassungslogik entwickeln muss.

Um diesem Problem zu begegnen, entwickelt, beschreibt und bewertet diese Arbeit Konzepte und Verfahren, mit denen die Anpassung einer verteilten Anwendung an die verfügbaren Ressourcen automatisiert werden kann. Neben der strukturellen Anpassung der Verteilung durch Konfiguration wird dabei auch die Anpassung einzelner Anwendungsteile durch Parametrisierung berücksichtigt. Der Fokus liegt vornehmlich auf einer reaktiven Anpassung bei der auch Änderungen berücksichtigt werden können, die während der Ausführung nur schlecht oder überhaupt nicht vorherzusehen sind.

Im Gegensatz zu existierenden Konzepten und Verfahren zielt die vorliegende Arbeit vorwiegend auf den Einsatz in spontan vernetzten Systemen ab, die aus einer Reihe von ressourcenarmen Rechnern bestehen. Da deshalb nicht von der ständigen Verfügbarkeit eines ressourcenreichen Rechners ausgegangen werden kann, sind die resultierenden Verfahren grundsätzlich vollständig verteilt. Zum einen kann der vorgestellte Ansatz dadurch in einem breiten Spektrum unterschiedlicher Szenarien effektiv eingesetzt werden und zum anderen kann dadurch die inhärente Parallelität dieser vernetzten Systeme zur Leistungssteigerung ausgenutzt werden.

2 Systemsoftware

Als Basis für die automatische Anpassung der Struktur und Parametrisierung einer verteilten Anwendung ist es zunächst erforderlich, die zulässigen Strukturen und Parametrisierungen sowie die umsetzbaren Konfigurationen in geeigneter Weise zu erfassen. Da die zulässigen Strukturen und Parametrisierungen anwendungsabhängig sind, müssen diese vom Anwendungsentwickler vorgegeben werden. Unter Berücksichtigung der jeweils verfügbaren Rechner und Ressourcen können daraufhin mögliche Konfigurationen automatisch berechnet, ausgeführt und falls notwendig auch angepasst werden.

Kapitel 2 diskutiert hierfür die grundlegenden Konzepte des Komponentensystems PCOM das speziell für verteilte Anwendungen entwickelt wurde, die zur Laufzeit automatisch konfiguriert und angepasst werden müssen. Um die Konzepte des Systems zu motivieren, werden zunächst die Anforderungen an ein solches System abgeleitet. Die Anforderungen ergeben sich dabei zum einen aus den Eigenschaften zukünftiger Rechnersysteme, die im vorhergehenden Kapitel erläutert wurden, und zum anderen aus dem Wunsch eine für den Anwendungsentwickler möglichst transparente Anpassung der Anwendung zur Laufzeit zu ermöglichen.

Auf Basis der Anforderungen wird daraufhin beschrieben, wie ein flexibles und häufig eingesetztes dienstbasiertes Anwendungsmodell in ein Komponentenmodell transformiert werden kann, das die automatische Konfiguration und Anpassung der Anwendung zur Laufzeit erlaubt. Im Gegensatz zu einem Dienst, bei dem lediglich die angebotene Funktionalität spezifiziert wird, werden Komponenten mit einer zusätzlichen Beschreibung ihrer Abhängigkeiten ausgestattet. Die Beschreibung der Abhängigkeiten umfasst Komponenten, die von der jeweiligen Komponente lokal oder entfernt benötigt werden, und lokale Ressourcen. Das Angebot und die Abhängigkeiten einer Komponente werden in Form von sogenannten Verträgen festgehalten. Verträge stellen eine Zusicherung dar, die besagt, dass eine Komponente die spezifizierte Funktionalität in der festgelegten Güte erbringen kann solange die dafür notwendigen Komponenten und Ressourcen in ausreichender Güte und Anzahl verfügbar sind. Um verschiedene Gütestufen zu spezifizieren, können Komponenten grundsätzlich mit mehreren, alternativen Verträgen ausgestattet werden, die wahlweise einsetzbar sind.

Durch einen automatischen Abgleich von vertraglichen Angeboten und Anforderungen können die Komponenten, die auf einer Reihe von vernetzten Rechnern verfügbar sind, zu verteilten Anwendungen zusammengesetzt werden. Die eigentliche Anwendung wird dabei immer durch die sogenannte Kernkomponente repräsentiert, die entlang ihrer rekursiven Abhängigkeiten einen Baum von Komponenten aufspannt. Da der Vertrag einer Komponente ihre Funktionalität nur dann zusichert, wenn die vertraglichen Anforderungen erfüllt werden können, ergibt sich daraus konsequenterweise, dass eine Anwendung genau dann ausgeführt werden kann, wenn die vertraglichen Anforderungen der gesamten Anwendung erfüllt werden können.

Da sich die verfügbaren Komponenten und Ressourcen während der Ausführung einer Anwendung ändern können, kann eine ausführbare Anwendung zu jedem Zeitpunkt nicht mehr ausführbar werden. Aus diesem Grund muss die Ausführung der Anwendung ständig überwacht werden und bei Bedarf muss die Zusammenstellung der Komponenten und Verträge angepasst werden. Da dies eine anwendungsübergreifende Aufgabe ist, wird sie von einem sogenannten Komponentencontainer übernommen, der auf jedem System ausgeführt wird. Neben der reinen Überwachung übernimmt der Komponentencontainer noch weitere Aufgaben, die die Anwendungsentwicklung vereinfachen. Beispiele hierfür sind die Steuerung des Lebenszyklus einer Komponente, die Erzeugung und Bindung von Stellvertreterobjekten zur entfernten

Kommunikation und die Sicherung und Übertragung des Anwendungszustands von zustandsbehafteten Komponenten sofern diese ausgetauscht werden müssen. Darüber hinaus initiiert der Komponentencontainer auch die Konfiguration einer Anwendung zum Startzeitpunkt und er liefert notwendige Aufwandsschätzungen während der Anpassung der Anwendung. Dazu greift er auf den Konfigurationsalgorithmus und der Optimierungsheuristik, die in den Kapiteln 3 und 4 entwickelt werden, zurück.

3 Automatische Konfiguration

Sobald eine Anwendung gestartet werden soll, muss zunächst eine ausführbare Konfiguration berechnet werden. Da die verfügbaren Komponenten und Ressourcen von der jeweiligen Zusammensetzung und aktuellen Auslastung der verfügbaren Rechner abhängen, muss die Berechnung zur Laufzeit unter Berücksichtigung der jeweiligen Verfügbarkeit stattfinden. Dies erfordert ein Berechnungsverfahren für Konfigurationen, das eine befriedigende Leistung erbringt und die Rechenkapazität der ressourcenarmen Rechner nicht überfordert.

Kapitel 3 stellt die grundlegenden Konzepte eines solches Verfahren vor. Hierzu wird zunächst das Problem der automatischen Konfiguration mit Hilfe eines Beispiels verdeutlicht und in einem weiteren Schritt formalisiert. Daraufhin wird die Problemkomplexität untersucht. Die Analyse zeigt, dass das Problem NP-vollständig ist. Nach der Problemanalyse werden die Anforderungen an Verfahren zur Konfiguration diskutiert. Neben der Eignung für ressourcenarme, spontan vernetzte Rechner sind vor allem Effizienz und Vollständigkeit zwei wesentliche Anforderungen. Eine detaillierte Betrachtung dieser Anforderungen verdeutlicht, dass sie aufgrund der Problemkomplexität in Konflikt zueinander stehen. Im Rahmen dieser Arbeit wird daraufhin davon ausgegangen, dass die Effizienz des Verfahrens der Vollständigkeit unterzuordnen ist, da ein unvollständiges Verfahren zu einem Systemverhalten führen kann, das für den Nutzer nicht nachvollziehbar ist.

Als Grundlage für die Entwicklung eines Verfahrens werden daraufhin vollständige Verfahren zur Lösung ähnlicher Probleme klassifiziert und bewertet. Ausgangspunkt sind dabei insbesondere Backtrackingalgorithmen zur Lösung sogenannter Bedingungserfüllungsprobleme (engl. Constraint Satisfaction Problems). Diese Verfahren kombinieren in der Regel zwei komplementäre Lösungsstrategien. Zum einen können sie systematisch Suchen indem sie alle möglichen Kombinationen bilden und prüfen. Zum anderen können sie das Ausgangsproblem so transformieren, dass das resultierende Problem einfacher zu lösen ist. Die verschiedenen Verfahren unterscheiden sich deshalb insbesondere hinsichtlich der Frage, wie viel Aufwand für die systematische Suche und für die Problemtransformation aufgewendet wird. Die Effizienz der Verfahren hängt im Allgemeinen allerdings vom jeweiligen Problem am. In der Regel gilt jedoch, dass eine vollständige Problemtransformation ineffizienter ist als eine teilweise Transformation mit anschließender systematischer Suche. Neben der Lösungsstrategie lassen sich die Verfahren auch hinsichtlich ihrer Ausführung in sequentielle und parallele Verfahren aufteilen. Sequentielle Verfahren führen in der Regel zu einem geringeren Rechenaufwand, da parallele Verfahren inkonsistente Zustände während der Ausführung zulassen um Synchronisation zu vermeiden. Allerdings erfordern sequentielle Verfahren einen leistungsstarken Rechner, da die verteilte Ausführung eines sequentiellen Verfahrens aufgrund der Latenz entfernter Kommunikation zu langen Rechenzeiten führt.

Da die zu erwartenden Rechnersysteme aus einer Vielzahl ressourcenarmer Rechner bestehen, wird das parallele Verfahren namens Asynchronous Backtracking als Basis für die Konfiguration vorgeschlagen. Dieses Verfahren kombiniert die beiden Strategien der Problemtransformation und systematischen Suche in geeigneter Weise und erfordert nur wenig Synchronisation wodurch eine hohe Parallelität ermöglicht wird. Um dieses Verfahren einzusetzen muss das Konfigurationsproblem zunächst in ein verteiltes Bedingungserfüllungsproblem (engl. Distributed Constraint Satisfaction Problem) überführt werden. Hierfür wird eine Abbildung entwickelt, die keinerlei Vorabberechnung erfordert sondern direkt während der Ausführung des Backtrackingalgorithmus angewendet werden kann. Dadurch wird vermieden, dass der kombinatorisch sehr große Suchraum vollständig durchschritten werden muss - sofern dies nicht zur Bestimmung einer Lösung erforderlich ist.

Zusätzlich zur Abbildung der Konfiguration auf ein verteiltes Bedingungserfüllungsproblem muss das Ausgangsverfahren um Mechanismen erweitert werden, die den Einsatz in dynamischen Rechnersystemen ermöglichen. Dazu gehören insbesondere Vorkehrungen zum Umgang mit der schwankenden Verfügbarkeit einzelner Rechner. Anstatt die gesamte Konfiguration von neuem zu starten, werden bei einem Verbindungsabbruch zusätzliche Bedingungen erzeugt wodurch die Wiederverwendung abgeleiteter Bedingungen bei einem Neustart ermöglicht wird. Darüber hinaus lassen sich noch weitere problemspezifische Verbesserungen umsetzen, die insbesondere darauf abzielen, den Kommunikationsaufwand zu reduzieren. Die vorgeschlagenen Verbesserungen nutzen dabei aus, dass die Semantik der Bedingungen und ihre Struktur, aufgrund der Abbildung der Konfiguration auf ein Bedingungserfüllungsproblem, bereits von vorn herein bekannt sind.

4 Automatische Anpassung

Das in Kapitel 3 entwickelte Verfahren ist in der Lage, vor dem Start einer Anwendung eine ausführbare Konfiguration zu bestimmen sofern diese existiert. Da die Suche nach einer Konfiguration den Start der Anwendung verzögert, wird die Konfiguration, die zuerst gefunden werden kann, verwendet. Für den Fall, dass die Anwendung zur Laufzeit angepasst werden muss, kann das Verfahren zwar grundsätzlich unverändert wiederverwendet werden, allerdings muss bei der Anpassung einer laufenden Anwendung beachtet werden, dass unterschiedliche Konfigurationen unterschiedliche Kosten verursachen können. Eine Ursache dafür sind beispielsweise zusätzliche Verzögerungen, die durch den Transfer von anwendungsspezifischem Zustand beim Ersetzen einer Komponente auftreten können. Aus diesem Grund ist es vorteilhaft, im Fall einer Anpassung die Konfiguration hinsichtlich der Anpassungskosten zu optimieren.

Deshalb wird in Kapitel 4 das Verfahren zur Suche einer Konfiguration aus Kapitel 3 um eine Heuristik erweitert, mit der die resultierenden Anpassungskosten bereits während der Suche minimiert werden können. Die Vorgehensweise zur Entwicklung der Heuristik entspricht der Vorgehensweise in Kapitel 3. Zunächst wird das Problem an einem Beispiel verdeutlicht und auf dieser Basis formalisiert. Im Anschluss an die Formalisierung wird die resultierende Komplexität betrachtet. Dabei wird zum einen festgestellt, dass die tatsächliche Komplexität von den Eigenschaften der Kostenfunktion abhängt. Zum anderem wird aber auch verdeutlicht, dass die Komplexität bestenfalls der des Konfigurationsproblems entsprechen kann, da auch die Optimierung eine ausführbare Konfiguration finden muss. Im Anschluss an die Betrachtung der Komplexität werden die konkreten Anforderungen an die Optimierung diskutiert. Dabei wird verdeutlicht, dass die Suche nach einer Konfiguration mit minimalen Anpassungskosten mit der Forderung nach Effizienz in Konflikt steht. Im weiteren Verlauf der Arbeit wird der Effizienz eine höhere Priorität zugewiesen, da die die Minimierung der Kosten in der Regel mit dem Wunsch einer Effizienzsteigerung begründet wird.

Als Basis für die Entwicklung des Optimierungsverfahrens werden im Anschluss zunächst unterschiedliche Ansätze und Verfahren klassifiziert und bewertet. Die Klassifikation erfolgt dabei entlang der Garantien hinsichtlich der Optimalität, die vom jeweiligen Ansatz erreicht werden können. Vollständige Ansätze sind in der Lage immer eine der besten Lösungen zu finden. Allerdings erfordert dies einen Rechenaufwand, der in der Regel zu hoch ist. Beschränkte Optimierungsansätze reduzieren den Rechenaufwand indem sie lediglich garantieren, dass die Güte der gefundenen Lösung lediglich eine vordefinierte Abweichung von der Güte der besten Lösung abweicht. Heuristische Ansätze hingegen machen keinerlei Garantien hinsichtlich der Güte der Lösung, allerdings führen sie im Allgemeinen auch zum geringsten Rechenaufwand. Da der Rechenaufwand für die Anpassung einer Konfiguration den höchsten Stellenwert einnimmt, wird die heuristische Optimierung als Basis für das Verfahren ausgewählt.

Im Anschluss an diese Entscheidung werden die konkreten Kostenfaktoren, die bei der Optimierung berücksichtigt werden müssen, diskutiert. Dazu werden zunächst die möglichen Kostenfaktoren vorgestellt. Darauf hin werden die primären Kostenfaktoren in einem

Kostenmodell formalisiert. Da Komponenten, die nicht mehr verfügbar sind, in jedem Fall ersetzt werden müssen, erfasst das Kostenmodell insbesondere die Kosten, die durch die Ersetzungen von Komponente anfallen, die noch verfügbar sind. Die tatsächlichen Kosten die bei der Ersetzung einer Komponente anfallen lassen sich dann rekursiv über die Kosten aller Komponenten berechnen, die durch die Ersetzung ebenfalls ersetzt werden müssen. Die Kosten einer Komponente können sich dabei aus unterschiedlichen Faktoren ergeben. Beispiele hierfür sind der Umfang des Anwendungszustands der Komponente, der bei der Ersetzung transferiert werden muss, oder die Funktion der Komponente.

Nach der Vorstellung des Kostenmodells wird das Optimierungsverfahren selbst entwickelt. Grundsätzlich trifft das Verfahren Entscheidungen, die jeweils für sich gesehen lokal optimal sind. Diese „gierigen“ Entscheidungen müssen allerdings nicht notwendigerweise zu einer in globaler Hinsicht optimalen Lösung führen. Das in Kapitel 3 vorgestellte Verfahren zur Konfiguration bietet hierfür zwei komplementäre Ansatzpunkte, die sich im Rahmen einer Sortierungsheuristik von Werten und einer Sortierungsheuristik von Variablen ausnutzen lassen. Die Sortierungsheuristik für Werte zielt darauf ab, dass Komponenten, die wiederverwendet werden können, zunächst auch wiederverwendet werden. Die Sortierungsheuristik von Variablen zielt darauf ab, dass bei einer unabdingbaren Ersetzung von Komponenten zuerst die Komponenten ersetzt werden, die die geringsten Anpassungskosten nach sich ziehen. Zwar können beide Vorgehen in globaler Hinsicht suboptimal sein, z.B. kann die Ersetzung vieler Komponenten mit geringen Kosten teurer sein als die Ersetzung einer teuren Komponente, allerdings kann angenommen werden, dass diese Fälle seltener auftreten.

Während sich die Heuristik für Werte problemlos in das bestehende Verfahren integrieren lässt, erfordert die zweite Heuristik zusätzliche Vorkehrungen. Um sicherzustellen, dass die Korrektheit des ursprünglichen Verfahrens nicht verletzt wird, muss die Sortierung der Variablen global konsistent sein. Die Arbeit beschreibt hierfür wie die Sortierung ohne Vorabberechnung und ohne zusätzlichen Nachrichtenaustausch zwischen verschiedenen Rechnern umgesetzt werden kann. Die notwendige Ordnungsinformation wird als Bestandteil von Nachrichten versendet, die bereits für das ursprüngliche Verfahren notwendig sind. Durch dieses Vorgehen kann der zusätzliche Aufwand für das Optimierungsverfahren vernachlässigt werden, da er sich auf minimal umfangreichere Nachrichten und einfache lokale Sortieroperationen beschränkt.

5 Prototyp

Um die Konzepte und Verfahren, die in den Kapiteln 2 bis 4 entwickelt und präsentiert wurden, zu bewerten, wurden diese prototypisch implementiert. Neben dem Algorithmus, der sich aus den Kapiteln 3 und 4 ergibt, wurden auch das Komponentensystem aus Kapitel 2 vollständig implementiert. Zusätzlich wurden Entwicklungswerkzeuge und graphische Oberflächen für unterschiedliche Rechnersysteme entwickelt. Darüber hinaus wurden der Algorithmus aus Kapitel 3 und 4 sowie eine Reihe weiterer Algorithmen als Teil eines ereignisbasierten Simulators implementiert. Dadurch ist es zum einen möglich, das Gesamtsystem unter realitätsnahen Bedingungen zu bewerten und zum anderen ist es möglich, die Grenzen der Leistungsfähigkeit der Algorithmen umfangreich unter synthetischen Bedingungen zu bewerten.

Im ersten Teil von Kapitel 5 wird die Architektur des Gesamtsystems beschrieben. Dazu werden zunächst die architektonischen Bausteine und Schichten eingeführt. Auf dieser Basis werden daraufhin die Abhängigkeiten beschrieben. Schließlich wird die Funktion der einzelnen Bausteine im Detail beschrieben und abschließend wird deren Interaktion als Gesamtsystem betrachtet. Im zweiten Teil des Kapitels wird dann die Implementierung der Architektur auf Basis einer existierenden Kommunikationsmiddleware beschrieben. Da die Implementierung recht umfangreich ist, wird sie auf einem relativ hohen Abstraktionsniveau beschrieben. Deshalb werden in der Regel nur die die wesentlichen Teilfunktionen der architektonischen Bausteine sowie deren Interaktion kurz skizziert.

6 Evaluation

In den vorhergehenden Kapiteln wurden Konzepte und Verfahren zur automatischen Anpassung einer verteilten Anwendung entwickelt. Für die dabei aufgetretenen Problemstellungen wurden zielgerichtet, d.h. unter Berücksichtigung der problemspezifischen Anforderungen, Lösungen entwickelt. Um die zu erwartenden Eigenschaften nachweisen zu können wurde das Gesamtsystem prototypisch implementiert. Die Architektur und Implementierung dieses Systems wurde in Kapitel 5 skizziert.

Ziel von Kapitel 6 ist die Validierung der Eigenschaften der Lösungen und die Bewertung der entwickelten Verfahren auf der Basis der Implementierung. Dazu werden zwei komplementäre Methoden eingesetzt. Zum einen werden Messungen mit vergleichsweise leistungsschwachen Rechnern durchgeführt. Zum anderen werden Simulationen durchgeführt, die von den Eigenschaften der Rechner und der zugrundeliegenden Netzwerktechnologie abstrahieren. Mit Hilfe der Messungen werden die grundlegenden Eigenschaften hinsichtlich der Leistung und des Aufwands im Kleinen untersucht. Die Simulationen werden dazu verwendet, die Tendenzen im Verhalten der Verfahren im Großen zu bestimmen.

Als erstes werden die Abstraktionen und Mechanismen des Komponentensystems untersucht. In einem ersten Schritt wird eine Reihe von Messungen durchgeführt die die zusätzlichen Kosten erfassen, die aus den Abstraktionen und Mechanismen resultieren. Um die Ergebnisse der Messungen im Kontext zu betrachten, werden sie soweit dies möglich ist mit ähnlichen

Messungen auf Basis der zugrundeliegenden Middleware verglichen. Betrachtet werden dabei der Speicherbedarf sowie die Rechen- und Kommunikationskosten. Die Ergebnisse zeigen, dass sich die Kosten in vertretbaren Grenzen halten. In einem zweiten Schritt werden daraufhin die Vor- und Nachteile der Abstraktionen entlang einer umfangreichen Beispielanwendung diskutiert. Die Diskussion verdeutlicht, dass die Abstraktionen und Mechanismen die gesteckten Ziele erfüllen können.

Im Anschluss an die Bewertung des Komponentensystems, werden das Verfahren zur Konfiguration und die Heuristik zur Optimierung hinsichtlich ihrer Leistung bewertet. Dazu wird insbesondere der Kommunikationsaufwand des Konfigurationsalgorithmus über eine Vielzahl von Simulationen bestimmt. Die Ergebnisse dieser Simulationen werden mit alternativen Ansätzen verglichen. Zusätzlich wird die entstehende Verzögerung über eine Reihe von Messungen erfasst und mit alternativen Ansätzen verglichen. Wie erwartet zeigen die Simulationen und Messungen, dass das entwickelte Konfigurationsverfahren in der Tendenz einen exponentiellen Kommunikationsaufwand verursacht. Allerdings zeigen die Ergebnisse ebenfalls, dass das entwickelte Verfahren in einem breiten Spektrum unterschiedlicher Anwendungsszenarien problemlos eingesetzt werden kann. Dabei ist das Verfahren in vielen Fällen den alternativen Ansätzen vorzuziehen.

Abschließend wird die Leistung der Heuristik zur Optimierung der Anpassungskosten bewertet. Dazu wird eine Reihe unterschiedlicher Szenarien simuliert. Da der Rechen- und Kommunikationsaufwand, der durch den Einsatz der Heuristik entsteht, vernachlässigt werden kann, konzentriert sich die Evaluation hierbei auf die Anpassungskosten. Da die unterschiedlichen Szenarien stark voneinander abweichen müssen um aussagekräftige Ergebnisse zu erzielen, werden nicht die tatsächlichen Anpassungskosten als primäre Metrik verwendet sondern es wird eine abstrakte Metrik eingesetzt, die die Kosten im Kontext des Szenarios betrachtet. Dazu werden die entstandenen Kosten mit Hilfe der minimalen und maximalen Kosten eines Szenarios normiert. Die so gewonnen Ergebnisse zeigen, dass die Heuristik in vielen Szenarien in der Lage ist, eine gute Lösung zu finden. In Szenarien, in denen die Heuristik eine teure Lösung findet, ist es möglich, die Ergebnisse durch eine randomisierte Wiederholung der Ausführung signifikant zu verbessern. Falls notwendig kann so eine problemspezifische Abwägung von Optimierungsaufwand und Adaptationskosten umgesetzt werden. Allerdings wird dies in einem breiten Spektrum von Szenarien nicht erforderlich sein.

7 Verwandte Arbeiten

Nach der Entwicklung der Konzepte und Verfahren zur Anpassung einer verteilten Anwendung und der abschließenden Bewertung im vorangegangenen Kapitel, ordnet das Kapitel 7 die Arbeit und die Ergebnisse in den Rahmen der verwandten Arbeiten ein. Dazu werden diese zunächst exemplarisch vorgestellt und klassifiziert. Auf Basis dieser Klassifikation wird diese Arbeit mit ihren Ergebnissen eingeordnet.

Die Vorstellung verwandter Arbeiten beginnt mit einem kurzen Überblick über klassische Systemsoftware, die in herkömmlichen vernetzten Rechnersystemen eingesetzt wird. Dabei werden vorwiegend die Abstraktionen und Mechanismen herkömmlicher Middleware zur Kommunikationsunterstützung und Komponentensysteme vorgestellt. Im Gegensatz zur Zielsetzung dieser Arbeit zielen diese Systeme in der Regel nicht auf den Einsatz in spontan vernetzten Systemen aus ressourcenarmen Rechnern ab. Aus diesem Grund bieten sie keine Mechanismen zur automatischen Konfiguration und Anpassung der Verteilung und Parametrisierung einer Anwendung zur Laufzeit an.

Die Systemsoftware, die speziell mit Blick auf die Vision des Pervasive Computing entwickelt wurden, lassen sich grob in zwei Klassen aufteilen. Die erste Klasse betrachtet sogenannte intelligente Umgebungen (engl. Smart Environments). Intelligente Umgebungen sind räumlich begrenzte Bereiche, z.B. ein Raum oder eine Wohnung, in denen die integrierten Rechner kooperieren um die Aufgaben ihrer Nutzer zu unterstützen. Aufgrund der Ortsbindung intelligenter Umgebungen kann in der Regel vorausgesetzt werden, dass die Koordination der Rechner von einem fest installierten, leistungsstarken und ständig verfügbaren Rechner übernommen wird. Abhängig von der Lokation werden mobile Rechnersysteme dynamisch in die umgebende intelligente Umgebung eingefügt wodurch eine begrenzte Dynamik entstehen kann. Auch wenn die konkret genutzten Abstraktionen in diesem Bereich von denen in dieser Arbeit entwickelten abweichen können, so betrachten einige Arbeiten ebenfalls das Problem der automatischen Konfiguration und Anpassung. Die existierenden Lösungsansätze machen sich jedoch die ständige Verfügbarkeit eines leistungsstarken Rechners zu nutze. Dadurch können sie nicht eingesetzt werden, wenn ein solcher Rechner nicht verfügbar ist.

Die zweite Klasse betrachtet sogenannte intelligente Gruppen gleichgestellter Rechner (engl. Smart Peer Groups). Im Gegensatz zu intelligenten Umgebungen, die an einen festen Ort gebunden sind, können sich intelligente Gruppen bewegen. Begrenzt werden sie durch räumliche Nähe, die durch eine kommunikationsbasierte Metrik wie z.B. Hop-Count angenähert werden kann. Da sich diese Gruppen an jedem beliebigen Ort bilden können, kann nicht davon ausgegangen werden, dass ein leistungsstarker Rechner ständig verfügbar ist. Dementsprechend müssen sich die Rechner, die eine intelligente Gruppe bilden, gemeinsam verteilt abstimmen. Bislang beschäftigten sich Arbeiten in diesem Bereich überwiegend mit der verteilten Koordination auf Kommunikationsebene. Dementsprechend sehen nur wenige Ansätze Abstraktionen zur automatischen Konfiguration und Anpassung einer verteilten Anwendung vor.

Die Ansätze, die sich explizit mit der Konfiguration und Anpassung einer Anwendung beschäftigen, zielen nicht auf deren Automatisierung ab, sondern sie überlassen diese Aufgabe dem Anwendungsentwickler oder dem Nutzer. Wie bereits zuvor erwähnt, kann das zu einer erheblichen Belastung für den Entwickler bzw. den Nutzer werden. Im Rahmen dieser Arbeit wurde gezeigt, wie das vermieden werden kann, indem die Konfiguration und Anpassung der Anwendung vom System automatisiert wird.

8 Fazit

Das Kapitel 8 schließt die Arbeit mit einer kurzen Zusammenfassung der wichtigsten Erkenntnisse dieser Arbeit. Diese lassen sich als die Entwicklung neuer Konzepte und Verfahren zur automatischen Anpassung von verteilten Anwendungen zusammenfassen. Abgrenzend zu existierenden Konzepten und Verfahren zeichnet sich die in die Arbeit vorgestellten insbesondere durch ihre Eignung für dynamische, vernetzte Rechnersysteme aus, die aus einer Vielzahl ressourcenarmer, drahtlos vernetzter Rechner bestehen. Die Eignung und Leistungsfähigkeit der Konzepte und Verfahren wurde durch eine umfangreiche Evaluation mit Hilfe einer vollständigen Implementierung des Systems nachgewiesen.

Abschließend werden in Kapitel 8 eine Reihe angrenzender und erweiternder Forschungsthemen skizziert. Dazu gehören z.B. die Entwicklung einer ergänzenden Metrik zur Unterstützung proaktiver Anpassungen, die zum einem Zeitpunkt ausgeführt werden, zu dem die Anpassung noch nicht zwingend erforderlich ist. Ein weiteres Beispiel ist die Entwicklung von Konfigurationsverfahren, bei denen der Grad der Verteilung zur Laufzeit angepasst werden kann. Solche Verfahren könnten mutmaßlich deutliche Leistungssteigerungen erzielen, sofern die Verfügbarkeit eines ressourcenreichen Rechners gegeben ist.

1 Introduction

The chapter motivates the work presented in this dissertation and introduces its foundations by discussing the vision of Pervasive Computing. It identifies adaptive applications as a key enabling technique of Pervasive Computing. Based on this insight, different ways of supporting adaptation are discussed. Subsequently, the automation of adaptation using system software is proposed as a way to reduce the complexities that persons face when they are creating, operating or using pervasive applications. Thereafter, the chapter briefly reviews existing system software for pervasive applications before it identifies the gap closed by this dissertation. Finally, the chapter summarizes the contribution and describes the structure of the remaining chapters.

1.1 Background

Over the last decades, computer technology found its way into many areas of our daily life. Like many other technologies, computers started as incredibly expensive and room-filling tools. Over time, they became smaller and prices dropped. This, in turn, enabled their application to new areas in ways that could not be foreseen by their inventors.

A primary example is commerce. There, computers had a tremendous influence on the way people work. Before the emergence of computers, people were using type-writers to issue invoices that were calculated with pen and paper. Files of vendors and customers where manually archived in cabinets that occupied the best part of many offices. The time for retrieving a single file could easily be measured using an ordinary wristwatch as it often required minutes. With the advent of computer technology, people began to store their files on disks and the time for retrieval has been reduced from minutes to milliseconds. Over time, the file cabinets that previously occupied the offices simply disappeared.

Probably even more amazing than the sheer number of application areas for computer technology is the speed at which they were conquered. This can be attributed to a trend that emerged in 1965. In his famous article Gordon Moore notes that during the last ten years the integration density of micro-electronic circuits roughly doubled every year (Moore, 1965). He also states that there are no reasons to assume that this trend would come to an end during the next ten years. As it turned out, this prediction remains to be valid in 2008 and thus, it is not overly surprising that it made history as *Moore's Law*.

1.2 Pervasive Computing

The remarkable advances of computer technology are the foundation for a vision that was first formulated by Mark Weiser in 1991 known as *Ubiquitous Computing* or *Pervasive Computing*[1] (Weiser, 1991). The overall goal of this vision is to provide seamless and distraction-free support for the everyday tasks of persons through computer technology. To enable this, Pervasive Computing relies on a radical shift in the way in which people interact with computers. The impact of this shift is comparable to the advent of inexpensive personal computers. In contrast to the emergence of personal computers, however, Pervasive Computing does not focus on the computer technology per se. Instead, this vision focuses on the realization of a paradox, namely numerous computers that are both, ubiquitous and invisible.

To this end, Pervasive Computing envisions a multitude of miniaturized computers that are integrated into all kinds of everyday objects. Through this integration, computers become ubiquitous. Many of the computerized objects are equipped with inexpensive short-range wireless communication technology which, in turn, enables them to form networks. In combination with various embedded sensors, the objects are able to perceive their physical environment and they are able to share their perceptions with other objects. Together, these networks of computerized objects can receive a large number of their inputs through the unobtrusive perception of their surroundings. As a result, the interaction with individual objects is more intuitive since in most cases it does not encompass the use of special human-computer-interfaces like mice or keyboards. Due to this natural interaction, the integrated computers become practically invisible.

From a technical perspective, Pervasive Computing introduces a new class of networked computer systems that differs from traditional ones. In the following this class of systems is referred to by the term *pervasive systems*. The primary differentiating characteristics of pervasive systems are:

- *Heterogeneity*: Just like ordinary everyday objects, their computerized pendants are specialized and typically, they are only able to provide a predetermined set of functionality. As a result, many integrated computers are highly task-specific. Since these specialized computers co-exist with traditional computers, the span of computers that constitute the technical foundation of Pervasive Computing ranges from small, resource-poor and specialized embedded systems to clusters of resource-rich general purpose computers. This heterogeneity also extends to the communication technology

[1] In contrast to other definitions such as (Mattern, 2001), this dissertation does not draw a distinction between the terms Ubiquitous Computing and Pervasive Computing. To avoid confusion, the term Pervasive Computing is used consistently throughout the remainder of the dissertation.

leveraged in pervasive systems. While many embedded computers are solely equipped with inexpensive short-range wireless communication technologies such as Bluetooth (Bluetooth Special Interest Group, 2004) or IEEE 802.11 (Institute of Electrical and Electronics Engineers, 2003), more resource-rich ones leverage multiple technologies including long-range technologies such as GPRS via GSM (European Telecommunication Standards Institute, 2000).

- *Dynamics*: As many everyday objects can be moved by persons, a large number of computers encountered in pervasive systems are not fixed to a single location. Due to the mobility of computers and the utilization of short-range wireless communication technology, pervasive systems exhibit dynamic network topologies. In many cases, the degree of fluctuations depends on the mobility of the persons that leverage the integrated computers. In office scenarios, for instance, the rate of fluctuations is rather low, since the workers do not change their location frequently. Similarly, many of the computerized objects, such as projectors or whiteboards, found in these scenarios are fixed to specific location. In shopping scenarios, however, persons move relatively fast from one location to another. Thereby, they implicitly move their wearable and portable computers which, in turn, can cause drastic and unforeseeable changes to the connectivity. Thus, the degree of dynamics of pervasive systems spans the complete range from infrequent changes that only affect the quality of the wireless network links to frequent changes that cause disruptive changes to the network topology.
- *Evolution*: As a consequence of Moore's Law, most general purpose computers, e.g. personal computers and laptops, exhibit comparatively short life spans. At the time of writing, a large fraction of computers is replaced by newer models within two to five years, depending on their application area. In contrast to that, many everyday objects exhibit life spans that exceed the ones of computers by factors of more than four. According to a study of the Australian government (Environment Australia, 2001), the average life span of refrigerators lies between ten to twenty-five years. Similarly, the freezers and air conditioning systems have life spans of twenty years and above. It is conceivable that the integration of computers into such objects must not significantly lessen their life spans as this will most likely not be tolerated by consumers. Thus, pervasive systems will consist of computers that represent four or five different computer generations which requires appropriate precautions to cope with the unforeseeable continuous evolution of everyday objects and computers.

The seamless and distraction-free support for everyday tasks envisioned by Pervasive Computing relies on applications that combine different sets of functionality. As the computerized objects residing in pervasive systems are specialized, applications need to combine the functionality of multiple objects to provide intuitive and natural task support. This leads to the conclusion that the applications encountered in pervasive systems are inherently distributed. The remainder of this dissertation refers to such applications that combine the distinct functionality of different computers in a pervasive system as *pervasive applications*.

1.2.1 Adaptation

The dynamics and heterogeneity of pervasive systems as well as the fact that pervasive applications are necessarily distributed raises an inherent need for adaptation. In order to function properly despite the changing network link quality and topology, applications need to adapt to the overall system properties and the available capabilities that can be leveraged at a certain point in time during their execution. If a computerized object that is used by an application is no longer reachable over the network because it has been moved by a person, the application must somehow react to this fact. Similarly, if the network link quality worsens significantly, the application must change its behavior.

It is noteworthy that adaptation can be avoided – at least to certain extend – by reducing the requirements of an application on its execution environment. As this approach reduces the dependencies of the application towards external factors, the number of cases that require the application to change its behavior can be reduced. For instance, if distribution can be avoided completely, an application can be executed independently from the properties of the network. In many cases, however, the reduction of the application requirements leads to a significant loss in quality. Since Pervasive Computing strives for distraction-free support of everyday tasks by combining the distinct functionality of various integrated computers, reducing the external dependencies is frequently not desirable and in some cases impossible. In fact, Pervasive Computing demands applications that leverage the available integrated computers as far as possible to achieve the goal of invisibility.

In general, the adaptation of a distributed application can be classified along three orthogonal dimensions. The first dimension classifies the adaptation of an application regarding the point in time when an adaptation takes place. In this dimension, the following two classes can be identified:

- *Proactive*: Proactive adaptation denotes a modification to an application that is performed *before* an application can no longer be executed. Thus, proactive adaptation tries to *avoid* application failures. To enable this kind of adaptation, the entity that controls the adaptation process needs to have a priori knowledge that can be used to avoid situations that lead to failures. In cases where such knowledge is not available, a prediction heuristic that provides a close estimate of the future situation can be used as substitute. An example for a system that performs proactive adaptation is the GSM network. There, mobile phones perform a cell hand-over before they can no longer contact their current cell. To do this, they change their current cell whenever they detect another cell with a higher signal quality.

- *Reactive*: Reactive adaptation denotes a modification to an application that takes place at a point in time when the application can no longer be executed. As such, reactive adaptation *fixes* an application *after* it has experienced a failure. To enable this kind of adaptation, the entity that controls the adaptation process needs to be able to detect failures whenever they occur. Examples for reactive application adaptation can be found in Internet-based video phone applications. Typically, these applications reduce the audio and video quality and thus, the amount of data that needs to be transferred whenever the available bandwidth drops below the required bandwidth.

The second dimension classifies adaptation depending on the type of modification applied to the communicating processes that constitute the distributed application. In this dimension, the following two classes can be identified:

- *Parameterization*: Adaptation by parameterization modifies the behavior by changing parameters of a subset of the functionality that constitutes the distributed application. To do this, the functionality that can be adapted defines a set of parameters and ranges for their values. Depending on the values, the functionality changes its behavior. A premier example of adaptation by parameterization can be found in multimedia play-back applications that transcode the multimedia content depending on the properties of the network connection. These applications can for instance reduce the required network bandwidth by increasing the compression applied to the media. This way these applications can dynamically balance the bandwidth requirements and the amount of processing required to render the media.
- *Reconfiguration*: Adaptation by reconfiguration modifies the behavior of an application by changing the structure or the distribution of the functionality that constitutes the application. A simple form of this kind of adaptation is an isomorphic transformation of the application by migrating a number of processes. Examples for this type of adaptation are web server replicas that use a load balancer to dispatch incoming requests for web pages to the server with the least load. More complex transformations can for instance split functionality into a number of individual parts or they can collapse a number of parts into one. To enable this kind of adaptation, the application must exhibit an adequate structure that supports multiple mappings to computers and enables the migration of all application-specific state.

The last dimension classifies the adaptation of a distributed application based on the control of the adaptation process. The control of the adaptation process can be classified into the following two categories:

- *Manual*: Manual adaptation is performed manually by a person. To support this type of adaptation, the person that controls the adaptation process must be supplied with a mental model of the application together with a set of possible modifications. Furthermore, the person must be able to perceive relevant properties of the overall system to make proper adaptation decisions.
- *Automatic*: Automatic adaptation is performed by the application without user intervention. To automate the adaptation control, the application must possess the same type of information that is required for manual adaptation. In order to perform adequate adaptation decisions, the application must select a possible modification based on the system properties. In addition to manual adaptation, however, automatic adaptation requires a strategy that selects the modifications.

1.2.2 Automation

The potentially high dynamics of pervasive systems as well as the goal of providing distraction-free support for tasks, limit the applicability of manual adaptation. In scenarios that are highly dynamic, adaptation must be performed frequently. Thus, if adaptation is performed manually, the person interacting with an application must frequently shift the attention from completing the supported task to performing adaptation decisions. Additionally, for technically unversed persons acquiring a mental model of the low level details of a distributed application – to an extend that allows educated decisions – might be an impossibly hard task.

To avoid the problems resulting from manual adaptation, the responsibility for adaptation can be pushed into the application. The application developer must then programmatically deal with all changes in the pervasive system that affects the application during its execution. To do this, the application developer must provide additional functionality to detect all relevant changes and determine an appropriate action that deals with each individual change. Depending on the type of change there might be multiple actions that can be taken. Thus, determining an action entails computing possible actions and selecting the most appropriate one.

Pushing the responsibility for adaptation into the application without further precautions complicates the task of application development. Selecting an appropriate action in response to a change requires reasoning about effects of the action on the execution environment of the application. Due to the dynamic and heterogeneous nature of pervasive systems as well as the fact that these systems evolve continuously, it is practically impossible to foresee all possible execution environments of an application at development time. Thus, this reasoning must be performed programmatically at runtime. Additionally, if adaptation is handled individually by each application, every application must implement its own set of monitoring mechanisms to detect relevant changes. Since such mechanisms consume system resources like memory, processing power, or network bandwidth, their duplicate implementation introduces undesirable overhead if multiple applications are used simultaneously.

In order to mitigate the drawbacks arising from automatic adaptation while achieving many of its benefits, the responsibility for adaptation can be pushed into adequate system software. This approach has been applied successfully to the domain of distributed multimedia applications (Dermler, 1999) and it has been proposed as a general solution to reduce the complexity of application deployment and maintenance in distributed software systems (Arshad, Heimbigner, & Wolf, 2003). Enabling the system software to adapt an application automatically requires explicit application knowledge. Specifically, the system software must be able to detect the changes that affect the execution of an application. Furthermore, the system software must be able to determine possible adaptations. If multiple adaptations can be performed in response to a change, the system software must be able to evaluate them in order to select the most appropriate.

1.3 System Software

In the past, a number of software systems have been proposed by various researchers that ease the development of pervasive applications. They can be classified depending on the underlying conceptual model of what constitutes the pervasive system. At the present time, two models – namely smart environments and smart peer groups – can be identified. The organization of the individual system software solutions and the provided support with respect to application adaptation depends heavily on the utilized model.

1.3.1 Smart Environments

Early system software solutions for pervasive systems such as (Garlan, Siewiorek, Smailagic, & Steenkiste, 2002), (Roman, Hess, Cerqueira, Ranganathan, Campbell, & Nahrstedt, 2002), and (Johanson, Fox, & Winograd, 2002) focused on supporting the concept of smart environments. A smart environment is thereby defined as a spatially limited area, e.g. a meeting room or an apartment, equipped with various sensors, actuators and computers. Computers in a smart environment can either be mobile – in which case they are dynamically integrated into the environment depending on their location – or they can be immobile – in which case they are continuously part of the same surrounding environment.

In order to simplify application development, system software for smart environments typically provides a set of basic services for the applications executed in their spatial area. Common services include authentication of computers, access control to resources, unified access to persistent storage, and management of data captured by the sensors of the environment. To provide further support for application development and administration, some systems additionally introduce abstractions used to structure an application. These range from basic service abstractions that enable the transparent usage of different implementations of the same functionality within an application up to comparatively complex component abstractions that support automated application adaptation.

To facilitate automatic adaptation, the component abstractions separate the task of application development into low-level component development and high-level application composition. Component development is then typically supported through a general purpose programming language and an adequate programming framework while composition is usually supported by a custom scripting language. At runtime the system software interprets the script and wires the specified components according to the defined composition. Depending on the design of the scripting language and the component abstraction, this approach can be used to automate the initial deployment of an application in a smart environment, e.g. by selecting the computers and components used within an application, and it can also allow the runtime adaptation of an application, e.g. by dynamically replacing or migrating individual components. Apart from fostering automatic adaptation, the separation of component development and application composition can also be used to empower persons with limited programming experience to create their own applications – given that the scripting language is kept simple or that adequate development tools are available.

Independent from provided services and the utilized abstractions for application development, most system software for smart environments relies on a single computer to provide the basic services and to coordinate the execution of applications. Since the services are used by most – if not all – applications executed in the smart environment, the presence of the coordinating computer is required at all times. If this computer is not present or if it fails, e.g. due to some hardware failure, no applications can be executed in the environment. Thus, the computer responsible for providing the services must be immobile and it must guarantee a high availability. As a result, most system software for smart environments assumes the presence of a dedicated reliable and resource-rich general purpose computer in a smart environment. While this assumption can greatly simplify the development of system services and applications, it suffers from a number of technical and economic drawbacks.

Without further precautions, the coordinating computer introduces a single point of failure and it limits the spatial area that can be covered by one smart environment. Clearly, managing a single meeting room with tens of integrated computers and a handful of running applications should not be a problem for current personal computers. However, trying to manage a complete office building with hundreds of computers and applications can easily overload the resources of a single coordinating computer. Apart from technical availability and scalability issues, relying on a dedicated computer also raises a number of economic issues that can limit the applicability of the concept of a centrally managed smart environment. Obviously, requiring the presence of a resource-rich coordinating computer in each smart environment introduces monetary costs. Since this computer is a single point of failure for the smart environment, the hardware and software running on the computer must be able to guarantee a high availability which increases the cost. Depending on the purpose of the smart environment, achieving the desired availability might even require costly server technology, e.g. redundant hard disks or processors and memory supporting hot replacement. In addition to the initial cost of the computer, this approach also introduces cost that incur continuously. Most notably the computer used for coordination must be administered properly. While this is most likely not a problem in business environments with technical staff that already administers other infrastructure, it can introduce an undesirable hurdle if the smart environment is utilized non-commercially, e.g. in a home environment, where persons do not exhibit an appropriate level of technical expertise.

1.3.2 Smart Peer Groups

In order to mitigate the limitations of smart environments, researches developed the concept of smart peer groups (Schiele, 2007). In contrast to smart environments that view a pervasive system as a set of computers located in a fixed and predefined spatial area, smart peer groups postulate a people-centric perspective on pervasive systems. The key idea is to view a pervasive system as the dynamic collection of computers that surrounds a person independent from the person's current physical location. Technically speaking, smart peer groups differ from smart environments with respect to the following four characteristics:

- *Networking*: In order to support mobility, the utilized networking technology is typically wireless and supports short-range communication with a comparatively high bandwidth that is free of charge. Current examples of such networking technology are IEEE802.11 and Bluetooth. These technologies enable the transmission of multiple megabits of data per second using a freely available band. It is noteworthy that computers of a smart peer group can be equipped with multiple communication interfaces at the same time.
- *Self-organization*: The computers forming a smart peer group are self-organizing. Specifically, this means that the computers do not have to be configured to participate in a group. Instead, they are capable of forming a group dynamically without manual intervention. They allow persons to use the functionality provided by individual computers of a group in an intuitive manner which reduces the required technical expertise to a minimum.

- *Cooperation*: The computers in a smart peer group cooperate spontaneously without requiring any additional technical infrastructure. From a technical point of view, this is probably the main differentiating characteristic of smart peer groups when compared to smart environments. Through spontaneous cooperation, computers can use the functionalities of others everywhere and at any point in time. Clearly, spontaneous cooperation requires adequate mechanisms to detect the computer systems that are close-by and to integrate them into a group.
- *Proximity*: Smart peer groups use physical proximity to create the necessary locality required to keep the unstructured and dynamic network manageable. The key assumption thereby is that persons are in general more interested in functionality that can be provided by computers close-by. Clearly, this assumption conflicts with the traditional goal of computer networks since they aim at hiding physical distance whenever possible. Yet, many – if not most – computers in a pervasive system are integrated into physical objects. Thus, there is a natural locality resulting from a reduced usefulness of distant objects. As an example, consider for instance a person that uses a display to view a document. In this – and many other scenarios – the person does not want to use a display that resides in another building or city. Instead, the person is interested in using one in viewing distance.

At the time of writing, there exists only one system software that specifically focuses on support for smart peer groups (Becker, Schiele, Gubbels, & Rothermel, 2003), however, there are various other solutions that can be adapted to this model (Aitenbichler, Kangasharju, & Mühlhäuser, 2005), (Grimm, 2004). Current system software for smart peer groups focuses mainly communication support. Towards this end, the solutions offer fundamental services such as device and service discovery as well as basic service abstractions used to unify access to the functionality available in a group. In order to communicate with other computers in a smart peer group, these systems support various communication paradigms, including message passing, remote method calls, and publish-subscribe-based event and data dissemination. Adaptation is typically supported at the communication layer but not at the application layer. BASE (Becker, Schiele, Gubbels, & Rothermel, 2003), for instance, can support multiple communication technologies on the same computer. If a remote computer can no longer be reached using one technology, BASE can dynamically switch to another technology even within an ongoing interaction. By adapting the communication, BASE can hide fluctuating network properties as long as an alternative network is available. If a computer is no longer reachable at all, BASE does not provide any further support for application developers.

Since the definition of a smart peer group is not bound to a specific location but is rather given in terms of physical proximity, this conceptual model for pervasive systems cannot rely on the permanent presence of a single computer. In principle, each computer might leave or enter a group at any point in time. As a result, the tasks performed by the system software are organized in a completely decentralized manner. While this approach naturally does not lead to a single point of failure, it can have a negative impact on the overall efficiency. In practice, however, one can assume that many groups will consist of a rather static core of computers that is formed by the mobile computers such as PDAs, mobile phones, and laptops carried by persons. This observation can be used to increase the efficiency of system services for instance by dynamically clustering the computers that form the core of a smart peer group. In the past, this approach has been applied successfully to increase the energy efficiency of a discovery service for smart peer groups (Schiele, Becker, & Rothermel, 2004).

By avoiding the use of a central resource-rich and immobile personal computer, smart peer groups are well suited to support applications that continuously support persons during their daily journeys. Furthermore, since they are self-organizing and do not require any coordinating computer, smart peer groups can execute applications everywhere and they are more resilient to failures of individual computers. Also they do not require persons to invest in hardware that does not provide application-specific functionality. Similarly, persons do not have to administer computers apart from those that they use for their applications. This has the potential to greatly speed up the overall adoption of pervasive applications. Right from the start, smart peer groups can cost-effectively support isolated application scenarios that are highly useful for persons. If at a later point in time more embedded computers become available, existing applications can make use of them and new applications, whose additional value would not justify the investment in computer and network technology, can be introduced. Clearly, in some cases investing in a coordinating computer might be desirable in order to increase the performance and efficiency of the pervasive system. Likely examples are business scenarios where people with the required level of expertise and administration skills are present already. For many non-commercial application scenarios such as home automation, however, this prerequisite does not hold true. In these scenarios, relying on a smart peer group as conceptual basis for pervasive systems is a more suitable approach.

1.4 Motivation

As pointed out in the previous sections, pervasive applications aim at easing our daily live by unobtrusively leveraging the cooperative capabilities of integrated networked computers. Induced by their inherent heterogeneity and dynamics as well as their continuous evolution, pervasive systems pose new challenges to application developers. The two main challenges that differentiate application development for pervasive systems from application development for traditional networked computer systems are dynamic integration of functionality provided by multiple computers and continuous adaptation to the ever-changing sets of available functionality.

Smart peer groups provide a conceptual model for pervasive systems that is worthwhile investigating from an economic as well as a technical point of view. While there are existing system software solutions for smart environments that provide suitable abstractions to overcome the dynamics, the current solutions for smart peer groups solely focus on adaptive communication support. Clearly, enabling adaptive communication between heterogeneous networked mobile computers is a mandatory step to enable the development of applications running in a smart peer group. Yet, providing only communication support and rudimentary service abstractions requires application developers to programmatically configure and adapt applications to the changing set of available functionality. In order to avoid this overhead, system software for smart peer groups must go one step further and automatically determine and adapt the composition of functionality used by a pervasive application.

Supporting automatic configuration and adaptation with system software requires abstractions for application development that go beyond services. More specifically, the utilized abstractions must not only model the functionalities available in a smart peer group, but they must capture the requirements of an application. Using adequate abstractions, system software can then provide algorithms that automatically determine possible application configurations. Furthermore, it can provide mechanisms to manage the execution of a specific configuration and it can provide mechanisms to support the automatic adaptation from one configuration to another.

From an application developer's perspective, such system software would ideally create the illusion of a static system that is preconfigured and does not change at runtime. For obvious technical reasons creating such an illusion, however, is a hard – if not impossible – undertaking in practice. Thus, designing system software for smart peer groups that supports adaptive pervasive applications entails compromises between adaptation transparency, efficiency and usability. Yet, even if the system software is not capable of fully hiding the dynamics of pervasive systems, it can still hide many details which can greatly simplify application development.

1.5 Scope and Focus

As briefly hinted in Section 1.3 and detailed in Chapter 7, there are various system software solutions that ease the development of pervasive application. Current solutions that facilitate automatic application adaptation have been specifically designed to meet the challenges encountered in smart environments. Even though these solutions are all targeted at the same class of pervasive systems, they frequently offer completely different programming abstractions. One of the key reasons for this results from the fact that they are targeted at supporting different types of pervasive applications.

Approaches like (Nam, Shin, Hur, & Han, 2007), for instance, are targeting applications that control the actuators present in a smart room using the inputs of embedded sensors. Thereby, they model an application as a set of event-condition-action rules residing on the coordinating computer. If a sensor recognizes that a person has entered the room, the coordinating computer is notified and it evaluates the conditions of installed rules. If the condition evaluates to true, the corresponding action is triggered. The action might then power a motor that closes the jalousies attached to the windows of the room.

Other approaches like IROS (Ponnekanti, Johanson, Kiciman, & Fox, 2003) view a pervasive application as an orchestrated set of traditional applications. To support their integration, IROS provides a central communication facility that enables an application to post events and to listen to events. By augmenting a number of traditional (and usually non-distributed) applications with scripts, this communication facility can be used to create the desired integrated behavior in a smart room. As an example, consider a presentation application that posts an event when a presentation is started or stopped. By listening to these events, a light control application running in the room can automatically dim the light when the presentation application is started and it can brighten the light when the presentation ends.

Finally, approaches like GAIA (Roman & Campbell, 2000) propose a relatively generic component model that separates the task of combining an application from components and the task of mapping components to individual computers in a smart environment. An application consists of a script that describes the composition of components and another script that describes the placement of the components. Using these scripts a central coordinating computer can start and stop the distributed component-based application and it can deliver additional services to unify the access to resources that are required by all components, e.g. access to persistent storage.

These three examples for system software clearly show that the application support is heavily dependent on the targeted application model. Thus, depending on the type of application that must be implemented by a developer, system software can either be well-suited or ill-suited. Implementing a complex application on the basis of event-condition-action rules, for instance, might lead to numerous rules that trigger each other in an unforeseeable way leading to software that is hard to program and debug. Yet, relying on multiple scripts as in GAIA in order to program a very simple application for a specific environment might introduce setup and configuration overhead that may very well exceed the effort for developing the application without any system software support at all.

Similarly, the notion of adaptation varies heavily depending on the application model. The abstractions introduced in IROS, for example, render adaptation an implicit process that does not have to (and cannot) be managed by the system software. By completely decoupling individual applications, each application can operate independently and an arbitrary number of applications can react to a single posted event. In contrast to that, the component abstraction provided by GAIA requires explicit measures to adapt an application. If some required component cannot be executed or if it fails, the application needs to be adapted explicitly in order to work properly.

At first glance, abstractions that foster implicit adaptation might seem preferable as they offer a high degree of flexibility and they are seemingly more robust. However, this perception is only true for a certain type of applications for which cooperation is an option but not a requirement. If coordinated interaction is required to achieve a desired application behavior, these abstractions can only provide very limited support for application developers. In such cases, abstractions that model the interacting parts explicitly can provide more thorough support, for example, by signaling that some part is currently not available or by replacing one part with another.

In summary, this leads to the conclusion that it is essential to define the class of applications that should be supported by system software. The work presented in this dissertation focuses exclusively on applications that require the coordinated interaction of a distributed set of functionalities. The key rationale for this deliberate focus results from the fact that such applications repeatedly pose a hard challenge for application developers. Namely, each application that requires the coordinated interaction of multiple functionalities will need to manage their exact composition continuously to cope with the dynamics of pervasive systems.

As a consequence, system software that tackles this challenge on behalf of the developer can greatly simplify application development. In addition to that, support for applications for which the interaction of multiple functionalities is optional can easily be layered on top, since they pose less restrictive requirements on the underlying system software.

1.6 Contribution

The contribution of this dissertation is the design and evaluation of an integrated component system for smart peer groups that eases application development by automating the initial configuration and runtime adaptation of pervasive applications. In contrast to other approaches that aim at automating the adaptation of distributed applications, the presented approach is fully distributed and does not rely on the availability of a resource-rich computer. Instead, it leverages the parallelism inherent in smart peer groups by relying on a distributed configuration and adaptation algorithm that operates asynchronously.

As base for automation, the dissertation introduces a lightweight component and resource abstraction as well as a generic application model to capture the requirements of applications and to describe the capabilities of different computers in a pervasive system (Becker, Handte, Schiele, & Rothermel, 2004), (Handte, Schiele, Urbanski, Becker, & Rothermel, 2005). The explicit knowledge of system capabilities and application requirements is then used to automatically compute an application configuration that satisfies all requirements. To determine such a configuration, the dissertation proposes an algorithm that can find configurations even in the presence of strictly limited resources (Handte, Becker, & Rothermel, 2005). To automate the runtime adaptation of an application, the dissertation additionally proposes a simple yet powerful model that captures the cost for adapting a running configuration (Handte, Herrmann, Schiele, Becker, & Rothermel, 2007). Furthermore, it presents a heuristic approach towards adaptation that aims at minimizing the adaptation cost without increasing the runtime overhead of the algorithm for initial configuration.

To demonstrate the applicability of the overall concepts, the dissertation presents a prototypical implementation of the system software as a component system running on top of BASE (Becker, Schiele, Gubbels, & Rothermel, 2003), (Handte, Becker, & Schiele, 2003), a communication middleware for smart peer groups. Furthermore, it discusses an exemplary application (Handte, Urbanski, Becker, Reinhardt, Engel, & Smith, 2006) that has been built using the component system and compares the proposed configuration and adaptation algorithm with alternative approaches. The comparison shows that the proposed approach is preferable in a broad spectrum of possible future scenarios.

1.7 Structure

The remaining chapters of the dissertation are structured as follows. Chapter 2 presents the design of the component system used to support the automatic adaptation of pervasive applications executed by smart peer groups. To motivate the design rationale of this system, the chapter derives the requirements on system software for pervasive applications that enables the automatic adaptation of applications. Thereafter, the chapter introduces the abstractions provided by the proposed system software and discusses why they are needed.

Chapter 3 describes the algorithm that computes the initial configuration of a pervasive application. Thereby, the chapter starts with a problem formalization of the configuration problem and shows that the overall problem of finding a single configuration in the presence of strictly limited resources is NP-complete in general. The chapter derives the requirements on automatic configuration of pervasive applications and discusses a set of candidate algorithm classes with respect to their suitability for solving the configuration problem. Based on this discussion, the chapter describes how an algorithm of the most suitable class can be modified to find an initial configuration.

Chapter 4 describes heuristic extensions to the configuration algorithm that aim at minimizing the adaptation cost without adding runtime overhead. Initially, the chapter formalizes the adaptation problem and introduces a simply yet powerful cost model to capture the cost for modifying a configuration. Subsequently, the chapter derives the requirements on algorithms for automatic adaptation and discusses possible algorithmic solutions. Finally, the chapter presents a heuristic extension to the configuration algorithm presented in Chapter 3 that aims at minimizing the adaptation cost without introducing additional communication.

Chapter 5 presents the overall architecture and some implementation details of a prototype realization of the component system described in Chapter 2 as well as the configuration algorithm and adaptation heuristics presented in Chapter 3 and 4. The prototypical realization allows the evaluation of the basic concepts of the system and it facilitates comparisons of alternative approaches to automatic configuration and adaptation.

Chapter 6 evaluates the system software and the associated configuration and adaptation algorithm presented in Chapter 2, 3, and 4. It starts with a discussion of the overheads introduced by the system software. Thereafter, it discusses the benefits and limitations of the approach using a realistic application that has been built with the system. Finally, the chapter evaluates the proposed configuration and adaptation algorithm using event-discrete simulations and real-world experiments. In order to restrict the parameter space of the simulations, the relevant parts of the parameter space are derived from the previously presented applications and the assumptions that have been introduced in Chapter 2.

Chapter 7 provides an overview of existing system software and discusses similarities and differences to the approach taken by this dissertation. To provide a more complete overview, the chapter first outlines system support for traditional distributed applications. Thereafter, the chapter provides a detailed comparison of existing system software for pervasive systems. Thereby, the chapter classifies the software depending on its requirements on the execution environment into system software for smart environments and system software that can be used in smart peer groups.

Finally, Chapter 8 concludes the dissertation by summarizing the major findings and contributions. In addition, the chapter also describes a number of possible enhancements and future research directions.

2 System Software

This chapter describes the overall design of a distributed component system that enables the automated configuration and adaptation of pervasive applications (Becker, Handte, Schiele, & Rothermel, 2004). To motivate the design, the chapter derives general requirements on system software that must be fulfilled to enable automatic application configuration and adaptation at the system level. Thereafter, the chapter describes the high-level design rationale and the individual abstractions introduced by the system. Finally, the chapter briefly outlines why and how the abstractions fulfill the requirements. A more detailed analysis is presented in Chapter 6.

2.1 Requirements

Using the smart peer group model presented in the previous chapter and the overall goal of supporting automatic application adaptation at the system level, we can derive a number of requirements that must be fulfilled by system software for pervasive systems. Based on their origin, we can classify them in requirements resulting from the goal of suitability for smart peer groups and the goal of supporting automatic adaptation. The first two requirements, namely minimalism and extensibility, and decentralized coordination result immediately from the smart peer group model. Whereas the last three requirements, namely flexible explicit application specification, continuous application monitoring, and high adaptation transparency result from the goal of automating adaptation at the system level.

2.1.1 Minimalism and Extensibility

Smart peer groups consist of heterogeneous networked computers ranging from resource-poor sensors integrated into small objects like pens and wrist-watches to resource-rich general purpose computers like laptops and personal computers. In order to support the resource-poor computers of a smart peer group as well as the resource-rich ones with one particular system software, the system software must be minimal with respect to its resource requirements. Since a system software per-se does not provide application-specific functionality, resources of interest are thereby mainly the required network bandwidth, the dynamic and static memory used to execute the system software and to store its binary image, and the consumed processing power. To keep the resource requirements minimal, the core of the system software should only introduce abstractions that are required by all applications. Introducing only a minimal number of abstractions has the additional advantage of reducing the learning effort for application developers. Clearly, relying solely on a minimal set of abstractions might lead to system software that cannot make use of the capabilities of resource-rich computers. Thus, the system software must be extensible in order to utilize the power resulting from the availability of resource-rich computers. By being both, minimal and extensible, the system software can be tailored to the individual capabilities of the target computer and provide the required functionality with minimal resource overhead.

2.1.2 Decentralized Coordination

A smart peer group is a dynamic collection of networked computers in which the continuous availability of single computer system cannot be guaranteed. In fact, any computer of a smart peer group may become unavailable at any point in time. While there is a body of research on predicting the future unavailability of a networked computer system based on the current signal quality of its wireless link, the utilized predictions are typically inaccurate. This is especially true for indoor scenarios with obstacles causing shadowing effects. In such scenarios, solid walls or steel beams embedded in the ceiling can lead to immediate disconnections of a mobile computer. Without a complete a prior knowledge of an environment and the movement of the mobile computer, such disconnections cannot be predicted. Therefore, the system software must be able to deal with the unavailability of any computer that is part of the group.

Apart from that, the smart peer group model also does not guarantee the availability of a single resource-rich computer. In many scenarios, groups are formed by a number of resource-poor computers that need to share their functionality cooperatively in order to create the desired application behavior. As an example consider a group of travelers that schedules a meeting using their mobile phones and personal digital assistants. Thus, system software for smart peer groups cannot rely on a resource-rich computer that centrally coordinates the interaction of the participating computers.

To deal with the dynamics as well as with the possible unavailability of a resource-rich computer, system software must coordinate the interaction of computers forming a smart peer group in a decentralized manner. Clearly, in some scenarios where resource-rich computers are available, one computer could be selected as coordinator in order to increase the performance. Yet, since this performance optimization is not possible in all scenarios, system software that supports centralized coordination must always be able to switch to decentralized coordination if no resource-rich computer is available.

2.1.3 Flexible Explicit Application Specification

In order to enable automatic adaptation at the system level, the system software needs to be able to reason about possible application configurations. To do this the system software needs to have explicit knowledge about the capabilities of the computers that participate in a group. Furthermore, the system software also needs to have explicit knowledge about the requirements of an application with respect to these capabilities. By explicitly specifying application requirements and computer capabilities in such a way that the system software can match them automatically, the system software can determine an application configuration that can be executed in a group without requiring manual intervention or programming.

Thereby, it is important to realize that the utilization of a specific capability usually requires resources on the computer that provides it. For instance, using an mp3-player to stream music to some wireless headphones requires memory and processing power on the player. Naturally, such resources are limited by the design of the underlying internal hardware and the external devices attached to the computer. Thus, in order to enable the system software to reason about the limitations of individual computers, the explicit knowledge of the capabilities must be able to capture available and required resources imposed by using them. Without knowing the limitations of individual resources, the system software cannot decide whether a certain configuration results in the desired behavior or whether it simply overloads the available resources which can often cause undesirable side-effects. Overloading the computational resources in the previous mp3-player example might for instance cause distracting interruptions in the audio playback since the player is no longer able to meet the deadlines for decompressing the mp3.

Apart from capturing resource utilization and limitations, the application specification must be flexible to cope with the dynamics, heterogeneity and the continuous evolution of pervasive systems. Since different computers might be able to provide the same or a similar functionality in different ways, the requirements specification must support varying degrees of freedom in order to execute an application whenever possible. As a consequence, it should not define restrictions that are not necessary to ensure the correct operation and it should support a high compositional flexibility to support different and continuously evolving execution environments.

As a simple example, the application specification should not require the availability of a certain computer if another one could be used instead. Clearly, for some applications it might be necessary to define the exact computer that must provide a specific functionality. In most cases, however, it is more desirable to capture the application requirements in terms of functionalities with desirable properties rather than in terms of identities. If an application, for instance, requires a large display to show some information, the application specification should rather describe the properties of the display, e.g. the required resolution and physical size, than its identity. This allows persons to use the application in all smart peer groups that contain an appropriate display whereas relying on the displays identity would limit the application to groups containing that specific display.

In summary, system software for pervasive applications requires an explicit application specification to reason about different configurations and possible adaptations. However, in order to support the distraction-free execution in vastly different, continuously evolving execution environments, the specification must be flexible and it must support a high degree of compositional flexibility without endangering the correctness of the resulting application.

2.1.4 Continuous Application Monitoring

Due to the inherent dynamics of smart peer groups, the set of available functionality is continuously fluctuating. The functionality available on a certain computer becomes unavailable to other computers of its smart peer group as soon as the computer leaves the group. Apart from such fluctuations that are a result of the short-range wireless network technology used by computers of a smart peer group, certain functionality can also become unavailable if the resource availability on a computer changes. If a person, for instance, manually disconnects an external resource, e.g. a display or a hard drive, the available resources on the affected computer change. Such a change can make it impossible to provide certain functionality.

In order to react to such potentially disruptive changes, the system software must be able to detect them. This requires the system software to continuously monitor the executed applications. It is noteworthy that this monitoring is not only required to manage running applications but it is also required to maximize the possible resource utilization in a smart peer group by detecting stale functionality. Consider, for instance, a computer that provides a display exclusively for an application. If the computer leaves its group, the system software must monitor the application in order to detect that it can no longer make use of the display. Using this information the system software can then assign the display to some other application.

2.1.5 High Adaptation Transparency

In order to execute a pervasive application in a smart peer group despite the fluctuations, the system software must not only be able to determine possible configurations, but it must also provide means to adapt a configuration at runtime. From the perspective of the user and the application developer, the reconfiguration should – ideally – be transparent. Meaning that neither one realizes that an adaptation takes place. In practice there are many situations in which it is not possible to adapt an application transparently. As an example consider for instance a person using a wall-mounted display to view some documents. If the power supply of the display fails, the application needs to adapt, e.g. by redirecting the output to some battery-powered computer. Clearly, this adaptation will be noticed by the person interacting with the display and there is no way of hiding it. Thus, in practice system software can only aim at hiding the details of the adaptation process as far as possible.

With the addition that sometimes it is not desirable to hide all details of the adaptation process, this argumentation holds also true for application development. As an example consider an application that uses distributed application-specific state in order to reduce its requirements with respect to network bandwidth. In order to deal with unpredictable failures, the system software must transparently replicate the state held on each computer on some other one. If a failure occurs the system software can then use the replicated state to continue the execution of the application. Yet, replicating the application-specific state introduces communication and thus, conflicts with the initial goal of reducing the network bandwidth required by the application. As a result, system software for smart peer group should target for high adaptation transparency but it should empower the application developer to manually control the degree of transparency, if necessary.

2.2 Design Rationale

This section describes the design rationale of a distributed component system that has been geared towards fulfilling the requirements derived and described in the previous section. To motivate the abstractions that are detailed in the next section, this section first presents the targeted model of pervasive applications. Thereafter, the section discusses the resulting implications on application adaptation and it proposes a way of automating adaptation using software components.

2.2.1 Pervasive Applications

The technical basis of a smart peer group is a set of spontaneously networked computers. Since the computers are frequently integrated into everyday objects, they are usually highly specialized. Thus, most meaningful applications will need to combine the specialized functionality of a number of computers in order to create a desired application behavior. While the exact combination of functionality is clearly application-dependent, the functionality provided by a single computer should not necessarily be tailored towards a specific application. As a result, applications executed in smart peer groups consist of application-independent functionality and application-specific logic that acts as overarching "glue".

In order to support such a high-level view on applications, we adopt a generic service-oriented model for pervasive applications. This model is frequently used by mainstream system software that is targeted at the development of traditional distributed applications. With Java RMI (Sun Microsystems, 2004) and CORBA (Object Management Group, 2004) as premium examples, the overall model can be considered to be time-proven. Apart from traditional distributed applications, this model is also proposed by a number of system software solutions for pervasive systems including BASE, the communication middleware for smart peer groups. The main features of this application model are detailed in the following.

- *Atomic core*: Each application is executed on behalf of a specific person. The overarching application-specific logic of an application is implemented in an application core. This core is atomic with respect to distribution and thus, it is executed on a single computer. If a person is using an application on-the-go, the computer that executes the application core is a mobile computer carried by the person. If the person uses an application solely in a certain room, the executing computer might as well be one that continuously resides there. In order to provide the desired application behavior, the core will typically require functionality that can only be provided by other computers of the smart peer group.

- *Distributed services*: Computers of a smart peer group offer their application-independent functionality in terms of remotely accessible services, i.e. interfaces with predefined methods, parameters and return values that hide the actual implementation. In order to use the functionality of some remote computer, the application core can call methods of the desired service. To do this, the application core must first find a suitable service in its smart peer group. To select the most suitable service, each service augments its functional description, i.e. its interface description, with properties that describe non-functional properties, e.g. the quality of the provided functionality.
- *Composed services*: Due to the fact that most computers provide only a small number of specialized services, application cores will frequently need to combine a number of services on multiple computers in a meaningful manner. In order to reduce the size of the application core and to enable the coarse-grained reuse of similar functionality across different applications, some computers of a group can be equipped with services that provide high-level functionality by combining a number of low-level services. Naturally, this type of service composition can be continued recursively. Moreover, since it might be possible to synthesize the same high-level functionality by combining different low-level services, a smart peer group might contain similar high-level services with different requirements.

In addition to these common features, we make additional specializations that adhere to the fact that many computers encountered in pervasive systems will be tightly integrated into everyday objects. From this foreseeable tight integration we derive the following two features.

- *Pre-deployed services*: Although it might be technically feasible for some computers to assume that services can be dynamically installed or migrated, we assume that the set of services provided by one computer is fixed and cannot be changed easily. The reason for this deliberate limitation originates from the observation that most computers in future pervasive systems will be tightly integrated into small and inexpensive everyday objects. Despite the effects of Moore’s Law, integrating computer technology into an object will always increase its monetary cost. For many objects, the cost resulting from an integrated computer will exceed the cost of the underlying object. Thus, in order to minimize the cost, manufacturers will refrain from adding costly features like over-the-air programming to their embedded micro-controllers.

- *Limited resources*: Utilizing a service within an application requires resources on the hosting computer. Independent from the type of service, each service requires a certain amount of processing power and memory. Yet, a majority of services found in a smart peer group is likely to offer functionality that is specific to the object in which the computer is embedded. Thus, many services require additional resources. Examples might be cameras attached to a smart phone or LEDs integrated in a smart pen. Naturally, such resources are limited by the hardware design and depending on the type of integration their availability might also fluctuate over time. As an example consider an SD card that can be dynamically plugged into personal digital assistant. If some limited resources cannot – or should not – be multiplexed between simultaneously running applications, the available resources of a computer impose constraints on the number of applications that can be served by its services at the same time.

2.2.2 Implications on Adaptation

After having defined the overall picture of what constitutes a pervasive application, it becomes clear that services play the central role. By defining services as atomic with respect to distribution, they act as starting point for application configuration and adaptation. In order to function properly, the application core requires a specific set of services and resources. The services may, in turn, require other services and resources. Thus, the configuration of an application is formed by the transitive closure of the services and resources used by it.

Since the set of services is usually distributed across a number of computers, the configuration of an application is typically distributed. Due to the dynamics of the underlying smart peer group model, the set of computers that constitutes the smart peer group is continuously fluctuating. This leads to the conclusion that maintaining a static configuration for a long period of time is not possible in general. Thus, in order to execute an application despite the changing set of computers, the set of services that constitutes the application needs to be adapted.

The same holds true for the configuration of an individual service. Since the set and the amount of resources available on some computer of a smart peer group might change in a non-predictable manner, e.g. because a person unplugs some external device without prior notice, the services that are using resources need to adapt to the changing availability at runtime. It is noteworthy that the availability of resources might also fluctuate due to the changing utilization of services on a computer. However, such fluctuations can be compensated through appropriate resource reservation mechanisms. In order to do that, the services can reserve the required amount of resources upfront whenever an application starts using them.

As indicated by this discussion, there two types of changes in a smart peer group that raise the need for adaptation. On the one hand, there are changes in connectivity that can lead to an immediate and unpredictable unavailability of a computer used by an application. On the other hand, there are changes to the resource availability that can be similarly unpredictable. As introduced in the previous chapter, there are two types of modifications that can be applied to a distributed application, namely parameterization and reconfiguration.

If we interpret parameterization in the context of this specific application model, we can apply parameterization to deal with fluctuating resources. To do this, we view the resources required by a service as its set of parameters. Analogous, we view the current reservation of a resource for a service as the value assignment of the corresponding parameter. If a previously reserved amount of resources can no longer be guaranteed, the values of the parameters change and the service can adapt its behavior accordingly.

Naturally, there are limitations on the resource fluctuations that can be compensated by such a technique. For some services, the availability of a certain set of resources might be absolutely necessary to provide the functionality. For other services, reducing the amount of a certain resource might lead to changes in its provided quality of service. As an example consider a service that requires a certain amount of processing power to answer requests within a certain time frame. If the amount of available processing power is reduced the time required to answer request increases.

In order to broaden the scope of parameterization beyond a single service, we can extend it to the non-functional descriptions of services. Thus, if the lack of available resources cannot be compensated by a single service, it can propagate this by dynamically modifying its non-functional properties. From the perspective of the using service or the application core respectively, the individual non-functional properties can then be seen as parameters. If some properties of a used service change, the using service can try to compensate the change by changing its behavior.

Even if parameterization is extended across different services, it cannot be used to compensate all changes. If a service can no longer provide its functionality at all or if it becomes unavailable because a computer left the smart peer group, the application must be reconfigured structurally. This means that a subset of the services and resources contained in an application configuration is replaced with another set. In the simplest form of structural reconfiguration, a service that is no longer available is just replaced with a similar service. Yet, there are scenarios where such a simple replacement is not possible because the smart peer group does not contain a service with the required interface or non-functional properties. In such cases it might be necessary to replace services that are still available as well in order to create another configuration that does not require such a service.

Based on this discussion of parameterization and reconfiguration, one can argue that adaptation by parameterization is not an essential type of modification as parameterization can be interpreted as special case of structural adaptation. To do this, the possible parameterizations supported by an individual service can be modeled as a virtual service that does not support different value assignments, i.e. parameterizations. As a result, each parameterization of the original service can now be handled by a virtual service and thus, each fluctuation can be compensated by structural adaptation.

While this argumentation is true in general, it is still relevant to make a distinction between parameterization and reconfiguration in practice. The key reason for this is the fact that reconfiguration introduces additional overhead compared to parameterization. Usually, the reconfiguration of an application leads to changes regarding the set of utilized services and resources. Thus, in order to allow an adapted application to continue its execution seamlessly, the internal state of the new set of services must reflect the program logic that has been executed already with the original set of services. Therefore, it might be necessary to repeat the execution of some parts of the already executed program after the reconfiguration took place.

Due to the trade-off between efficiency and general applicability encompassed with parameterization and reconfiguration, it is desirable to support both types of modifications. Thereby, the main goal should be to apply parameterization whenever possible and to rely on structural reconfiguration in cases where parameterization cannot be applied. This approach combines the efficiency of adaptation by parameterization with the broader applicability of adaptation by reconfiguration.

2.2.3 Automation with Components

Mainstream system software for service-oriented applications frequently relies on service descriptions and associated centralized registry services, e.g. naming and trading services, to model and to share services. Whenever a service is installed or started, the service registers its own service description at a certain registry service. Using this registry service, another service can perform arbitrary queries that specify the desired properties of required services, e.g. their names or types and the desired quality of service. In response to a query, the registry service returns a list of candidate services that exhibit the desired properties. Using this list, the requesting service can then select a particular service to interact with.

While this approach has proven to be successful for the development of traditional distributed applications, it is not well-suited to support automatic application adaptation in smart peer groups for the following two reasons. First, this approach relies on the permanent presence of a centralized registry service which cannot be guaranteed in smart peer groups. This problem can be avoided by utilizing a federated registry service as done in BASE, for example. Secondly and more importantly, this approach enables services to perform arbitrary queries at runtime which makes it hard – if not impossible – for the underlying system software to reason about possible dependencies between services. To avoid this problem, services must explicitly specify their dependencies towards their execution environment.

A service that explicitly specifies its dependencies qualifies as a software component under the definition given in (Szyperski, 1997) where "a software component is a binary unit of composition that exhibits solely explicit contextual dependencies". According to the previous discussion on configuration and adaptation, a service can exhibit dependencies to services as well as resources. Thus, each service must explicitly model these two types of dependencies. Thereby, the resource dependencies are used to model required functionality that must be provided by the computer executing the service. Dependencies to services, on the contrary, are used to model required functionality that can be provided by some arbitrary computer of the smart peer group.

The necessity of modeling dependencies explicitly immediately raises the question whether or not a service should be enabled to change its dependencies dynamically. In the traditional service-oriented systems, such changes are supported by the fact that arbitrary queries for other services can be performed at any point in time. In most cases, however, the set of services that will be contacted by one service is limited by its provided functionality. As an example consider a service that displays a document. While such a service might require external services to convert a document, it will – by design – never contact services that control the room temperature.

This leads to the conclusion that for a large number of services it is possible to derive the complete set of possible dependencies by deriving the superset of all potentially required services and resources. Thus, one might argue that it is possible to utilize a completely static set of dependencies. Yet, it should be clear that using all potentially required resources and components as static dependencies might be inefficient in cases where only a small subset is really used. This is especially problematic for functionality that is rarely required. As an example consider the dependency of a word processor service on some printer. The word processor only requires the availability of a printer in cases where a person that uses the service actually wants to print a document.

On the other hand, if required services and resources are only treated as dependencies at the point in time when they are actually required, the set of dependencies of a certain service might fluctuate frequently. Such frequent changes introduce two major problems that result immediately from the fact that each change will require adaptation. At best – that is if all dependencies can be resolved – this introduces runtime overhead. This alone already limits the frequency of changes that can be supported in practice. However, in cases where an additional dependency cannot be resolved, the resulting unresolved dependency must be handled programmatically by the application developer or the execution of the service will fail.

In order to deal with these issues, we propose a solution that lies in the middle of both extremes. Thereby, we assume that the dependencies of a service can be captured statically for a concrete usage of a service within a single run of an application. That way, the superset of all possibly required services and resources can be narrowed down significantly without incurring the drawbacks of highly dynamic dependencies. In order to support this approach, each service needs to specify its dependencies on the basis of the requirements that it should fulfill. On the basis of this specification, the system software can then ensure that these requirements are met during a single run of the application and the service developer can always rely on the presence of all required services and resources.

For the exemplary word processor service this would mean that if it is used as part of a simple spell checker application, the processor service would not specify a dependency on a printer service. If the word processor would be used in a document viewer application that supports printing, the processor would exhibit a dependency on a printer. Clearly, for some scenarios this solution might not be flexible enough as the question of whether or not a printer is required might depend on the person that uses the application. In order to support such scenarios, we describe ways of integrating user preferences into the configuration in the next section. Yet, these preferences follow the same overarching pattern in the sense that they should be mostly static for a single run of an application.

2.3 Component System

Based on the application model presented in Section 2.2.1, we have developed a component system that supports automatic application adaptation at system level (Becker, Handte, Schiele, & Rothermel, 2004). To underline the fact that this system is specifically targeted at pervasive applications, we named it PCOM as abbreviation for *p*ervasive *com*ponents. This component system introduces a strict separation between the development of individual components and the runtime configuration and adaptation of the set of components that constitutes an application. By separating these two tasks, the latter can be automated completely using the concepts for configuration and adaptation that have been derived in Section 2.2.2. To do that each component specifies its provided functionality as well as its required functionality explicitly as indicated in Section 2.2.3.

In the following, we first provide a high-level overview of all abstractions and their interdependencies before we discuss their purpose and design in detail. Thereafter, we describe the mechanisms of PCOM that are required to enable adaptation. Finally, we close the chapter by relating the individual abstractions and mechanisms to the requirements derived previously. The algorithms that are necessary to automatically configure and adapt an application are detailed in Chapter 3 and Chapter 4. Chapter 5 describes the overall architecture of PCOM as well as some implementation details and Chapter 6 contains a more detailed evaluation of the abstractions. To keep the following description of the component system as lean as possible we to refer to these chapters for details. Where necessary, we highlight the interdependencies between the concepts and models introduced in this section and the functions of the algorithms.

2.3.1 Overview

PCOM uses components as uniform abstraction to model basic and composed services as well as application cores. These components themselves are atomic with respect to distribution but when a component is used, it may interact with other components across the boundaries of a single computer. An application may thus be distributed by combining a set of non-distributed components running on different computers.

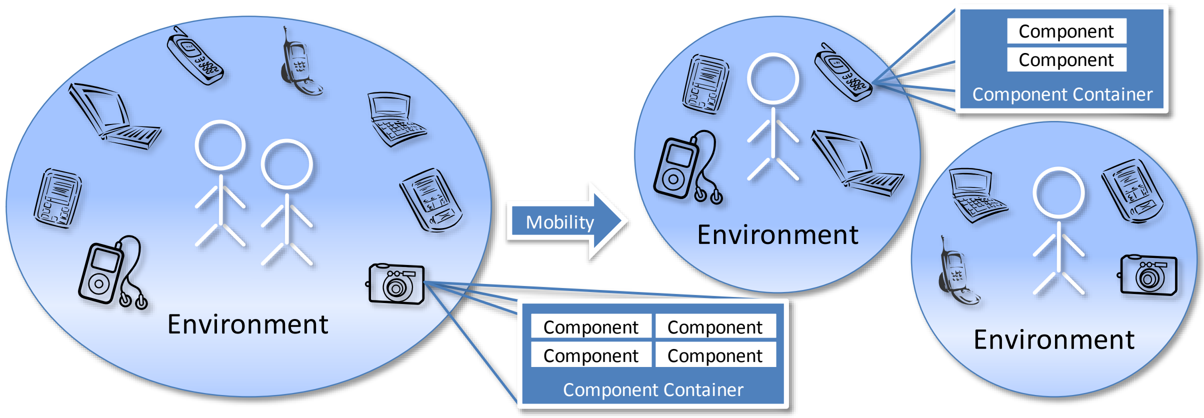


Figure 1 – Dynamic Execution Environment

The execution of a component is controlled by a so-called component container. To support multiple services on one computer, each component container is capable of hosting multiple components. As a result, each computer of the pervasive system requires only a single component container. The set of component containers that can communicate with each other forms a homogeneous execution environment for their hosted components and applications.

Induced by mobility and failures, the execution environment may be highly dynamic as indicated by the example shown in Figure 1. On the left side, the execution environment is formed by eight computers carried by two persons. If the persons are moving away from each other, the physical distance between the persons eventually exceeds the communication range supported by the short-range wireless communication adapters of their computers. This in turn causes the network to split in two independent partitions consisting of the individual computers carried by each person. Thus, the network partitioning induced by mobility creates two separate environments as shown on the right side of Figure 1.

The component containers cooperate to automate the task of configuring an application upon startup and adapting an application at runtime. To enable this, each component specifies its dependencies towards the execution environment explicitly using so-called component contracts. Thereby, each contract denotes the functionality provided by a specific component as well as the required functionality in terms of components and resources. Thus, each component container can determine whether the available resources suffice to execute a component by inspecting the requirements stated in its contract. By providing adequate matching operators for contracts, the component containers can furthermore automatically determine whether the requirements denoted in a contract can be satisfied by the functionality described in contracts of other components.

To support a high degree of compositional flexibility, the contractual specification refrains from describing the internal implementation details of a component. Instead, the functionality provided and required by a certain component is modeled indirectly through syntactical descriptions that can be extended with properties to describe the non-functional characteristics of the implementation. Thus, component implementations with the same syntactical interface are considered to be equivalent, given that they exhibit the same non-functional characteristics.

An application in PCOM is formed by recursively composing components along their contractual requirements starting from the corresponding application core. Thereby, the composition assumes that different requirements can be satisfied independently from each other, i.e. there are no further dependencies between components other than the ones that are specified contractually. Since each contract may specify an arbitrary number of required components, this type of independent composition will always form a tree rooted at the application core.

In order to simplify the remaining detailed description of PCOM, we adopt some very basic terminology from graph theory to describe the tree. The term child component or simply child is used to denote a component that is used by some other component. Similarly, the term parent component or parent is used to denote a component that is using another component. Finally, the term root is used to describe the application core in cases where the tree structure of PCOM applications should be emphasized.

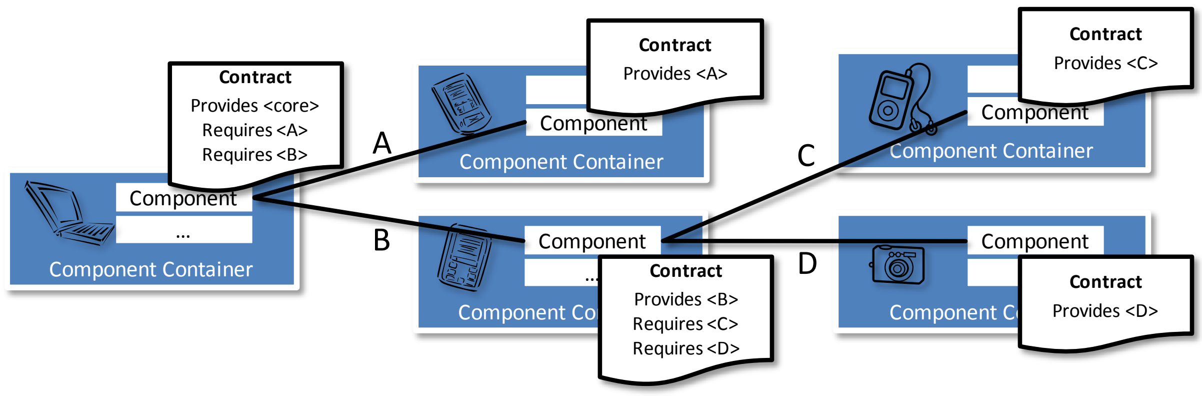


Figure 2 – Exemplary Application Configuration

An exemplary application configuration is depicted in Figure 2. In this simplified example, the contract of the application core specifies two required components that can provide the functionality A and B. Such components can be found on two different computers in the environment. The component providing the functionality A does not require further components. However, the component that provides B requires further components that are capable of providing the functionality C and D. In this example, C and D are the children of B and the application core is the parent of A and B.

By convention, an application can be executed if and only if all contractually specified requirements can be met. This means that for each recursively occurring component requirement there must be one component with a contract that provides a matching functionality. Furthermore, this also means that there must be sufficient available resources on all component containers that are hosting components of the application such that the components can utilize their required resources simultaneously. As a result, the components that are forming the leafs of an executable application will never specify further component requirements and the containers that are hosting components of an executable application will always exhibit an equal or greater amount of resources than the sum of the resources required by the components.

When the component containers cooperatively configure an application, they ensure that only executable configurations are computed and started. However, since the set of component containers and resources that is available in an environment can change during the execution of an application, an executable application might become non-executable at runtime. If this happens, the component containers need to adapt the configuration in such a way that the resulting configuration becomes executable again. If this is not possible, they need to stop the execution of the application.

To adapt an application, PCOM provides mechanisms that can replace sub-trees of an application with newly computed sub-tress that result in an overall executable configuration. While this type of adaptation is a comparatively simple process for sub-trees that are stateless, additional precautions must be taken if some components exhibit application-specific state. In order to continue the execution of an application at the point where it left off when its configuration became non-executable, the state of all modified sub-trees need to be restored appropriately.

Each component container provides a set of semi-automatic check-pointing, logging and replay mechanisms that can greatly simplify the task of developing components that can be adapted automatically. The underlying principle is to continuously capture information during the execution of a component that can be used to restore the state of its required components at any point in time. However, to restore the state of a reconfigured sub-tree in a consistent manner, the mechanisms need to reset all previous state that is possibly carried by reused components of an adapted sub-tree. Thus, in order to avoid such explicit resets, PCOM simplifies adaptation by replacing complete sub-trees. This approach does not only simplify the restoration of state, but it also supports a greater degree of compositional flexibility. For instance, the component B in Figure 2 can be replaced by any component that is capable of resolving the corresponding dependency of the core, since it does not have to reuse the components D and E.

The previously described model already provides a basic overview of the fundamental concepts of PCOM components and applications. However, there are a number of refinements of the component model that need to be mentioned to complete the overview.

First of all, PCOM components may support multiple contracts with different non-functional provisions or requirements. The key reason for this results from the observation that a single component implementation might support multiple modes of operation. Depending on the implementation, one component might for instance be able to provide a high-performance as well as a low-performance mode of operation that require a high and a low amount of resources respectively. Thus, supporting multiple contracts can be used to model different component parameterizations. As indicated by the discussion in Section 2.2.3, supporting component parameterization can greatly improve the adaptation performance by avoiding more costly reconfigurations.

Furthermore, in order to enable the usage of one component in multiple applications or in different sub-trees of a single application, components are instantiated for each individual usage. Thus, applications are not formed directly by components but they are formed by a set of component instances. Thereby, each component instance used within the application is equipped with exactly one contract at any point in time. Thus, different parameterizations of one component might be used for different component instances at the same time. In order to manage components during the execution and adaptation of an application, PCOM defines a basic lifecycle for component instances that is controlled by the hosting component container by means of callbacks. As explained in greater detail in the following sections, this enables component instances to react to changes in an application configuration and it allows the container to ensure that all unused component instances are released properly.

A similar approach is taken with respect to resource utilization where the usage is abstracted in a so-called resource assignment that is issued by a resource manager. Just like component instances represent a single usage of a component in some sub-tree of an application, resource assignments represent a single resource usage by one component instance. Similar to component instances, resource assignments might be parameterized differently and thus, they also follow a predefined lifecycle. However, in contrast to the component lifecycle which is completely controlled by the component container, the lifecycle of resource assignments only partly controlled by the component container since resources may become unavailable at runtime.

Finally, as we discuss in depth in Chapter 3 and Chapter 4, automatic configuration and adaptation of PCOM applications are tasks that might have to compute partial configurations frequently. During the configuration and adaptation, the component system needs to reason about possible configurations by composing components along their contractually specified dependencies. Since the instantiation of a component might be a comparatively expensive task in terms of processing and memory requirements, PCOM components and PCOM resources are represented by so-called component factories and resource managers respectively. These representatives are used to efficiently create and issue appropriate contracts without instantiating a component or reserving a resource and they are also used to instantiate components and to make resource reservations when they are really needed.

2.3.2 Contracts

To enable automated reasoning on possible configurations and necessary adaptations, PCOM relies on an explicit contractual description of dependencies between all parts that constitute an application. As a natural consequence of the component and the resource abstraction introduced by PCOM, the component system employs component contracts to model the provision and the demand of a component and resource contracts to model the provision of a resource. Since a component may depend on other components as well as resources, the component demand may specify required components and required resources. In contrast to that, resources are assumed to have no further dependencies, thus, their contracts do not contain demands.

Although there are a number of technical differences between components and resources, PCOM uses very similar concepts to describe dependencies on components and on resources. In order to avoid repetitive descriptions, we will first discuss the scope of the contract model including the overall design decisions and the general matching rules, before we present the details of component contracts and resource contracts. In the general discussion, we refer to dependencies on software components as placeholder for dependencies on components and on resources. From a theoretical point of view, this does not conflict with the definition of the term software component given earlier since resources are also units of composition with explicit dependencies as well.

2.3.2.1 Contract Scope

There are different possible ways of describing the provision and demand of a software component using contracts. According to (Beugnard, Jezequel, Plouzeau, & Watkins, 1999) contracts of software components can be classified according to their contents into the following four levels of increasing expressiveness:

- *Syntactic level*: On the lowest level, a contract can model the syntactic interface of the software component. The syntactic level usually describes the supported operation of a component in terms of names, return type(s) and parameter types as well as possible exceptions. Syntactic descriptions form the basis of most mainstream imperative and object-oriented programming languages such as C++ or Java and they are also frequently used to model services or remote objects in many traditional communication middleware systems such as CORBA or Java RMI.
- *Behavioral level*: In addition to solely describing the syntax of the interface of a software component, a contract can also describe its internal behavior. To do this, the contract can describe pre- and post-conditions as well as invariants that hold during execution. Behavioral descriptions allow a more thorough reasoning about the correctness of a program that is composed from different parts without looking at the implementation of the individual parts. This form of description is used in some "newer" programming languages such as Eiffel and it is also used by the Object Constraint Language which is a part of the Unified Modeling Language.
- *Synchronization level*: The behavioral level already captures some of the internals of a software component without exposing the implementation. However, behavioral descriptions usually assume sequential execution and they fall short of capturing the internals in cases where parallelism is present. In order to model the effects of parallelism, contracts can also contain synchronization descriptions. At the time of writing, such synchronization contracts are rarely found in existing products. In fact, some component-based technologies such as Enterprise Java Beans explicitly try to hide

the complexities of parallel programming by ensuring that components are always executed within the context of a single thread.

- *Quality of service level*: The previous levels of descriptions are mostly concerned about what the software component performs. In addition to simply describing what a software component does, a contract can also model how well it performs this task under certain conditions. In this case the contract would describe the quality of service delivered by the concrete implementation. Possible quality parameters could be the execution time of a certain operation or the precision of some result value.

Many techniques that facilitate the reuse of functionality such as program libraries solely describe the syntactical interface explicitly. However, it should be noted that this does not mean that there are no descriptions of the functionality on the higher levels. Usually such descriptions are available as some kind of human readable documentation. Yet, descriptions based on natural language are difficult to process automatically and they are hard to check for completeness or correctness. As a result, weak low-level descriptions can easily endanger the correctness of programs, especially, if they are used to perform automatic configuration. On the other hand, correct and complete descriptions in natural language may be easier to understand for a developer than potentially complex formal descriptions and thus, they may simplify the manual reuse of functionality.

To contractually describe dependencies between software components, we decided to adopt a solution that lies in the middle ground between low- and high-level descriptions that has proven to be successful in the past. Essentially, we utilize descriptions based on interface names to capture the syntactic interfaces and we utilize typed name-values pairs to model relevant aspects of the implementation. Thus, the resulting types of contracts capture the syntactic and the quality of service level. However, it is important to mention that the overall approach on configuration and adaptation as well as the programming model of PCOM applications is not tightly coupled to the features that are modeled in contracts. Thus, it is possible to change the type of contract at will without affecting most other parts of the component system.

For the syntactic description of interfaces, we assume that there exists some common agreement on the names, their associated behavior, their synchronization as well as their semantic. This assumption is frequently made by many programming languages such as Java or .NET, for example. Accidental collisions of interface names can be effectively prevented by adopting appropriate naming conventions. As an example consider the convention for the Java programming language as proposed by Sun Microsystems (Sun Microsystems, 1999) where Internet domain names are used as primary prefix to ensure uniqueness of package names. In order to avoid the potentially tedious definition of a single interface for each type of software component, we allow them to expose multiple interfaces. Thus, types are modeled by listing the name of the interfaces that are supported or required.

For the typed name-value pairs, we support basic primitive types such as Booleans, Integers and Strings to capture the quality of service provided by a component implementation. In analogy to the assumptions made for interfaces, we assume that their number, their types and their semantics are known in advance and tied to the corresponding interface. Thus, the declaration of the properties related to quality of service becomes a task that needs to be done by the designer of an interface whereas the task of assigning concrete values for a certain implementation needs to be performed by the developer of a component. To simplify the task of assigning meaningful names to type-value pairs, we group them into so-called dimensions.

2.3.2.2 Contract Matching

Apart from defining the scope of contractual descriptions, it is also important to define how the captured provision and demand can be compared with each other. Thereby, it is important to mention that the complete demand of a contract can encompass multiple individual demands either on other components or on resources. Since the PCOM application model assumes independence between different requirements, the notion of comparison only needs to be defined on the basis of provisions and individual demands.

For the syntactical descriptions on the basis of interfaces the comparison is relatively straight-forward. Due to the uniqueness of interface names, we can simply perform a comparison of the interface names to match individual interfaces. However, since contracts may contain multiple interface names, we need to define additional rules on the basis of sets of interfaces. Naturally it should be clear that if a contractual provision does not contain all interface names that are contained in the demand, the software component that will run under this provision cannot satisfy the demand as it misses some required functionality. Thus, in order to satisfy some demand a contractual provision must at least provide the interfaces that are also required. The question whether a contract that offers more interfaces than required should also match a certain requirement can simply be defined as true according to the usual sub-typing rules of programming languages.

While we can simply use the equality of interface names to compare the provision and demand of a contract, comparing the typed name-value pairs is slightly more complicated. This can be attributed to the fact that their meaning is not predefined as with interfaces. In order to support a variety of different usage scenarios, PCOM contracts support different evaluation methods for the name-value pairs. These evaluation methods can be attached to individual name-value pairs. Based on the type of the pair, a contract may contain simple comparators such as *Equals* that can be used to demand a provision that represents a certain point in the multidimensional space spanned by the corresponding name-value pair. In addition, a contract may also contain range comparators such as *Greater*, *GreaterOrEquals, Less*, *LessOrEquals*, *InRange* and *OutRange* that declare a certain region of the space as matching.

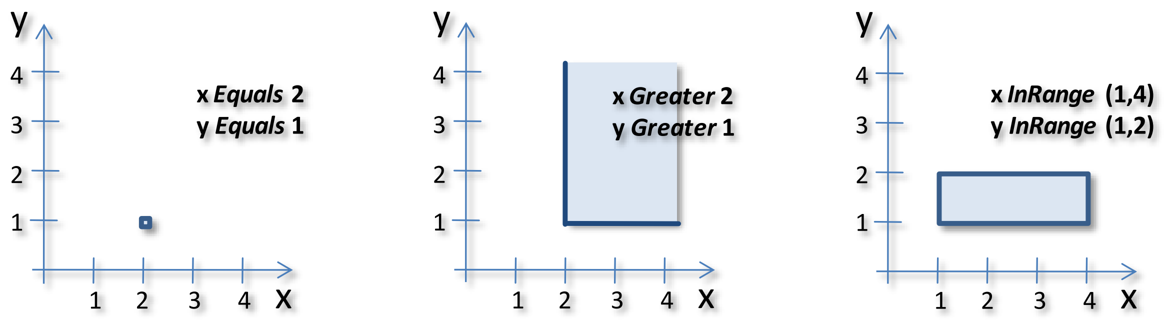


Figure 3 – Contract Comparators

Figure 3 shows three examples for points and ranges resulting from different comparators in a two-dimensional space spanned by the two integer name-value pairs x and y. The leftmost example shows a point that is spanned by two *Equal* comparators. The rightmost example shows a rectangle that is formed by two *InRange* operators and the example in the center indicates the infinite range spanned by two *Greater* operators. Naturally, these are only very simple examples. Real component contracts will usually declare a greater set of name-value pairs and they will frequently combine different operators.

Clearly, one might ask whether these comparators suffice cover all application scenarios. As discussed in Chapter 4.2.4.5, the comparators that have been discussed so far are sufficient to effectively support our set of exemplary demonstration applications but there might be cases where the limitations of these comparators require rather inconvenient workarounds. To mitigate such cases, the set of comparators could be extended but since the comparators need to be available on all component containers, such an extension cannot be done in isolation. Yet, there are no mechanisms in PCOM that are dependent on the specifics of the set of comparators. Thus, the question whether the set is complete can be neglected from a conceptual point of view.

In summary, we can define a match between a provision *p* (with the provided interfaces *ifs(p)*) and a demand *d* (with the required interfaces *ifs(d)*) more formally as follows:

$$(\forall d_i \in ifs(d) \exists p_i \in ifs(p) : match_i(d_i, p_i))$$

Formula 1 – Contract Matching

The Boolean functions *match$_i$(d_i,p_i)* between a required interface d_i (with the required type *name(d_i)* and the dimensions *dms(d_i)*) and a provided interface p_i (with the provided type *name(p_i)* and dimensions *dms(p_i)*) is defined as:

$$(name(d_i) = name(p_i)) \wedge (\forall d_d \in dms(d_i) \exists p_d \in dms(p_i) : match_d(d_d, p_d))$$

Formula 2 – Interface Matching

The Boolean function *match$_d$(d_d,p_d)* for a required dimension d_d (with the name *name(d_d)* and the set of properties *prs(d_d)*) and a provided dimension p_d (with the name *name(p_d)* and the set of properties *prs(p_d)*) is defined as:

$$((name(d_d) = name(p_d)) \wedge (\forall d_p \in prs(d_d) \exists p_p \in prs(p_d) : match_p(d_p, p_p))$$

Formula 3 – Dimension Matching

The Boolean function *match$_p$(d_p,p_p)* for a required property d_p (with the name *name(d_p)*, the type *type(d_p)*, the value *value(d_p)* and the comparator *cmp$_{dp}$*) and the provided property p_p (with the *name(p_p)*, the type *type(p_p)*, and the *value(p_p)*) is defined as:

$$(name(d_p) = name(p_p)) \wedge (type(d_p) = type(p_p)) \wedge (cmp_{dp}(value(d_p), (value(p_p)))$$

Formula 4 – Property Matching

The comparator functions *cmp$_{dp}$* are defined as their corresponding mathematical expression, e.g. *Greater(x,y)* corresponds to $x > y$ and *InRange((x,y),z)* is defined as $x <= z <= y$. The concrete comparator can be flexibly specified by the developer and the set of comparators could be extended if needed.

2.3.2.3 Component Contracts

As mentioned previously, components may exhibit dependencies on other components as well as on resources. In both cases, these dependencies are captured by specifying the interfaces using identifiers and the quality-related aspects using typed name-value pairs that are grouped into dimensions. However, in order to efficiently support interaction, PCOM components can support two different and incompatible types of interfaces. As explained in more detail in Section 2.3.3, these two types of interfaces are mainly a technical optimization. Yet, since they are not compatible, they need to be described separately to ensure that they are not confused.

To do this, component contracts contain separate keywords for interfaces based on method calls and on events. By extending the matching rules in such a way that they compare these two types of interfaces separately, we ensure that only interfaces based on the same type of interaction are compared with each other. More formally, we can extend Formula 1 for a component demand *d* (with the interfaces for method calls $ifs_m(d)$ and the interfaces for events $ifs_e(d)$) and a component provision *p* (with the interfaces for method calls $ifs_m(p)$ and for events $ifs_e(p)$) as:

$$(\forall d_m \in ifs_m(d) \exists p_m \in ifs_m(p) : match_i(d_m, p_m)) \wedge (\forall d_e \in ifs_e(d) \exists p_e \in ifs_e(p) : match_i(d_e, p_e))$$

Formula 5 – Component Contract Matching

Thereby, the Boolean functions $match_i$ between a required interface based on method calls (d_m) or events (d_e) and a provided interface based on method calls (p_m) or events (p_e) is defined as shown in Formula 2.

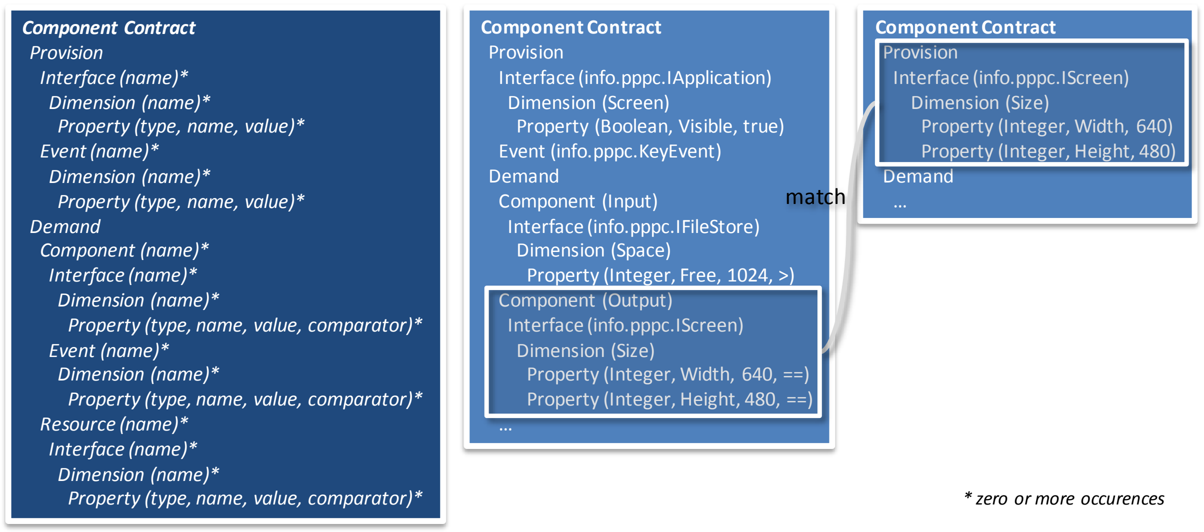

Component Contract
Provision
Interface (name)*
Dimension (name)*
Property (type, name, value)*
Event (name)*
Dimension (name)*
Property (type, name, value)*
Demand
Component (name)*
Interface (name)*
Dimension (name)*
Property (type, name, value, comparator)*
Event (name)*
Dimension (name)*
Property (type, name, value, comparator)*
Resource (name)*
Interface (name)*
Dimension (name)*
Property (type, name, value, comparator)*
Component Contract
Provision
Interface (info.pppc.IApplication)
Dimension (Screen)
Property (Boolean, Visible, true)
Event (info.pppc.KeyEvent)
Demand
Component (Input)
Interface (info.pppc.IFileStore)
Dimension (Space)
Property (Integer, Free, 1024, >)
Component (Output)
Interface (info.pppc.IScreen)
Dimension (Size)
Property (Integer, Width, 640, ==)
Property (Integer, Height, 480, ==)
...
match
Component Contract
Provision
Interface (info.pppc.IScreen)
Dimension (Size)
Property (Integer, Width, 640)
Property (Integer, Height, 480)
Demand
...
* zero or more occurences

Figure 4 – Component Contracts

Figure 4 shows the resulting generalized structure of component contracts as well as two exemplary instances. As indicated by the generalized structure shown on the left side, a contract always contains the provision and the demand section. In the provision section, a contract may contain an arbitrary number of interface definitions for interfaces based on method calls or events. These may in turn contain an arbitrary number of dimensions that group a set of properties, i.e. the typed name-value pairs that model the quality related characteristics of the corresponding component implementation.

In the demand section, a contract may declare an arbitrary number of dependencies either on components or resources. To simplify the development process, developers must attach a locally unique name to each of the specified dependencies. These names can later on be used to identify the dependencies in the component implementation. Each dependency on a component can in turn contain an arbitrary number of interface and event declarations that must be supported by a matching component. Each interface declaration may declare the required set of properties grouped into dimensions. However, in addition to just providing typed name-value pairs, a property that describes a demand must also contain a comparator to enable automated matching. The description of resource dependencies is similar to the description of dependencies on components. However, resources may only exhibit one type of interface. More details on resource dependencies are given in the following section on resource contracts.

With these definitions, it is generally possible to specify a provision that does not contain any interface declaration. Similarly, it is possible to specify component demands without any required interfaces or events. While PCOM can operate with such descriptions, specifying them will usually not make sense. This results from the fact that a provision that does not contain any interface specification does not specify any provision at all. As a result, such a component will usually not be usable by others. Similarly, a demand that does not specify at least one interface will be satisfied by every possible provision. However, in practice such contracts should not occur as component developers would not be able to use the functionality of some component whose type is not known during development.

According to the generalized structure introduced previously, the two exemplary contract instances shown on the right side of Figure 4 declare a provision and a set of demands. The leftmost contract instance declares that the corresponding component provides an interface and an event and it requires two further components. The contract shown on the right side declares that the component provides one interface. According to the matching rules of contracts, the provision specified by the contract on the right side would match a declared dependency of the contract shown on the left side. This is a result of the fact that both, the demand and the provision, are specifying the same set of interfaces with the same dimensions and the same properties and the *Equals* comparators specified together with the properties of the demand evaluate to true for the values specified by both contracts.

2.3.2.4 Resource Contracts

In addition to dependencies on components, component contracts may also declare dependencies on resources. In contrast to dependencies on components which differentiate two types of interfaces, resource dependencies may only support one type of interface. This is a result of the fact that resources are always allocated on the same container as the component that uses the resource. Thus, there is no need for supporting different types of interactions since specialized interaction patterns can be integrated easily.

In general, dependency declarations on resources follow the same overall approach used for dependencies on components. A component contract may contain an arbitrary number of resource specifications that are identified by a locally unique name. Each resource specification may then contain an arbitrary number of interface specifications which may contain typed name-value pairs with comparators that are grouped into dimensions. Similar to component provisions, resource contracts will contain a provision section that follows the same general structure with interfaces, dimensions and properties.

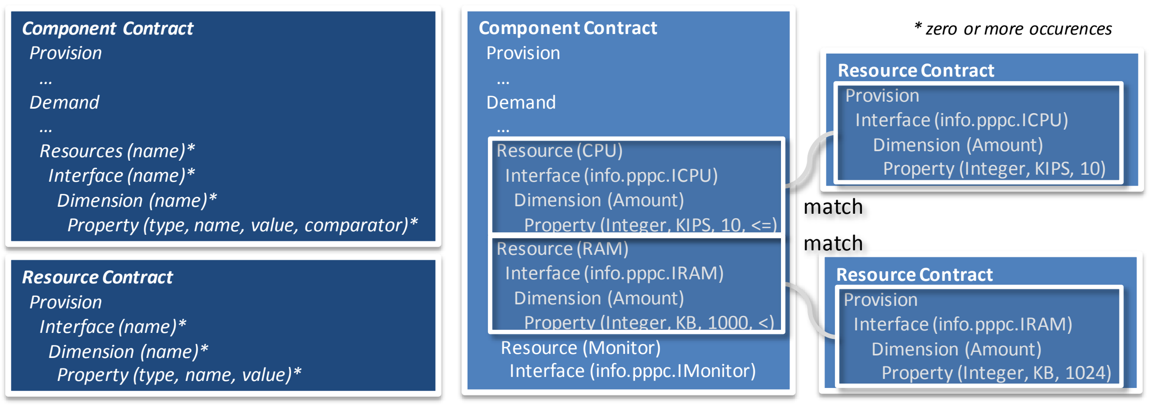


Figure 5 – Resource Contracts

The left side of Figure 5 shows the resulting generalized structure of the relevant part of the demand section of a component contract as well as the generalized structure of a resource contract. In addition, the center and the right side of Figure 5 show an exemplary fragment of a component contract that requires three resources. According to the general matching rules of contracts defined earlier, the provision of the resource contracts shown on the right side can be used to satisfy two of the three resource requirements defined by the component contract.

2.3.3 Components

In order to enable the development of services that can be adapted by the system, PCOM introduces a light-weight and unifying component model to represent application cores, services and composed services. Following most mainstream component systems such as .NET (Chappell, 2002) and COM (Microsoft Corporation, 1995), the component model does not regulate the granularity of individual components, but it leaves this decision up to the application developer. However, due to the fact that PCOM performs automatic application configuration and adaptation, there is an inherent tradeoff between smaller components, e.g. to improve the reuse, and configuration overhead, e.g. due to a higher number of component instances that need to be composed as an application. The main programming abstractions of this model are component contracts, component factories and component instances. Their high-level relations are depicted in Figure 6.

As described in Section 2.3.2, dependencies between components and resources are explicitly modeled using component contracts. The component factory is responsible for creating component contracts and component instances whenever needed. Component instances are the basic unit of composition and they are used solely to satisfy one dependency in one application at a time. The component container that hosts the component is responsible for managing the execution of the component factory as well as all component instances. It requests the creation of component contracts as well as component instances, it signals changes that affect the execution of a component instance and it is responsible for controlling the configuration and adaptation process. In the following, we describe the internals of component instances and component factories and the interaction with the component container in more detail.

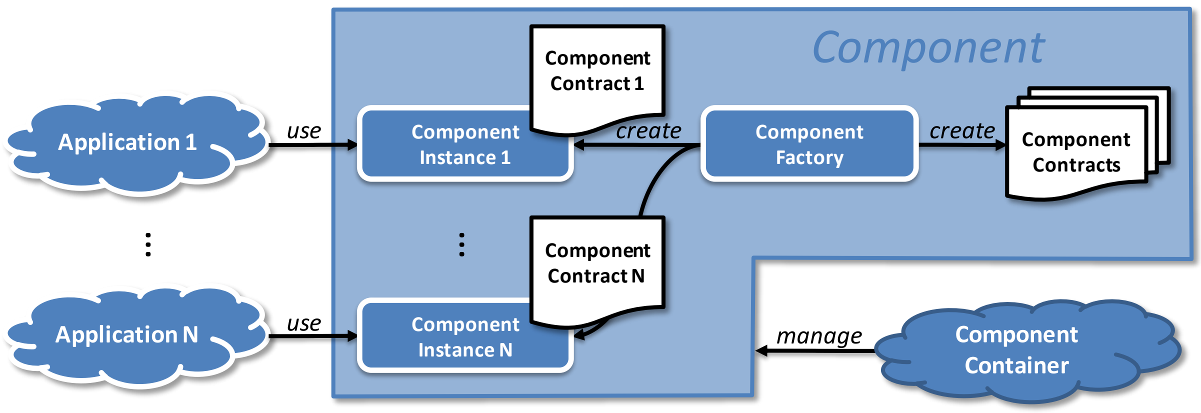


Figure 6 – Component Model

2.3.3.1 Component Instances

In PCOM, component instances represent the basic unit of composition. As a result of the tree-based application model, each component instance that is not representing an application core is used by exactly one other component instance and every component instance may require an arbitrary but fixed number of other component instances. These component instances can be hosted either on the same component container or on some remote component container. In addition, a component instance may also require an arbitrary but fixed number of resources. However, resources will always be provided by the component container that is hosting the instance.

Component instances provide the implementation of the component-specific functionality that may be utilized by other component instances during the execution of an application. This component-specific functionality is made available indirectly through interfaces. The interfaces hide the internals of a concrete implementation which causes the so-called dependency inversion where both – the service and its clients – are solely depending on the interface instead of on each other.

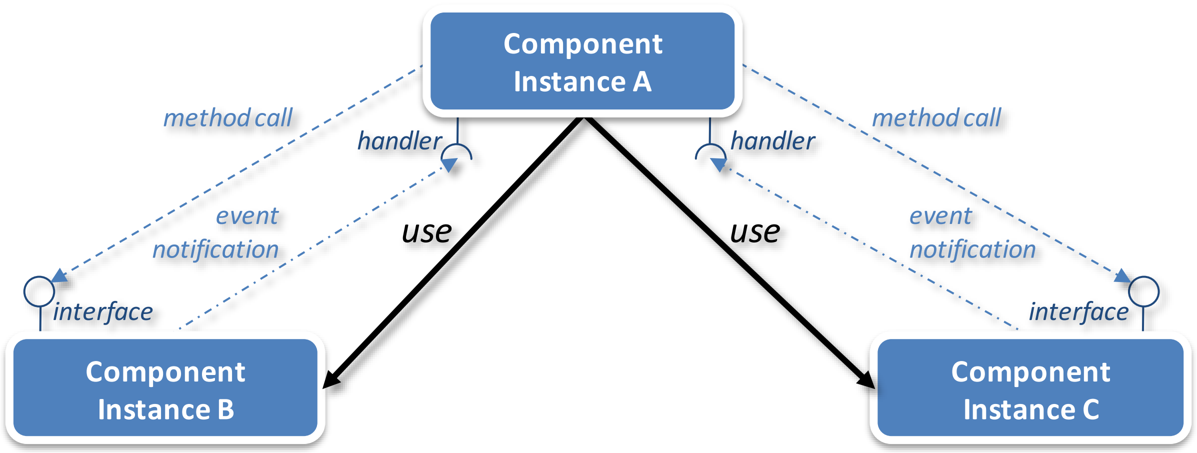


Figure 7 – Component Instance Interaction

A component instance can interact with the components that it requires as well as with the component instance that uses it. Similarly, a component may interact with the resources that are required by it. However, PCOM only supports interaction between component instances or resources that are directly depending on each other, i.e. between parent and children. This deliberate restriction is made mainly for two reasons. First of all, it ensures that implementation-specific details such as the utilization of resources or component instances are not exposed to other component instances. As a result, this enables component implementations to synthesize their functionality in vastly different ways without affecting their clients. Secondly, this restriction also ensures that component instances in different sub-trees can be adapted independently without affecting references that are hidden within a component implementation. Thus, this strong information hiding is a necessary prerequisite to support the high degree of compositional flexibility that is targeted by PCOM.

As depicted in Figure 7, the primary interaction mechanism between two PCOM component instances is a (usually synchronous) method call that originates from a parent component instance and targets a child component instance. This enables parent instances to utilize the functionality provided by its required instances as needed. In addition, a component instance can also signal changes to its parent by firing a (usually asynchronous) event. Such events can used to signal internal state changes in an efficient manner by avoiding costly polling.

Since both types of interaction involve remote communication (in many cases), PCOM uses the well-known remote proxy concept (Gamma, Helm, Johnson, & Vlissides, 1995) in order to hide all details of remote communication except for failure handling. Thereby, the component container that is hosting the corresponding component instance takes care of preparing the proxies that can be used to perform method calls and to fire events. Furthermore, the hosting component container is also responsible for redirecting the calls to the proxy to a newly bound component instance in cases where the application configuration has been adapted.

In contrast to component instances, which can be running in separate component containers on different computers, resources are always guaranteed to be available locally. Thus, PCOM components can make use of them in a direct manner. To do this, a resource may provide a handle as a (local) object which acts as a façade for the functionality that is provided by the resource. However, such a façade is not useful for all types of resources as some might either be used implicitly or because abstracting accesses using a façade would cause too much runtime or programming overhead.

The PCOM component containers try to ensure that every required component instance is resolved appropriately and that all required resources are available during the lifetime of the component instance. However, due to unforeseeable changes and the distributed nature of PCOM applications this strong guarantee can only be approximated in general. Thus, there might be (usually short) periods of time, in which the resource requirements of an instantiated component are not met or in which one or more required component instances are not available. During this period, accesses to an unavailable resource or component instance will result in a failure that needs to be handled programmatically by the component developer.

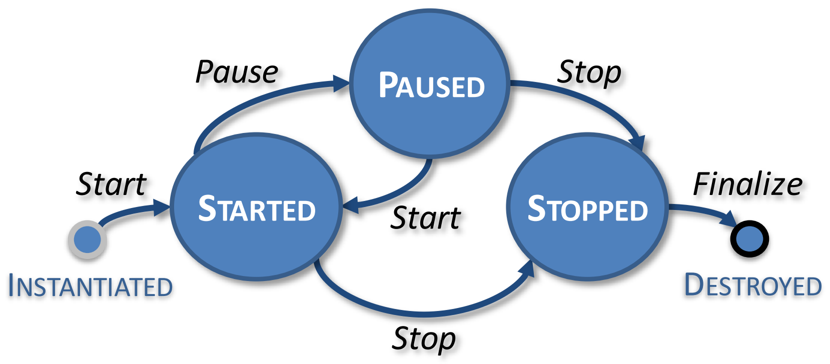


Figure 8 – Component Instance Lifecycle

At a first glance, this might seem to limit the applicability of the overall approach, since this means that developers need to deal with all failures that can occur at runtime. However, in many cases providing appropriate failure handlers is extremely simple, since the hosting PCOM container will eventually detect and resolve the problem. Thus, in most cases failures can either be signaled immediately to the component instance that initiated an interaction, for instance by raising an exception, or the component instance that detects a failure can simply wait until the component container has resolved the problem, for example by adapting the application, and repeat the action to complete it successfully.

In order to enable this type of failure handling and in order to minimize the period of time in which such failures can occur, the PCOM component model introduces a lifecycle for component instances. Conceptually, the lifecycle consists of three states, namely STARTED, STOPPED and PAUSED as shown in Figure 8. The STARTED state models the fact that a component is used and is resolved properly, meaning that a failure has not yet been detected by the component container. The PAUSED state is used to model that a component may not be resolved properly and that it should refrain from performing actions until its state changes. The STOPPED state is used to model the fact that the component instance is no longer in use and that it must immediately stop using other component instances and resources.

Transitions between these states are triggered externally by the hosting component container using Start, Pause and Stop triggers as shown in Figure 8. According to the depicted lifecycle, a component instance will be eventually started after it has been instantiated. Thereafter, it might be paused and started again several time until it is finally stopped. It might be either stopped while it is paused or started. After an instance has been stopped, it will eventually be finalized and destroyed. The reasons for triggering each state transition depend on the overall lifecycle of the application which is discussed in greater detail in Section 2.3.5.

Ideally, a component instance should not require separate triggers for these states as the pure fact that a component has been instantiated can be used to model the fact that it is used by an application. Similarly, the destruction of a component instance suffices to model the fact that it is no longer needed or is not resolved properly. However, in order to avoid frequent and potentially costly component instantiations, the PCOM component model separates these states from component instantiation.

2.3.3.2 Component Factories

As discussed in Section 2.2.3, PCOM is based on the idea of keeping dependencies between components and resources static at runtime while supporting the flexible declaration of different dependencies on the basis of the usage of a component. Since PCOM contracts are created individually for each instance, keeping dependencies static can be achieved by prohibiting contract manipulations at runtime. However, in order to support the specification of different dependencies on the basis of the usage of a component, contracts need to be adaptable to different demands.

To support contracts that are both, modifiable during configuration and static at runtime, PCOM introduces the concept of component factories. Component factories act as the local representative of a component for the hosting component container. The component factory is responsible for creating component instances and for creating contracts during configuration of an application without actually instantiating a component. This allows the component system to reason about different possible configuration without introducing the overhead of instantiating components that cannot be executed later on. To create the contents of a component contract for a component instance on the basis of its usage by an application, the component container supplies the component factory with a corresponding demand that needs to be satisfied. Based on this demand, the component factory can refine the dependencies.

Since the component container should be strictly separated from component implementations in order to support the development of arbitrary components, it cannot and should not have knowledge about the contracts supported by a certain factory. Thus, the component container may pass demands to component factories that cannot be satisfied by its instances, e.g. because the instance does not support a required interface or because the desired quality of service cannot be achieved with the implementation. In such cases the component factory will not be able to create a contract that matches with the demand. In other cases, where the component demand can be potentially met by the instances of the component, the factory will return one or more matching contracts. Creating more than one contract might be necessary to express that the instance can support various tradeoffs to support the desired quality of service.

In principle, tradeoffs between different demands can also be modeled by providing appropriate mapping functions that explicitly model how quality of service related provisions can be broken down into requirements. However, since such mappings can become arbitrarily complicated – at least from a theoretical point of view – designing a generic mapping language that covers all relevant cases – without developing a general purpose language – is a non-trivial task. Thus, instead of expressing the mapping explicitly using a domain specific language, we allow developers to specify a set of possible mappings as individual contracts. As discussed in the next chapter, this approach additionally transforms the overall problem of finding a configuration from a continuous problem into a discrete one.

The potential loss of expressiveness that is accompanied with this discretization can to some extend be mitigated by utilizing ranges instead of points to capture requirements. Thus, instead of modeling that a required parameter should have a certain value, a contract can specify that the parameter should be within a certain range. As discussed later on, this approach has the additional advantage that the ranges can be used to support a very light-weight form of adaptation by parameterization and thus, they reduce the frequency of more costly reconfigurations. However, from a theoretical perspective, introducing such ranges will waste some resources since they will not be assigned optimally.

As discussed in Chapter 6, we have not found this to be a major problem for the applications and components that we have built to evaluate the component system. However, this may be a result of the fact that our exemplary components are using the quality of service related parameters mostly to describe the context of components, e.g. the location or the capabilities of the hosting component container, instead of their performance. In order to provide additional support for component developers during contract creation, negotiation mechanisms and protocols could be offered to component factories as system services. Yet, the utilization of such mechanisms is orthogonal to the overall approach on automatic configuration and adaptation.

2.3.4 Resources

Besides from facilitating extensibility at the application level by supporting the development and use of arbitrary components, the PCOM component system also supports extensibility on the system level by enabling the development and use of arbitrary resources. Thereby, resources follow the same pattern as components. Like a component, a resource consists of three parts, namely resource assignments that are issued to components, resource contracts that describe the provision of a resource and resource managers that act as local representative for a certain resource. The relation of these parts is shown in Figure 9.

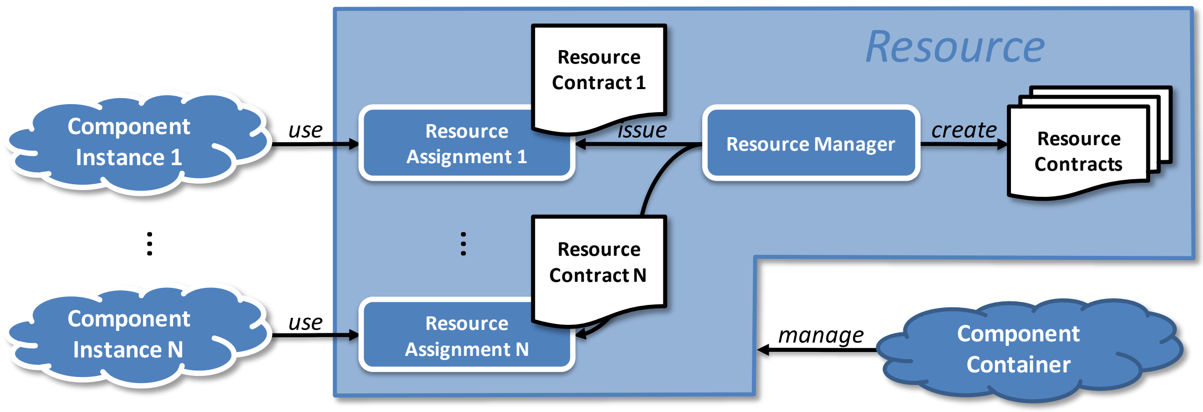


Figure 9 – Resource Model

Resource contracts are used to describe the provision of a resource with respect to a certain demand in a way that enables automated reasoning about possible configurations and adaptations. If a resource is used by a component instance, the resource manager is responsible for issuing an appropriate resource assignment to the instance. Thereby, the resource assignments are representing the fact that a resource or parts of it are exclusively assigned to the instance. Following the general approach of component factories, resource managers are also responsible for creating contracts during the configuration without actually making reservations for assignments. In addition to the responsibilities of component factories, resource managers are also responsible for ensuring that the resource is actually available as long as a resource assignment is issued. Thus, it needs to revoke resource assignments in cases where the corresponding resources become unavailable.

2.3.4.1 Resource Assignments

Resource assignments are similar to component instances with respect to the fact that they are the runtime representation of a certain resource that is used by some component instance. In contrast to component instances whose type depends on the functionality provided by the component, PCOM resource assignments are generic entities that are created by the container and issued by the resource manager. This shift results from the observation that there are a number of resources that are used implicitly or whose abstraction as a single entity would be too expensive – either in terms of runtime or programming overhead. Examples for resources that are usually used implicitly are processor cycles or main memory in high-level programming languages that support automated memory management, e.g. by means of garbage collection. An example for a resource whose abstraction might be too expensive in terms of programming overhead could be a program library with a broad interface. Clearly, there are also resources that are used explicitly. To cover such resources in an adequate manner, resource assignments may contain a handle that provides a façade to the actual resource. An example for such a handle might be a window handle for a resource that represents a graphical user interface.

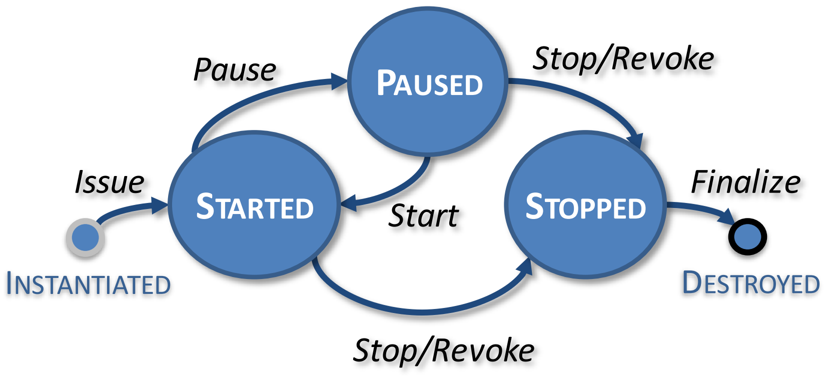


Figure 10 – Resource Assignment Lifecycle

In contrast to components that may depend on other components or resources, resources are not allowed to have further dependencies. However, this does not mean that resource assignments can be issued unconditionally. Unlike components that are solely used to encapsulate a piece of program logic and thus, represent a virtual part of the application, resources in PCOM can represent physical resources such as cycles of a processor, main memory or external I/O devices. So the question whether a resource assignment can be issued typically boils down to the question whether there are sufficient physical resources available in order to provide the requesting component instance with the required amount. Since this question cannot be answered without additional knowledge about the resource, it is the responsibility of the resource manager to issue assignments only if sufficient resources are available. Furthermore, as physical resources might become unavailable at any point in time, a resource assignment that once represented an available resource might need to be revoked later on.

In order to enable the resource managers to correctly compute the amount of resources that are available and that can be assigned to component instances, resource assignments are equipped with a lifecycle that is similar to the lifecycle of component instances. Each resource assignment can be STARTED, STOPPED or PAUSED. Resource assignments in the STARTED state represent resources that are reserved for component instances. Resource assignments in the STOPPED state represent resources that are no longer needed or that can no longer be provided. Resource assignments in the PAUSED state represent resources that are bound to instances that are currently also in the PAUSED state.

Probably one of the most important differences between the lifecycle of resource assignments and the lifecycle of components is that the lifecycle of resource assignments is not completely under the control of the hosting component container. Instead the control over the assignment is shared between the component container and the corresponding resource manager. Similar to component instances, the container signals the resource manager when to issue resource assignments and when their execution should be suspended. Furthermore, the container also signals that a certain resource assignment is no longer needed. This is done by the corresponding start, pause and stop signals depicted in Figure 10. However, in cases where a resource or parts of a resource become unavailable, the resource manager can revoke a resource assignment without further notice by invoking a revoke method. This shared responsibility is necessary to cover cases where a resource or some parts of it become unavailable. An example for such a case can be a resource that represents a USB webcam. As long as the webcam is plugged into the computer, it can be assigned to some component. However, since the webcam may be unplugged at any point in time without further notice, it is not possible to hide the unavailability of an issued assignment.

2.3.4.2 Resource Managers

Just like resource assignments are the resources' equivalent to instances of components, resource managers are the pendants to component factories. Thus, they are the local representation of a resource installed in the component container. They take care of creating resource contracts and they are responsible for issuing resource assignments. In addition, they are also responsible for revoking resource assignments when the corresponding resource is no longer available.

Just like a component factory, a resource manager can create zero or more contracts in response to a request for a resource by the component container. Yet, since resources can represent physical resources that can be strictly limited, resource managers need to perform additional tasks. In order to support the automated reasoning about potential conflicts between different resource assignments in a generalized way, the resource manager is responsible for mapping the amount of available resources as well as the amount of physical resources required in order to issue an assignment with a certain resource contract to a non-negative integer value. As a result, the component system can check whether two assignments can be issued simultaneously by summing up the required amount of resources and comparing it with the available amount of resources. Intuitively, for this to work correctly, the mapping performed by the resource manager needs to ensure that a set of resource assignments with a certain set of resource contracts can be issued as long as the amount of available resources is larger than or equal to the sum of the resources required by the set of assignments.

It is noteworthy that this kind of mapping is primarily targeted at resources that are limited and that do not have a relevant identity. Typical examples for such resources are main memory and network bandwidth. For main memory, it is usually considered to be irrelevant which part of a memory chip is assigned to an application. Similarly, for network bandwidth it is irrelevant which bits are used to guarantee a certain rate of bits per time interval. However, using simple workarounds the overall approach can easily extended to cover unlimited resources as well as resources where the identity matters. For example, in order to support resources that are not limited or that can be virtualized easily, a resource manager can simply return 0 as integer value for the required amount of resources. Similarly, in order to support resources with a relevant identity, a component container can be equipped with multiple resource managers for the same type of resource where each resource manager is responsible for managing an individual identity.

2.3.5 Applications

The PCOM component system is geared towards hiding many details of distribution from the application developer. From an application developer's perspective, the ultimate goal is to create the illusion of a static execution environment that contains sufficient resources and suitable services. While it should be clear that such a goal can only be approximated in general, the overall idea is the main rationale behind the application model. To achieve this goal as far as possible, the application model closely approximates the guarantee that a started component instance does not have unsatisfied requirements. If this cannot be guaranteed, the component instance should not be started.

2.3.5.1 Application Model

By extending this concept from a single component to a set of components recursively, PCOM defines an application as a tree of component instances and resource assignments that is spanned by some root component instance. This root component instance represents an application core and thus, it is the only part of the tree that does not fluctuate at runtime. Due to the fact that the root component instance does not change at any time during its execution, it is also called the application anchor. The availability of the other component instances and resource assignments may fluctuate unexpectedly at runtime. Thus, the exact composition of the tree might need to be adapted dynamically in order to ensure that each recursively specified contractual demand is satisfied by some matching provision.

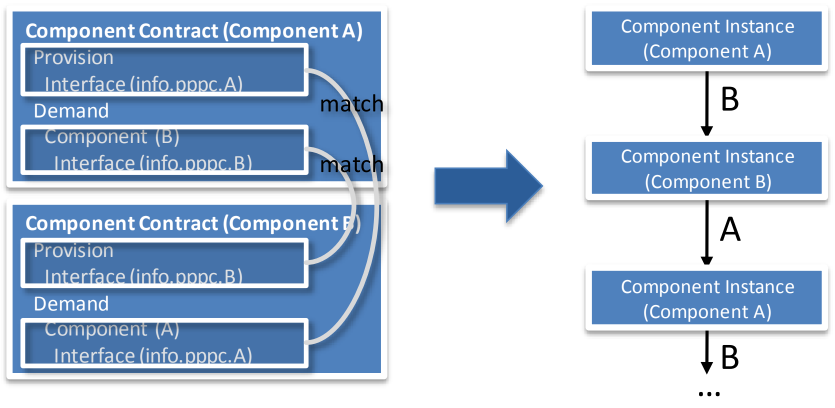


Figure 11 – Cyclic Component Contracts

Clearly, this type of recursive definition is not guaranteed to terminate in cases where a demand contained in a component contract can be satisfied – either directly or indirectly – through the same component contract. An example for such contracts that may result in such infinite cycles is depicted in Figure 11. There, the demand of a contract created by component A can be satisfied with the provision of a contract created by component B. The demand of this contract can, in turn, be satisfied by the provision of the contract of component A. In order to avoid the problems associated with non-terminating definitions, PCOM prohibits such definitions conceptually. However, since cycles can also be the result of unrelated component developments, the PCOM component container also performs runtime checks to detect and to break cycles.

2.3.5.2 Application Anchor

The application anchor plays a special role in PCOM applications since it is the only component instance that is not started on behalf of another component instance. Instead, it provides stand-alone functionality that may require other functionality. As a result, it is the only component instance that does not have to satisfy a specific demand. However, from a conceptual point of view, it satisfies some demand. Namely, it satisfies the demand of the user that uses the application implemented by the application anchor.

Due to this observation, we decided to apply the same rules to application anchors as to other components instances. Instead of providing special models and abstractions, we require that an application anchor must specify a certain provision as well. This can be done by creating a virtual interface that is provided by the application anchor. Thus, in order to start a certain type of application anchor, we can specify a demand on the virtual interface. This allows us to express the fact that a user demands a certain type of application using the component demand part of a component contract.

Besides from minimizing the number of abstractions introduced by the component system, this has a number of additional advantages. First, as a side-effect of reducing the number of abstractions, it simplifies the implementation of PCOM. Secondly and more importantly, it allows the definition of application suites by specifying a virtual set of demands that must be fulfilled simultaneously and thus, it allows that the approach towards automatic configuration detailed in the next section can be used to configure multiple applications simultaneously. Finally, as we explain in the following, it can also be used to support user preferences during the configuration of an application.

2.3.5.3 Application Preferences

After unifying the concepts of component instances and application anchors using virtual interface declarations, we can use the features of the contract model to integrate user preferences into the configuration. To do this, an application that supports multiple modes of operation may declare these modes as different provisions using dimensions and properties that are usually used to express the non-functional characteristics of an implementation. By using specific demands that require a certain value for the properties, a user may express his demand to start an application in a specific mode of operation.

Using this approach of requesting a certain mode of operation, we can simply extend this to preferences on different modes by supporting the specification of an ordered list of demands that are optional. Clearly, there are many other possible approaches of modeling preferences between different modes. However, this approach has the benefit that it does not require any additional models. In fact, since component factories can specify multiple optional contracts that can be used to execute a component instance, such preferences can be integrated without requiring any additional mechanisms.

This is especially beneficial since it does not require specialized algorithms to deal with preferences. Instead, a configuration algorithm can simply try to configure the application with the demand that represents the highest preference. If the configuration fails, it can try to start the next demand and so on. Due to this reason, the approach towards automatic configuration that is detailed in Chapter 3 and Chapter 4 does not specifically discuss preferences.

However, it should be noted that this simple approach also introduces overhead that is linear with the number of optional contracts that represent the preferences. As we will briefly discuss in Chapter 8, this can be mitigated by taking the internals of contracts into account and by defining additional operators on contracts. Specifically, by defining an inclusion relationship on contracts, it is possible to revise more intelligent search strategies for some preferences that represent relaxations. For more details on these optimizations, we refer to this chapter.

2.3.5.4 Application Lifecycle

Intuitively, the lifecycle of the application anchor defines the lifecycle of the other component instances and resource assignments and thus, it defines overall lifecycle of the application. If the anchor is started, the application is started. If the anchor is stopped, the remaining component instances and resource assignments are no longer needed. Thus, they can be stopped as well in order to avoid unnecessary resource utilization. Since the dynamics of the execution environment cannot be hidden completely, the execution of the application anchor may need to be paused temporarily to minimize the time in which the application anchor is started despite the fact that the tree exhibits unsatisfied dependencies. This should be signaled to the all other component instances and resource assignments to avoid interactions between them during adaptation.

Due to the distributed nature of PCOM applications, the state transitions in the lifecycle of component instances and resource assignments cannot be triggered at the same instant of time. However, since component instances that are started may interact at any point in time with their child component instances and resource assignments, the state transitions to the STARTED state should also not be triggered in an arbitrary order. Similarly, the state transitions to the STOPPED and PAUSED states should also not be triggered in an arbitrary order as component instances that have been stopped or paused should not be required any longer by their parents.

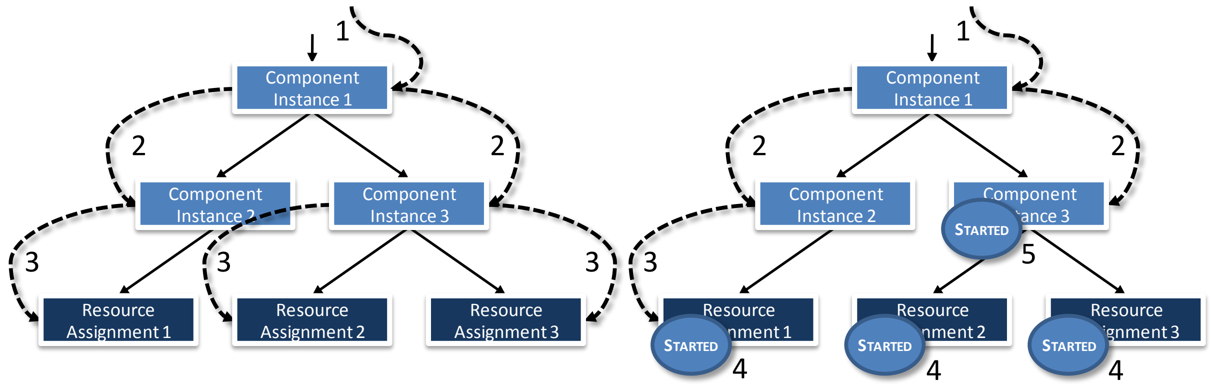


Figure 12 – Parallel Bottom-Up Lifecycle Transition

For transitions to the STARTED state this means that the component system should ensure that all (available) children have already made the transition when a parent is triggered. This can be achieved by triggering this state in a bottom-up fashion. If this would be done sequentially, it would correspond to a post-order traversal of the tree. However, in order to utilize the potential parallelism resulting from the distribution of the application, it is favorable to perform the state transition in parallel in different sub-trees. As shown in Figure 12, this can be done by initiating a series of synchronous triggers in parallel to all children that need to be completed before the trigger of their parent is called.

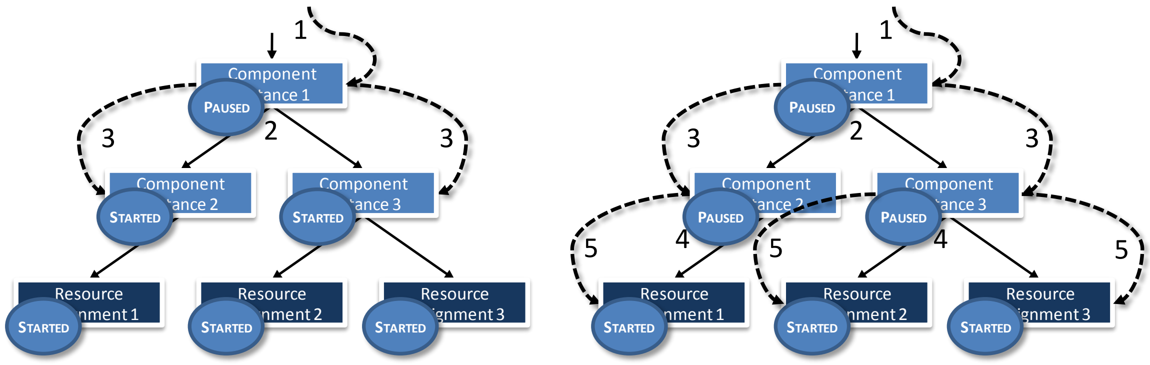


Figure 13 – Parallel Top-Down Lifecycle Transition

In contrast to transitions to the STARTED state, which should be triggered bottom-up, transitions to the STOPPED and PAUSED state should be triggered in a top-down fashion. This ensures that the parent is no longer interacting with the children when they are paused or stopped. If done sequentially, this would correspond to a pre-order traversal of the tree, but similar to transitions to the STARTED state it can also be done in parallel as depicted in Figure 13.

2.3.6 Adaptation

After having defined the concepts of the component system that are used to support the model of pervasive applications introduced in Section 2.2.1, we now present how these concepts can be applied to support automatic adaptation. As motivated in Section 2.2.2, the basic principle of our approach towards supporting automatic adaptation at the system level is to utilize parameterization whenever possible and to use reconfiguration as a last resort. In the following, we first describe how parameterization and reconfiguration can be supported using the previously introduced concepts. Thereafter, we classify the reasons for adapting a PCOM application and we briefly outline how the need for adaptation can be detected automatically. Using this classification, we discuss cases in which parameterization and reconfiguration should be applied and we describe the scope that need to be considered when adapting an application.

2.3.6.1 Parameterization

PCOM component instances and resource assignments rely on contracts to capture their functional and non-functional dependencies in a way that enables the automated matching. As presented in Section 2.3.2, a contract models a conditional statement that guarantees that a certain provision can be achieved if all specified demands can be fulfilled.

According to the contract model, the provision is always representing a point in the parameter space spanned by the non-functional properties. With respect to the demands specified by a contract, a contract may denote either points or regions in the parameter space. Although this provides certain flexibility, the definition of a single region alone is not sufficient to cover all cases. Specifically, in cases where a component implementation can make tradeoffs between the values of different properties in order to fulfill its provision, it may need to specify a number of regions that cannot be merged into a single connected region. As a simple example, consider a component that requires that the sum of two properties – *A* and *B* – should add up to some value *V*.

In order to support the specification of different regions of the parameter space, the component system enables component factories and resource managers to specify multiple optional contracts that can be used to satisfy the same demand. So given the previous example, a component could specify several possible combinations such *A=V* and *B=0*, *A=V/2* and *B=V/2* or *A=V/4* and *B=3V/4*, etc. According to the conditional statement represented by a contract, the implementation must be able to provide a matching provision if all the requirements of one contract can be met and each contract is equally well suited. Thus, by definition, a correct component implementation must be able to support each of the specified contracts. As a result, the specification of optional contracts essentially entails the explicit definition of possible alternative parameterizations of a component instance or resource assignment. Due to this fact, the component system can switch between the different contracts in order to parameterize a component instance or resource assignment differently.

In addition to this explicit parameterization, there is also an implicit parameterization which is a result of the fact that a demand may specify regions whereas a provision may only specify a point in the corresponding parameter space. Although, the point denoted by the provision of a child will – by definition – always lie within the region spanned by the demand of its parent, the exact location of the point may depend on the provisions of its children. Thus, a contract that uses regions within a demand will usually model a potentially unlimited number of possible parameterizations that are hidden from the component system.

In order to avoid that a potentially high number of contracts needs to be considered on an individual basis during the configuration of an application, the component system allows component factories and resource managers to specify wildcards in provisions during the configuration of an application. Since the resulting parameterization is not explicit, switching between these parameterizations cannot be done by the component system. Instead, this requires the cooperation of the corresponding component instance. However, in order to switch to the right parameterization, i.e. in order to compute the exact provided point in the parameter space, the component instance needs to know the provided points of its child component instances and resource assignments.

As a result, the parameterization of a component may also be changed in cases where the provision of child component instance or a resource assignment changes. In these cases, the component system cannot control the new parameterization. However, due to the definition of contracts, the resulting point must always lie within the region demanded by the parent in cases where the children specify solely matching points.

2.3.6.2 Reconfiguration

In addition to supporting different parameterizations, the PCOM application model also supports the reconfiguration of the overall application at runtime by changing the set of component instances and resource assignments that constitute an application. However, in contrast to changes to the parameterization of components instances, changing the set of component instances and resource assignments is a more complex undertaking that requires further considerations due to the following two reasons.

First, two component instances that declare the same provision might have vastly different demands. Thus, replacing a single component instance might not only require the replacement of its children with similar component instances and resource assignments, but it might also require a completely different set of children in terms of the functionality provided by them and the resources represented by them. As a result, changing a single component instance can induce the need for further reconfigurations. It is noteworthy that some reconfigurations could be done in isolation. As an example consider the replacement of a component instance hosted by one component container with the same implementation hosted by another component container. However, if we would restrict adaptation to such reconfigurations, we would severely restrict the compositional flexibility of the overall application model.

Secondly, since component instances may contain application-specific state, changing the set of component instances requires additional care in order to ensure that the state is migrated as well. However, there are two issues that complicate such a migration. First of all, since a replaced component instance may require a different set of children, the component system cannot know how the application-specific state that was held by the individual component instances of the initially bound sub-tree needs to be migrated to the individual component instances of newly bound sub-tree. Secondly, since we want to apply reconfiguration to cope with the unforeseeable unavailability of component instances, there is no guarantee that a component instance that is replaced during the course of a reconfiguration is actually available. Thus, without further precautions the state of a component instance that becomes unavailable at runtime might not be recoverable. Clearly, these problems could be avoided by prohibiting component implementations that carry application-specific state, but introducing such a restriction could have significant negative performance implications.

To enable the reconfiguration in such a way that it supports different structural configurations of the functionality and that it can take care of the application-specific state held by the corresponding component instances, we need to extend the concepts of the component model that have been introduced previously. The extensions are an immediate response to the problem that dealing with the application-specific state encapsulated in component instance requires further application knowledge. In order to externalize this knowledge, we split the responsibility of dealing with the application-specific state between the application developer and the component system. Since our goal thereby is to provide a generic but efficient solution, we introduce the following additional requirements on component implementations.

First of all, an application developer that is designing the interface of a new component additionally needs to define how the internal state of the functionality can be represented independently from the implementation. Each component instance that implements the interface must be capable of loading and storing this state. It is noteworthy that a component instance does not have to use the same representation internally. It only has to perform the mapping between its internal representation and the predefined generic representation. As a result, the system can migrate the state between different implementations of the same functionality. To do this, it stores the state of the old component, transfers it and loads the state into the new component.

Secondly, we require that each component instance is capable of restoring the state of the sub-tree spanned by it using the state stored in the implementation independent representation. This way, we can avoid the problem that the component system does not know how to distribute the state that was held by other sub-trees. Due to this requirement, we can additionally simplify the reconfiguration process by always replacing the complete sub-tree spanned by a component instance in cases where it is replaced. Clearly, doing that can increase the amount of time and resources that are required to reconfigure an application. At the same time it also eases the task of the application developer as a component instance can simply restore the state of its children whenever it is loading its client-specific application state. Furthermore, this also ensures that the restoration can be performed consistently in cases where a component instance replaced because it is no longer available.

Using these two extensions, the component system can migrate the state between different component implementations that are providing the same functionality and it can ensure that the state of a replaced component instance can be restored as long as the component instance is available to retrieve the state during migration (Handte, Schiele, Urbanski, Becker, & Rothermel, 2005). In order to support cases where a replaced component instance is no longer available, the component system needs to keep track of the changes to the internal state of the component instance continuously. To do this in a generic way, the component system can proactively store the state held by each component instance whenever it changes and it can transfer this state to the component container that hosts the parent instance. As a result, the component container of the parent component instance is able to restore the state at any point in time without requiring the presence of the component container that hosts the child. However, this simple approach can lead to high performance overheads in cases where the state is comparatively large and where changes to the state occur frequently.

In order to mitigate this, we do not store the state continuously for all component instances. Instead, we introduce a supplemental mechanism that works well for component instances that exhibit the behavior of a state machine (Schneider, 1990). In interactive applications, this type of components is rather common. Specifically, most components that provide output-related functionality exhibit this characteristic. Within such component instances, the internal state solely depends on the sequence of calls that they have received from their parent instance. Thus, we can restore the state either by setting it directly using a previously stored state or by replaying the exact sequence of calls that they have received from their parent. Furthermore, we can also restore the state using a combination of both where we directly set the state to some intermediary state and then apply the remaining sequence of calls that have been made after the state has been stored. To do this, we log the sequence of calls performed by the parent component instance and occasionally, we store the state of the child component instance and remove all previously logged calls. In order to increase the efficiency of this mechanism even further, we can allow the application developer to manipulate the logged sequence of calls and we enable the developer to record the internal state of component instance whenever needed.

2.3.6.3 Changes

According to the underlying system model and the abstractions introduced by the component system, we can classify the changes that may occur in the system along different orthogonal axes. In the following, we consider the source of the change and the effects of the change on the application configuration.

First of all, changes can be either a result of the fact that the underlying system has changed or that the requirements on the application have changed. The former is a result of the changing availability of resources on a single component container and the changing availability of component containers that are part of the same smart peer group. The latter is a result of changing user preferences that are directed towards an executed application.

Furthermore, changes may either invalidate the application configuration or they might leave the configuration intact. Changes that invalidate the application configuration can be changes to the user preferences that are not fulfilled by the current configuration and changes to the available resources or component instances that break one or more contracts. Changes that leave the application configuration intact can either be changes to the preferences that are met by the current configuration and changes to the availability of resources that can be compensated.

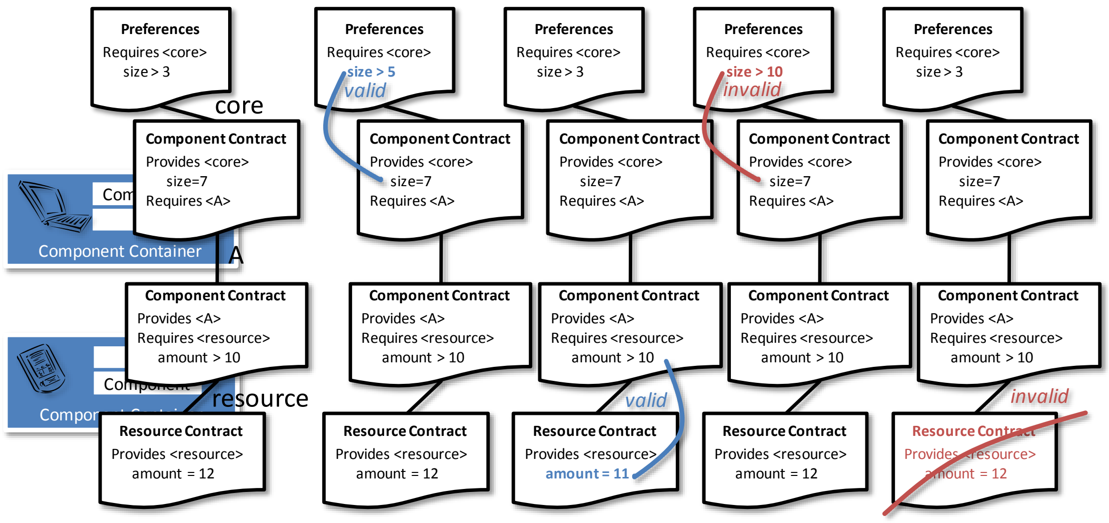


Figure 14 – Effects of Changes

Figure 14 shows the basic categories and their effects on an exemplary application configuration. The configuration consists of two components, namely the application core and component A. Component A requires a certain amount of some resource in order to function properly. The changes are (from left to right) a change to the preferences and a change to the available resources that do not invalidate the configuration as well as a change to the preferences and a change to the available resources that invalidate the configuration.

With this set of possible changes, it should be clear that most changes can be detected by monitoring changes to contracts. Usually, this can be done locally by the component container that hosts the corresponding component instance. If a contract is modified, for example, by a resource manager that revokes some resource assignment, the component container can notify the affected component instance. In general, this holds true for all changes except for changes that result from the unavailability of a computer. If a computer leaves the smart peer group, it can no longer contact the remaining computers and thus, it cannot inform the affected component containers.

In order to detect such cases, a component container that relies on component instances provided by some other component container needs to monitor its availability continuously.

Similarly, a component container that provides component instances for some other container needs to monitor its availability in order to release stale component instances. Both, monitoring and avoiding stale state, can be done by relying on soft-state approach. To implement this, the containers are periodically exchanging messages to ensure that the required component instances are still available and that the provided component instances are still needed. If the periodic message exchange fails, the providing component container can stop the unused component instances and the other component container can detect and signal the broken contract.

2.3.6.4 Utilization

Changes in the smart peer group that do not affect the validity of a configuration are a natural consequence of contracts supporting certain ranges as part of their requirement specifications. If such changes occur, it is not necessary to modify the configuration. However, since the provision specified by a contract may depend on the individual provisions specified by the required components and resources, it is necessary to propagate the changes in order to enable component instances to update their provisions accordingly. Thus, a change to a resource might trigger recursive modifications to provisions that ripple through the configuration from bottom to top. Yet, a change that does not break a contract at a lower level of the configuration does not lead to broken contracts on higher levels of the tree. According to the differentiation of parameterization in explicit and implicit presented in Section 2.3.6.1, these changes can be thought of as handled by implicit parameterization.

Changes that affect the validity of cannot be handled by implicit parameterization as they break at least one contract of the configuration. As a consequence, they must be handled by changing one or more contracts of the configuration, i.e. by explicit parameterization, or by changing some of the component instances, i.e. by reconfiguration. Although, parameterization is usually preferable over reconfiguration, there are cases in which reconfiguration cannot be avoided. Besides from the obvious case that some required component instances are no longer available at all, reconfiguration may also be required in cases where all component instances are still available. As an example consider a required resource that becomes unavailable at runtime. Although, all component instances are still available, it might be necessary to replace a component instance with some instance on another computer to ensure that the resource requirements can be met. As a result, it is not possible to associate a particular adaptation mechanism to the remaining changes that cannot be handled by implicit parameterization. Instead, the decision of whether explicit parameterization or reconfiguration should be applied must be performed dynamically at runtime on the basis of the properties of the configuration and the capabilities of the smart peer group. In Chapter 4, we describe our approach to perform these decisions as an extension to the configuration algorithm presented in the next chapter.

2.3.6.5 Scope

Now that the reasons and the possible ways of adapting a PCOM application are defined, the only remaining issue that has yet to be determined is the scope. Intuitively, it seems more efficient to limit the scope of an adaptation as this minimizes the effects on the unaffected parts of a configuration. However, as we discuss in the following, it is usually not possible to define a meaningful scope a priori – other than the complete configuration.

For changes that can be compensated by implicit parameterization, a meaningful local scope can be identified easily. For instance, if the resource availability on some computer changes, it is sufficient to notify the component instance that utilizes the resource. If the modified resource availability does not induce a change to the provision of the component instance, it is not necessary to perform further notifications. If the contract of the instance changes, it is sufficient to notify the parent as it is the only part of the configuration that could be affected, and so on. As a consequence, the effects of changes that can be compensated implicitly are limited to the set of ancestors in a configuration.

For other changes, i.e. the changes that affect the validity of a configuration, it is not possible to identify a smaller scope than the overall configuration. The reason for this is that the attempt to deal with them might require multiple changes to various parts of the configuration. This, in turn, is a result of the fact that modifying a contract may lead to modified resource requirements and thus, it can lead to arbitrary conflicts. As an example, consider a configuration in which some sub-tree has become unavailable. Although, it might be possible to find a set of component instances to replace the sub-tree, the set of instances will typically introduce new resource requirements. In order to fulfill these, it can be necessary to reduce the resource requirements in order parts of the configuration, e.g. by changing the parameterization or by replacing the existing component instances as well. Since it is not possible to restrict the scope of adaptation in general, it is necessary to consider the complete configuration as soon as a single contract needs to be modified. The approach towards automatic adaptation detailed in Chapter 4 takes this into account by computing a complete configuration whenever a contract needs to be changed. Thereby, it introduces a cost model to capture the costs of different alternative adaptations that is geared towards minimizing the effects on parts of the configuration that are not directly affected by a change.

2.4 Discussion

In this chapter, we derived the requirements on system software that enables the automatic adaptation of a pervasive application in smart peer groups. Based on the characteristics of the underlying system model, we have identified the requirements on minimalism, extensibility and decentralized operation. Due to the goal of automating adaptation at the system level, we have furthermore identified the requirements on a flexible and explicit application specification, continuous application monitoring and high adaptation transparency.

By applying these requirements to the widely used service-based model for building distributed applications in which a non-distributed application makes use of local and remote services, we have revised a component system for smart peer groups. This component system can be used to automate the initial configuration as well as the runtime adaptation. The component model introduced by this system is minimal as it solely introduces abstractions that are necessary to support the automatic configuration and adaptation using the targeted application model. Since the component system does not impose any restrictions on the functionality and the granularity of components and resources and due to the fact that the component container may host an arbitrary set of components and resources, the system can be customized to the capabilities of different computers, ranging from resource-poor specialized systems up to powerful general purpose computers. The mechanisms that are needed to execute an application do not depend on a single system and as we discuss in more detail in Chapter 5, it is comparatively simple to implement them in a distributed fashion on top of existing communication middleware.

In order to explicitly specify an application, the component model utilizes contracts. Contracts describe the syntactical and non-functional properties of a component in an abstract manner and it is possible to provide various implementations for the same contract. Due to the fact that the application model is defined recursively on the basis of contracts, it allows a great deal of compositional flexibility. As a consequence, it is possible to compose a complete application as well as individual components from vastly different sets of components and resources. This enables the introduction of new applications that reuse existing components and it also enables the continuous evolution of applications, components and resources. In order to support the efficient adaptation of an application, the component model supports parameterization. To enable parameterization, a component may specify contracts with ranges of allowable values as well as multiple optional contracts to model alternative modes of operation. In order to cope with the unavailability of resources and components at runtime, the component model supports reconfiguration. Due to the recursive definition of a configuration, the reconfiguration is not limited to replacing components with identical ones. Instead, each component that can provide a matching functionality can be used interchangeably at any point in time.

Due to the explicit application specification, the component system can automate the initial configuration of an application. As a consequence, the application developer does not have to provide program logic that deals with composition. Furthermore, to enable a high degree of adaptation transparency, the component system takes care of continuously monitoring relevant changes to contracts and changes to the availability of required components. If a relevant change occurs, the component system notifies the affected components so that they can update their provisions. If the change requires more complex modifications, such as changing a contract or a component, the component system automatically computes a new and valid configuration. Thus, instead of developing the program logic for configuration and adaptation as part of each application, the developer solely needs to specify possible modes of operations for the individual components as contracts and implement the component with the pre-defined lifecycle. In order to simplify the consistent management of application-specific state during an adaptation, the component system provides mechanisms to migrate the state of individual components, if necessary.

As a result of these simplifications for the application developer, the component system is fully responsible for computing valid configurations, for detecting invalid configurations and for computing and performing appropriate adaptations. As we show in the following chapters, both tasks are non-trivial and computationally expensive, since they need to consider the global configuration. Yet, they are necessary to support the application model of the component system which is, in turn, based on a service-based model that is widely-used. In contrast to the proposed component system, however, configuring and adapting the composition of services is usually left as a task for the application developer which implies that the developer must solve the resulting problems in a satisfying way for each individual application. In the following two chapters, we describe how configuration and adaptation can be handled automatically in a generic fashion. Thereafter, we describe a prototypical implementation of the overall component system and we evaluate it using an exemplary application, simulations and experiments. Finally in Chapter 7, we compare it with other existing system software and in Chapter 8, we discuss possible future extensions.

3 Automatic Configuration

This chapter introduces and formalizes the configuration problem introduced by PCOM applications. It discusses the overall problem complexity and derives the requirements on approaches for automatic configuration. Consecutively, the chapter presents our approach for automatic configuration (Handte, Becker, & Rothermel, 2005). The basic idea is to interpret the configuration problem as a constraint satisfaction problem. Using this interpretation, we can apply arbitrary algorithmic solutions for solving such problems to automatic configuration. From the set of possible algorithms, we show how asynchronous backtracking can be modified to meet the requirements on automatic configuration. Thereby, we discuss a number of problem specific optimizations to the basic algorithm that can be used to reduce its communication overhead. Finally, we close the chapter with a discussion.

3.1 The Configuration Problem

Automatic configuration denotes the task of automatically determining a composition of components that can be executed simultaneously as application. Such a composition is subject to two classes of constraints. The first class is given by structural constraints. They describe what constitutes a valid composition in terms of functionalities. The second class is given by the resource constraints. They are a result of the resource requirements of components and the limited availability of resources on the individual computers that host them.

Structural constraints can be either specified in advance, e.g. as an architectural model expressed in some description language, or they can be individually associated with components, e.g. as dependencies contained in contracts. If an architectural model is used, the configuration must ensure the availability of a suitable component for each part of the model. If structural constraints are individually associated with components, the configuration must ensure that all recursively specified dependencies can be resolved with a suitable component. As discussed in the previous chapter, we associate each component with such contracts.

Resource constraints can be modeled in various ways. For the sake of simplicity, we rely on a simple but powerful model that is also used in (Xu, Nahrstedt, & Wichadakul, 2001). In this model, components specify their local resource requirements in advance as part of their contracts. Each contract contains an integer vector whose dimensions denote a specific resource. The actual requirements on the resource are represented by the corresponding value of the dimension. A requirement is specified by a positive value and thus, the vector does not contain negative values. Similarly, the available resources on each computer can be modeled as a vector with non-negative values. Since the availability of resources can change, the values might fluctuate at runtime. To satisfy the resource constraints, the configuration has to ensure that at any time the index-wise sum of all requirement vectors of local instances is index-wise less than or equal to the vector that specifies the locally available resources.

The complexity of automatic configuration arises from the fact that both, resource and structural constraints must be fulfilled simultaneously. Due to the recursive definition of structural constraints, it is not possible to calculate the resource requirements of a certain sub-tree in advance without determining all possible configurations of that sub-tree. But even if it was possible, the strictly limited resource availability might lead to exclusions between structurally possible configurations of arbitrary sub-trees. Thereby it is important to mention that finding all exclusions is as complex as finding a possible configuration in general.

3.1.1 Example

To illustrate the configuration problem, we briefly describe the process of automatic configuration with the previously introduced abstractions using a simple application in an exemplary smart peer group. As depicted in Figure 15, the smart peer group consists of three computers, i.e. a desktop, a personal digital assistant (PDA) and a laptop.

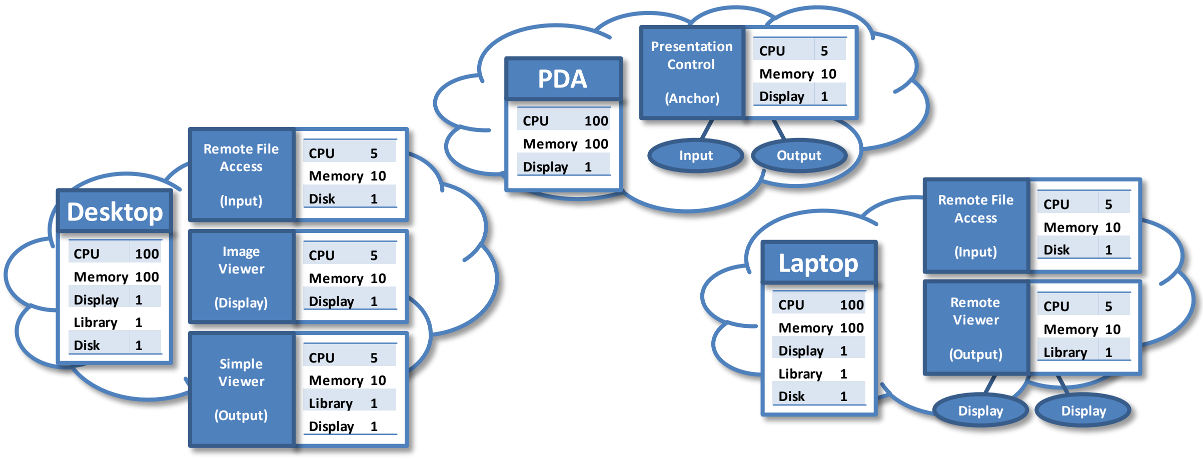
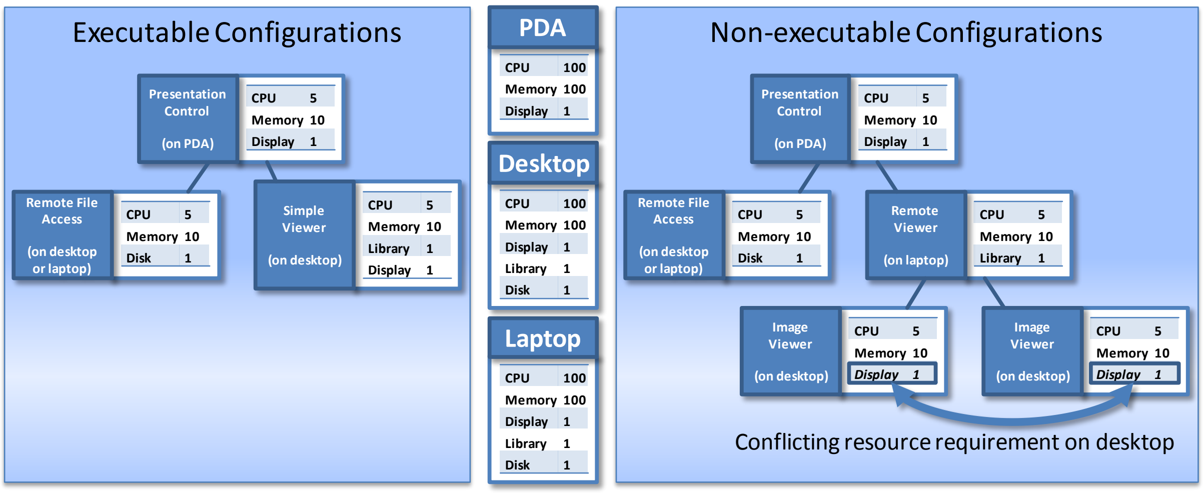


Figure 15 – Exemplary Smart Peer Group

Each computer of the group has a certain amount of resources. The desktop and the laptop are equipped with one *Display*, one *Library* and one *Disk* resource. In addition they possess a certain amount of processing power (*CPU*) and a certain amount of memory (*Memory*). The personal digital assistant is also equipped with a *Display* resource and a certain amount of memory as well as certain amount of processing power.

Each computer hosts some components that provide certain functionality. To simplify the description, we assume that each component solely provides a single mode of execution and thus, each component can be described using a single contract. Furthermore, we omit the dimensions in the vector that specify the resource demand whose value would be zero. Instead, we assign a name to clarify the mapping between the available amount of resources and the contractually specified resource demands.

The personal digital assistant hosts the application anchor (*Presentation Control*) that enables the user to display a presentation on some computer of the smart peer group. In order to be executable, the application anchor requires the *Display* resource, a certain amount of processing power and a certain amount of memory. Furthermore, it requires a functionality to load the presentation (*Input*) and a functionality to show the presentation (*Output*). The laptop hosts a component that enables a computer of the group to access its file system (*Remote File Access*) and another one that is capable of displaying a presentation (*Remote Viewer*). Each usage of this component requires *CPU*, *Memory* and access to the local *Library* resource as well as two components that provide the *Display* functionality. The desktop also hosts a component to access its file system (*Remote File Access*). In addition, it hosts a component that can be used to show images on the display of the desktop (*Image Viewer*). Furthermore, it hosts a component that is capable of displaying a presentation on the desktop (*Simple Viewer*).



Executable Configurations
Presentation Control (on PDA)
CPU 5
Memory 10
Display 1
Remote File Access (on desktop or laptop)
CPU 5
Memory 10
Disk 1
Simple Viewer (on desktop)
CPU 5
Memory 10
Library 1
Display 1
PDA
CPU 100
Memory 100
Display 1
Desktop
CPU 100
Memory 100
Display 1
Library 1
Disk 1
Laptop
CPU 100
Memory 100
Display 1
Library 1
Disk 1
Non-executable Configurations
Presentation Control (on PDA)
CPU 5
Memory 10
Display 1
Remote File Access (on desktop or laptop)
CPU 5
Memory 10
Disk 1
Remote Viewer (on laptop)
CPU 5
Memory 10
Library 1
Image Viewer (on desktop)
CPU 5
Memory 10
Display 1
Image Viewer (on desktop)
CPU 5
Memory 10
Display 1
Conflicting resource requirement on desktop

Figure 16 – Possible Configurations

If the application is about to be started, the personal digital assistant must assign sufficient resources to the component and it must resolve the dependencies (*Input* and *Output*). To resolve the dependencies, it needs to find components on the computers of the smart peer group that are capable of delivering compatible provisions. Thereafter, it can decide to use one of the possible components to satisfy the dependency. If a component is selected, this component needs to be configured recursively by assigning the required amount of resources and resolving all of its contractually specified dependencies using a matching contract of some component.

In this example, the *Input* dependency can be resolved using a component instance of the *Remote File Access* on the laptop or on the desktop. The *Output* dependency can be resolved by the *Remote Viewer* component on the laptop or the *Simple Viewer* component on the desktop. If the personal digital assistant uses the *Remote Viewer*, the laptop must assign the resources and resolve the *Display* dependencies. To do this, it can only use the *Image Viewer* component on the desktop.

Altogether, there are four structural possibilities to configure the application depending on the choice for the *Input* and *Output* dependency as shown in Figure 16. Since the *Image Viewer* component can only be used once due to the limitation of *Display* resources, there is no way of using the *Remote Viewer* component in such a way that all resource requirements are met. The two executable configurations use a *Remote File Access* component on the desktop or on the laptop and the *Simple Viewer* component on the desktop.

Besides from illustrating the configuration process using the previously introduced abstractions, this example also nicely demonstrates the interrelation of resource and structural constraints. Choosing an instance that represents a locally valid option such as the *Remote Viewer* can induce conflicting resource requirements later on. Such conflicts can only be discovered gradually since they require the computation of parts of the configuration along the structural dependencies.

3.1.2 Formalization

To formalize the configuration problem, we need to introduce some standard definitions from graph theory. For a tree $G = (E,V)$ consisting of the nodes $v \in V$ and the directed edges $(v_i, v_j) \in E$, we use the function $\eta(v) : V \to V \cup \{\}$ to denote the parent of node v which will be $\{\}$ if and only if v is the root, i.e. $\neg\exists(v_j, v_i) \in E : v = v_i$, and some other node $v_j \in V$ for all other cases where $\exists(v_j, v_i) \in E : v = v_i$. To denote the transitive closure of $\eta(v)$, we use the notation $\eta^*(v)$.

Furthermore, we use the following definitions and simplifications to represent the abstractions introduced by the component system. Let $\mu(d, p) : D \times P \to \{true, false\}$ be the Boolean matching function between the infinite set of contractual component demands D and the infinite set of contractual provisions P. Without loss of generality, let c_i be a contract with the provision $\pi(c_i) = p_i \in P$, the set of n component demands $\delta(c_i) = \{d_{i,1}, \dots, d_{i,n}\} \subset (D \cup \{\})$ and the m resource requirements $\rho(c_i) = (r_{i,1}, \dots, r_{i,m}) \in N_0{}^m$. Furthermore, let m_j be a computer that has k available resources $\alpha(m_j) = (a_{j,1}, \dots, a_{j,k}) \in N_0{}^k$ and that host components which can support the (possibly infinite) set of contracts $C_j = (c_{j,1}, c_{j,2}, \dots)$. As a simplification, we assume that the n-tuples of the resource demand and the resource provision have the same dimensionality, if the contract is provided by a component on the computer, i.e. $c_i \in C_j \Rightarrow m = k$.

Finally, in order to represent the resource model, we use the following definitions. For two n-tuples $a = (a_1,...,a_n)$ and $b = (b_1,...,b_n)$, we define $a \le b : N_0{}^n \times N_0{}^n \to \{true, false\}$ as component wise comparison $a_1 \le b_1 \wedge ... \wedge a_n \le b_n$ and we define $a + b : N_0{}^n \times N_0{}^n \to N_0{}^n$ as component wise addition $(a_1 + b_1,...,a_n + b_n)$ of the tuples.

Using these definitions, we can formalize the configuration problem as follows. Given a set of l computers that form a smart peer group $M = \{m_1,...,m_l\}$ and the contract of an application anchor $c_{anchor} \in C := C_1 \cup .. \cup C_l$ find a directed tree $G = (E, V)$ with a function that maps nodes to contracts $\chi(v) = c : V \to C$ such that the following five conditions hold true:

(1) $\exists v \in V : \eta(v) = \{\} \wedge \chi(v) = c_{anchor}$

This condition ensures that the right application is configured by enforcing that the root node of the tree must represent the contract of the application anchor.

(2) $\forall (v_i, v_j) \in E \; \exists d \in \delta(\varepsilon(v_i)) : \mu(d, \pi(v_j))$

Condition two ensures that an edge may only be contained if the contract represented by a source node has a component demand that can be fulfilled by the provision of the contract of the target node.

(3) $\forall v_x \in V \; \forall d \in \delta(\chi(v_x)) \; \exists! (v_x, v_y) \in E : \mu(d, \pi(\chi(v_y)))$

Condition three ensures that each component demand of a contract represented by a node requires the presence of exactly one edge in the tree. Together with condition two, which prohibits the existence of edges that do not represent a demand and a matching provision, this ensures that the tree consists solely of edges that represent demands and each demand is represented once.

(4) $\forall v \in V : \chi(v) \notin \chi(\eta^*(v))$

Condition four ensures that the contracts do not cause cycles. To do this, it enforces that the same contract is not used twice on the path to the root of the tree. Together, the previous four conditions enforce the structural constraints defined by the application model are fulfilled for the right application, i.e. the contract of the application anchor.

(5) $\forall V_i = \{v_j \mid v_j \in V \wedge \chi(v_j) \in C_i\} : \alpha(m_i) \leq \sum_{j=1}^{|V_i|} \rho(\chi(v_j))$

Finally, condition five ensures that the resource constraints are met by enforcing that the sum of the resource requirements of contracts on each computer does not exceed the available resources on it.

3.1.3 Complexity

In the following, we show that the automatic configuration problem as discussed above is NP-complete. To do this, we first show that there exists a non-deterministic polynomial time algorithm to solve the configuration problem. From this, we can conclude that the configuration problem lies in NP. Furthermore, we show that we can reduce 3-SAT (Cook, 1971), i.e. a special form of Boolean satisfiability in which the input is restricted to a conjunctive normal form that exhibits at most three literals per clause, using a polynomial time algorithm to automatic configuration. To do this, we embed 3-SAT into automatic configuration. Since 3-SAT is known to be NP-hard, we can therefore conclude that automatic configuration must be at least NP-hard as well. By combining these two results, we can conclude that automatic configuration must be NP-complete.

Without loss of generality, we assume that the configuration problem consists of the smart peer group $M = \{m_1,...,m_l\}$ and the contract of the application anchor c_{anchor}. Then the total number of resources is given by the problem-specific but constant value $r = \sum_{i=1}^{l} |\alpha(m_i)|$. Starting from the application anchor, we construct a directed tree by recursively adding nodes along the dependencies that represent contracts in a non-deterministic fashion. Again without loss of generality, assume that the resulting solution consists of a directed tree $G = (E,V)$ and the function χ. Now, we initialize an integer vector of length r with the available amount for each resource and we traverse the tree spanned by the contracts. In every step, we subtract the resource requirements defined by the contract from the corresponding dimensions of the vector and we check whether the edges for the node fulfill the requirements, i.e. resolve the dependencies. If one of the dimensions of the vector becomes negative or if the edges of the tree are invalid, we can abort the algorithm since the solution is not correct. If the traversal has reached all nodes and the algorithm did not abort, the solution is correct. The complexity of this algorithm is $O(r|V|)$ for validating the resources and $O(|V|+|E|)$ for validating the dependencies. Since $G = (E,V)$ is a tree $|E| = |V| - 1$. Thus, the overall algorithm is $O(r|V|)$.

To perform the second step, we reduce 3-SAT to automatic configuration using a polynomial time algorithm by means of embedding. To do this, we construct a specific instance of a configuration problem for every possible conjunctive normal form of 3-SAT. Assume without loss of generality that the conjunctive normal form consists of c clauses with at most three literals. As a result, there is a linear relationship between the number of literals and the number of clauses, i.e. the total number of distinct literals can be at most $3c$.

We begin the reduction algorithm by constructing c resources for each distinct literal and we set the amount of available resources for each resource to c. Then, we create one contract for each distinct literal and c contracts that represent its negation. Thereby, we set the resource requirements of the contract that represents the literal to one for each of the corresponding c resources. For each of the contracts that represent a negation, we set the resource requirements to c units of one of the resources. This ensures that a valid configuration can either contain contracts that represent the literal or contracts that represent its negation, but not both. In addition, it ensures that we can use the literal or its negotiation c times which is the maximal number of occurrences of the literal in any conjunctive normal form.

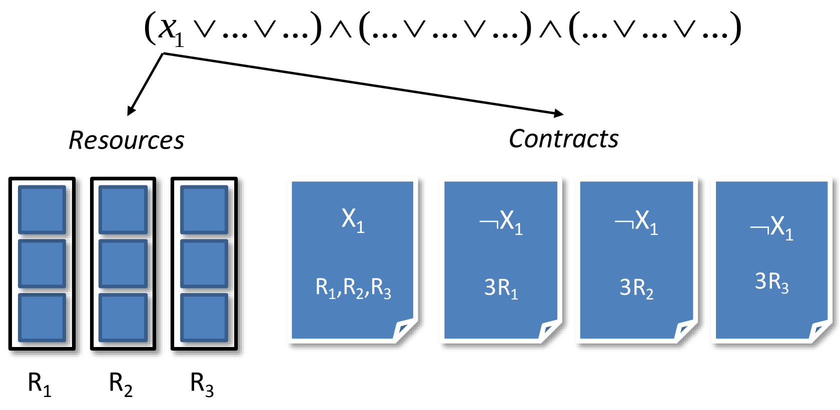

$(x_1 \vee \ldots \vee \ldots) \wedge (\ldots \vee \ldots \vee \ldots) \wedge (\ldots \vee \ldots \vee \ldots)$
Resources
Contracts
X_1
R_1,R_2,R_3
$\neg X_1$
$3R_1$
$\neg X_1$
$3R_2$
$\neg X_1$
$3R_3$
R_1
R_2
R_3

Figure 17 – Reduction Procedure for Literals

The result of this construction for a literal of conjunctive normal form with 3 clauses is depicted in Figure 17. In this example, the contract representing x_1 requires one resource of R_1 and R_2 and R_3. The contracts representing the negation of x_1 require either three resources of R_1, R_2 or R_3. Since each resource can be used three times, the contract x_1 can be used three times and each of the contracts representing the negation can be used once. More importantly, however, the construction guarantees that x_1 and its negotiation can never be used at the same time as this would require at least four resources from R_1 or R_2 or R_3, depending on the choice of the contract for the negation.

Now, we construct an application anchor with c distinct dependencies. These dependencies represent the individual clauses of the conjunctive normal form that must be fulfilled simultaneously. Finally, we create contracts that map each dependency of the application anchor to the literals or their negotiation according to the corresponding clause of the conjunctive normal form. These contracts represent the individual entries of the clause that can be used to satisfy the complete clause. An example of this construction is shown in Figure 18.

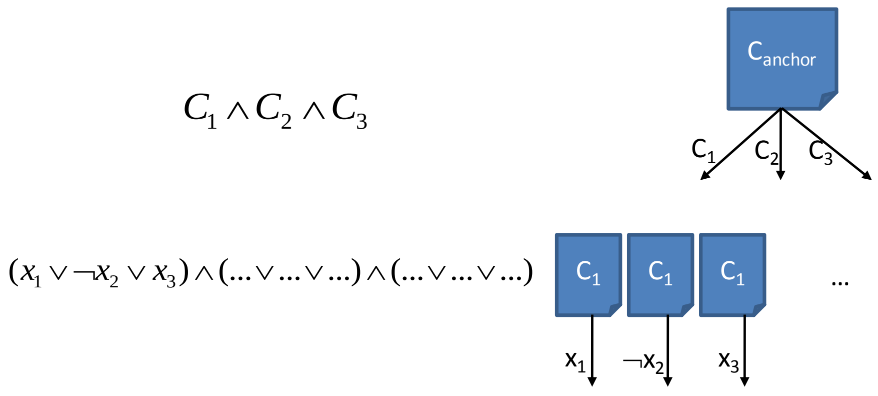

$C_1 \wedge C_2 \wedge C_3$
C_{anchor}
C_1
C_2
C_3
$(x_1 \vee \neg x_2 \vee x_3) \wedge (\ldots \vee \ldots \vee \ldots) \wedge (\ldots \vee \ldots \vee \ldots)$
C_1
C_1
C_1
...
x_1
$\neg x_2$
x_3

Figure 18 – Reduction Procedure for Clauses

A solution to the conjunctive normal form can now be found by computing a valid configuration and by looking at the literals represented by the leaf contracts of the configuration. Since the specific construction of the resource dependencies prohibits the simultaneous use of contracts that represent positive and negative assignments for literals the contracts will always be positive, negative or unassigned. If a certain literal does not occur in the overall configuration, its concrete value assignment in the CNF is irrelevant for the specific value assignments that have been chosen for the remaining literals.

The correctness of this mapping should be immediately apparent since the overall structure of the search space corresponds directly to the conjunctive normal form. Resolving the dependencies of the anchor corresponds to solving the individual clauses of the conjunctive normal form and vice versa. The resource requirements of the contracts that represent literals and their negation ensure that a literal can only be used in its positive or negative form.

The most complex operation performed by the algorithm is the construction of the resources and the construction of the contracts that represent the negation of a literal. Since these operations are both quadratic in the number of clauses the overall complexity of the reduction $O(c^2)$. Therefore, we can conclude that 3-SAT can be reduced to automatic configuration using a polynomial algorithm. As a result, automatic configuration must be at least NP-hard as well.

3.1.4 Requirements

In the following, we derive the general requirements on solving the configuration problem that has been presented previously. The requirements can be derived directly from the underlying smart peer group model and the overall vision of Pervasive Computing with respect to the distraction-free support of user tasks through applications that are executed on invisibly integrated computers.

Completeness

If a valid application configuration exists an approach for solving the configuration problem should be able to determine one. Also, it should be capable of detecting that a certain application is currently not executable at all. If an approach for automatic configuration is frequently not able to find a valid configuration even though it exists, the usability will suffer and users will eventually become frustrated. This clearly contradicts the vision of distraction-free support for user tasks. However, as discussed previously the problem of finding a single configuration is NP-complete. As a result, achieving completeness for large problem instances is not practicable. Thus, in practice we can only demand that automatic configuration is capable of finding solutions in a broad range of application scenarios.

Efficiency

Since the configuration problem is NP-complete, even the best (known) complete solutions exhibit exponential runtime complexity (if $P \neq NP$). Thus, any complete solution will necessarily introduce a configuration delay that increases exponentially with the size of the problem. From the perspective of the user, long configuration delays are especially problematic since the user might be waiting for the application to start. Thus, long configuration delays will conflict with the goal of providing seamless support for user tasks. As a result, efficiency becomes a major requirement on approaches towards automatic configuration in order to be able to solve problem instances of relevant sizes with reasonable configuration delays. Thus, approaches for solving the configuration problem should include optimizations that enable speed-ups whenever possible but without overloading the resources of the computers of a smart peer group and without sacrificing completeness. While such optimization cannot reduce the inherent complexity of automatic configuration, they can reduce the values of the constants.

Optimism

Ideally, approaches for automatic configuration should be fast in resource-poor as well as resource-rich scenarios. Typically there is a trade-off between optimizing worst- and best-case scenarios. For example, for some instances of the configuration problem, it might be more efficient to perform a pre-computation to reduce the search space instead of performing a search directly. However, for other problem instances such pre-computations might cause additional overhead. Since users would expect to achieve speedups by adding computers and resources, optimizations of the worst-case delays in resource-poor scenarios at the cost of higher execution times in resource-rich scenarios are not desirable. Therefore, automatic configuration should be optimistic.

Distribution

In smart peer groups, the availability of a powerful and reliable computer cannot be guaranteed. As a result, the efficiency of a centralized approach will be limited in environments that consist of a large number of resource-poor computers. In order to utilize the resources of all kinds of smart peer groups effectively, automatic configuration should be performed cooperatively by the available computers. Furthermore, to reduce the configuration delay as far as possible, approaches for automatic configuration should be able to utilize the parallelism inherent in smart peer groups that consist of multiple computers. Thus, instead of computing the solution sequentially, they should try achieve a speed up through parallel computations.

Resilience

The mobility of users and computers in pervasive systems leads to continuous and possibly unpredictable fluctuations regarding the availability of functionality and resources. As a result, applications in such systems have to cope with the resulting dynamics at runtime. Since computing a valid configuration is a process that might take multiple seconds, approaches for automatic configuration should be able to deal with the fluctuations that can be detected during the process. Furthermore, since automatic configuration is a fundamental part of the system software, the overall process itself must be resilient to failures.

3.2 Approach

As discussed previously, finding a single executable configuration in the presence of strictly limited resources is an NP-complete problem. As a result, approaches for automatic configuration can apply all NP-complete formalisms. By applying such formalisms, we can reuse existing algorithms which allows us to benefit from many optimizations that have been developed in the past. Constraint satisfaction (Russell & Norvig, 2003) is one such formalism and there exists a large body of algorithmic solutions for solving Constraint Satisfaction Problems.

As we show in the following, automatic configuration can be mapped efficiently to a Constraint Satisfaction Problem (Handte, Becker, & Rothermel, 2005). Due to the specifics of the system model, sequential approaches towards solving Constraint Satisfaction Problems cannot fulfill the requirement regarding distribution. The foundations for parallel algorithms have been developed in the field of Distributed Artificial Intelligence. In this field, the notion of Distributed Constraint Satisfaction Problems has been formalized (Yokoo, Durfee, Ishida, & Kuwabara, 1998) which has led to the development of a considerable set of parallel algorithms for solving such problems (Yokoo & Hirayama, 2000). From this set of algorithms, we show how asynchronous backtracking (Yokoo, Durfee, Ishida, & Kuwabara, 1992) can be extended to fulfill all requirements on automatic configuration.

In the following, we briefly outline the theoretical foundation of Constraint Satisfaction Problems and their algorithmic solutions. For the sake of brevity, we focus on complete solutions that are based on backtracking. Following the discussion, we show how automatic configuration can be mapped to a Constraint Satisfaction Problem. Thereafter, we show how we can integrate the mapping into the configuration process and we discuss the necessary extensions that are required to utilize asynchronous backtracking as configuration algorithm.

3.2.1 Constraint Satisfaction

Constraint Satisfaction Problems (Russell & Norvig, 2003) can be described as follows: Given a set of variables $V = \{V_1,...,V_n)$ with corresponding finite domains D_i and a set of constraints $K = \{K_1,...,K_m\}$ between a subset of the variables that describe the allowed combinations of values for that set, find a variable assignment $x = (x_1 \in D_1,...,x_n \in D_n)$ such that all constraints are met. Constraint Satisfaction Problems can be represented graphically as constraint network. In order to represent such a network graphically it might be favorable to restrict the set of constraints to unary and binary constraints. As discussed in (Rossi, Perie, & Dhar, 1990) it is possible to convert n-ary constraints to binary constraints.

A slight variation of the Constraint Satisfaction Problem is the Distributed Constraint Satisfaction Problem. Distributed Constraint Satisfaction Problems are simply Constraint Satisfaction Problems in which the set of variables are distributed across a number of agents. In contrast to Constraint Satisfaction Problems that have inspired the creation of a broad range of sequential algorithms, the notion of Distributed Constraint Satisfaction Problems has inspired the development of an ever increasing set of parallel algorithms. An important simplification that is frequently made during algorithm design, is that each agent maintains just one variable. If this problem can be solved, other mappings of agents to variables can be solved by applying the algorithm multiple times, i.e. once for all variables managed by the agent. Since many of these parallel algorithms are based on the ideas of one or more sequential algorithms, we first discuss the general algorithm design space and we present basic sequential backtracking algorithms before we introduce parallel solutions.

3.2.1.1 Backtracking Algorithm Design Space

Approaches for solving constraint satisfaction problems can follow to complementary strategies to which we will refer to as systematic search and constraint propagation in the following. Backtracking algorithms can be seen as a combination of these strategies. As discussed in (Kumar, 1992), it is possible to order the backtracking algorithms depending on the amount of constraint propagation performed by them. It is noteworthy that there is a broad spectrum of approaches in which search and constraint propagation are not performed in a systematical manner, e.g. (Selman, Levesque, & Mitchell, 1992) . These strategies usually result in incomplete algorithms, i.e. they are not able to guarantee that they find valid solutions for all problem instances. Due to the completeness requirement on automatic configuration, we restrict the following discussion to systematic approaches.

The first strategy simply performs a systematic search on the original problem. The basic approach is to generate a complete assignment which is then validated against the constraints. If no constraint conflicts with the generated assignment, the algorithm has found a solution. If the generation is done in a systematical manner, this algorithm will eventually find a solution if it exists and it is also able to detect that a certain problem instance cannot be solved in cases where all possible solutions have been tested exhaustively. Since this algorithm needs to generate all possible solutions, its complexity depends on the product of the size of all variable domains.

The second strategy tries to transform the problem into a simpler one by eliminating invalid (combinations of) assignments. In the context of this discussion, the most important concept is arc consistency. Arc consistency is a directional concept between two variables that denotes the fact that for each variable assignment of the first variable, there is a variable assignment in the second variable that does not conflict with the constraints between the variables. Two variables can be made arc consistent by deleting values from the domain of the first variable for which no valid assignment in the second variable exists. Removing such values from the domain will not change the solutions of the problems, since the values can never be selected simultaneously. There are efficient algorithms such as AC-3 (Mackworth, 1977) that can make constraint problems arc consistent. However, arc consistency alone is in general not sufficient to avoid search. Stronger notions of consistency (i.e. k consistency, a generalization of arc consistency which results in global consistency for a large enough value of k) can be defined to achieve that. However, applying them such that the problem does not require search is usually considered to be more expensive than searching.

3.2.1.2 Sequential Backtracking Algorithms

As mentioned previously, most algorithms combine the notion of constraint propagation and systematic search in order to reduce the search space. These schemes usually assign values to the variable sequentially and after each assignment, they perform some tests. Depending on the performed tests that reduce the search space, the schemes can be classified in look-back and look-ahead (Dechter & Rossi, 2000). Look-back schemes try to improve the value assignment by learning from previous assignments that failed, i.e. try to avoid the same failures to some extent. Look-forward schemes try to improve the value assignment by looking at the remaining solutions for the variables that have not been assigned already. Thus, they try to avoid future failures to some extent.

The most basic algorithm in this category is backtracking. Starting from one variable, the algorithm sequentially assigns a first valid value to one variable after another. If it detects that it is not possible to assign a valid value to the next variable, it revisits the variable that has been assigned last and modifies the assignment by selecting the next value in the domain. If the domain has no further values, it revisits the previous variable and so on. In contrast to generate-and-test, backtracking ignores some parts of the search space that cannot contain a valid solution. Thus, backtracking will always perform better than generate-and-test. However, due to the fact that backtracking changes values without considering the cause, it still performs many redundant assignments and test that fail repeatedly for the same reason. This problem is commonly referred to as thrashing.

Thrashing can be partially avoided by considering the cause of the conflict as done in intelligent backtracking schemes. Instead of blindly changing the variable that has been assigned last, these schemes backtrack directly to the variable that caused the problem. Since they “jump” over the intermediate variable assignments without trying to change them, these schemes are sometimes also referred to as back jumping (Gaschnig, 1977). However, while going back to the variable, intelligent backtracking schemes will invalidate intermediate assignments. Thus, they will have to perform the associated assignments and validations again, after they have changed the value of a variable. Thus, they might have to rediscover invalid combinations of assignments for intermediate variables.

An approach that also avoids this cause of thrashing is called dependency-directed backtracking (Bayardo & Miranker, 1994). The basic idea is to combine back-jumping with a set of so-called nogoods that record causes of invalid assignments after they have been detected. During the variable assignment, this set of nogoods is used to restrict the domain of a variable that needs to be assigned in order to avoid failures that have been made already. As a result, dependency-directed backtracking is able to prune the search space just like plain backtracking, it can directly jump to variables that may resolve a conflict and it will never make the same failure twice.

Plain backtracking, back jumping and dependency-directed backtracking are solely looking back at previously made failures. In contrast to that, schemes that look forward are performing tests on the search space that has not been assigned already. A simple scheme for doing this is forward checking. Forward checking essentially enforces arc consistency between a partial assignment and the remaining variables using the procedure described in the following. Before a new value is assigned to the next variable, forward checking will tentatively remove all values from the domains of the remaining variables that conflict with the new assignment. If one of the domains of the remaining variables becomes empty, another value is chosen and if there is no other value in the domain, the algorithm performs backtracking. Otherwise, the value is chosen, the tentative restriction of the domains is made permanent and the search continues with the next variable.

Forward checking essentially enforces arc consistency solely between a partial assignment and the remaining variables. In addition it is possible to enforce a higher degree of arc consistency by enforcing it also between (parts of) the variables that have not been assigned yet. Schemes that do this in an increasingly manner are for example partial look-ahead and full-look ahead. However, the basic problem with all advanced schemes is that they essentially trade-off the overhead of systematic search with the overhead of enforcing consistency. It has been noted that for some problems even dependency-directed backtracking can create a higher runtime overhead than plain backtracking due to the higher computational effort for recording nogoods and finding an assignment. So the question whether the higher degrees of consistency are beneficial depends on the type of problem.

It is noteworthy that the backtracking schemes presented previously do not completely specify how a solution should be generated. Specifically, they do not prescribe how to order the variables and how to order the values in a domain. The resulting degree of freedom can be leveraged by variable and value-ordering heuristics. A variable heuristic may, for example, try to rearrange the variables such that the most constraining variables are assigned first. The main problem with such variable ordering heuristics is that static orderings may not be optimal and rearranging the variables dynamically at runtime requires additional precautions in order to maintain completeness. A complete algorithm that performs such reordering while maintaining completeness is dynamic backtracking (Ginsberg, 1993). A simple value ordering heuristic could try to assign the values such that it opposes the lowest amount of constraints on the remaining unassigned variables. This value ordering heuristic is commonly referred to as min-conflict heuristic and it provides a considerable speedup for many problems.

3.2.1.3 Parallel Backtracking Algorithms

While the notion of Constraint Satisfaction Problems spawned the development of a broad number of sequential backtracking algorithms, the notion of Distributed Constraint Satisfaction Problems (Yokoo, Durfee, Ishida, & Kuwabara, 1998) has lead to the development of various parallel algorithms (Yokoo & Hirayama, 2000). Many parallel algorithms are inspired by the design rationales of one or more sequential algorithms, however, that is not to say that they are simply distributed variants. In order to maximize the parallelism, the algorithms usually require some form of non-trivial coordination mechanism that minimizes the amount of synchronization between individual agents.

Asynchronous backtracking (Yokoo, Durfee, Ishida, & Kuwabara, 1992) is one of the earlier parallel algorithms that have been developed to solve the Distributed Constraint Satisfaction Problem. It is a dependency-directed backtracking algorithm which means that it records nogoods in order to avoid repetitive failures during search. In asynchronous backtracking all agents assign values to their variables independently in parallel. Agents whose variables share one or more constraints are exchanging their variable assignments. If a conflict is detected, it is recorded as nogood and it is exchanged with potential candidates that can resolve the conflict. Using a set of rules for the previously described exchanges of variable assignments and nogoods, asynchronous backtracking ensures completeness and termination. The basic idea to ensure completeness is to perform backtracking steps resulting from nogoods sequentially and in a predefined and fixed order. The basic idea to ensure termination is to use a total ordering between variable to construct (and maintain) a directed acyclic communication graph in which assignments are always sent top down and nogoods are always sent bottom up.

More recently, there have been a number of proposals to improve the basic scheme. Aggregations (Silaghi, Sam-Haroud, & Faltings, 2000), for instance, improve upon asynchronous backtracking by exchanging aggregated nogoods that represent a certain range of exclusions instead of individual exclusions. The two potential advantages of this are a lower communication overhead due to aggregation and higher privacy of agents due to less precise information sharing. Another possible optimization is the nogood generation procedure described in (Hirayama & Yokoo, 2000) that tries to derive the most constraining nogood in cases where different nogoods can be derived. Since the performance of asynchronous backtracking depends on the number of backtrackings which is in turn dependent on the precision of nogoods, improving the generation of nogoods ultimately reduces the communication and computational overhead.

As mentioned, asynchronous backtracking is a dependency-directed algorithm that is solely based on look back techniques. However, there are also algorithms that leverage look forward techniques. Examples are the asynchronous forward-checking algorithm (Meisels & Zivan, 2007) or the distributed arc consistency algorithm described in (Hamadi, 2002). Asynchronous forward checking is essentially a parallel version of the synchronous forward-checking algorithm. Similarly, distributed arc consistency is an algorithm that enforces full arc consistency over a problem in a distributed manner using an optimal amount of messages. Furthermore, there are also complete and parallel algorithms that perform dynamic variable reordering. Examples are the asynchronous weak commitment search (Yokoo, 1995), dynamic distributed back jumping (Nguyen, Sam-Haroud, & Faltings, 2004) and distributed dynamic backtracking (Bessiere, Maestre, & Meseguer, 2001).

The previously mentioned algorithms have in common that they try to maximize the amount of parallelism in order to achieve the highest possible speedup. However, maximizing parallelism is not a cost-free undertaking. In order to reduce the amount of synchronization, all algorithms necessarily allow temporary inconsistencies between the local information of the individual agents. These temporary inconsistencies cause additional communication and computation overhead. In some scenarios, the overheads may be so drastic that a parallel algorithm has not only higher computation and communication costs but also longer execution times than its sequential pendant (Zivan & Meisels, 2003). To mitigate such problems, it is possible to combine a parallel algorithm with sequential procedures in order to reduce the degree of inconsistency as proposed in (Hamadi, 2002), for example.

3.2.2 Configuration as Constraint Satisfaction

To use existing techniques for solving Constraint Satisfaction Problems as basis for automatic configuration, the functionalities present in a smart peer group as well as structural and resource constraints must be represented as variables, domains and constraints. Our basic approach is to map dependencies to variables and contracts that can resolve them to their domains. Furthermore, we define a set of constraints to ensure that the resulting configuration adheres to the application model and we introduce one constraint per computer to ensure that its resources are not overloaded. Since the automatic configuration only needs to find a partial value assignment for the resulting constraint graph, we introduce a pseudo value into the domain of the variables. Using additional constraints for this pseudo value, we can effectively transform the search for a partial assignment into a search for complete assignment.

Before we describe this mapping formally, we sketch the idea using the example smart peer group and application shown in Figure 15. For the *Presentation Control* which has the two dependencies *Input* and *Output*, we create two variables. Since there are two possibilities to satisfy the dependency *Output* (*Remote Viewer* and *Simple Viewer*) and two possibility to satisfy the dependency *Input* (*Remote File Access* on two different computers), the domain of all variables will be {0, 1}. For the *Remote Viewer*, we also create two variables, one for each *Display* dependency. The domain of these variables will be {0} since there is only one contract, i.e. the contract of the *Image Viewer* which can be used to resolve the dependency. We now extend all domains with the pseudo value -1. This allows us to model the fact that not all dependencies need to be resolved with one of the contracts that are forming the remainder of their domains. As an example consider that if the dependency *Output* is satisfied by the *Simple Viewer* it is not necessary to resolve the dependencies of the *Remote Viewer*.

In order to ensure that all required dependencies are resolved, we prohibit the use of the pseudo value in variables of the application anchor by adding a corresponding constraint.

Intuitively, the dependencies of the application anchor always need to be resolved. Furthermore, we add constraints to denote that a dependency must be resolved, if and only if the contract that contains the dependency is used as part of the configuration. Finally, we add constraints to ensure that the resulting configuration does not overload the resources of the individual computers. Together these constraints ensure that the resulting variable assignment for all variables represents a valid application configuration.

More formally, we can describe this mapping procedure using the functions $\mu, \alpha, \rho, \delta, \pi$ introduced as part of the problem formalization in Section 3.1.2. Thereby, we assume without loss of generality that $M = \{m_1, ..., m_l\}$ is the set of computers of the configuration problem with the contracts $C := C_1 \cup .. \cup C_l$ and the application anchor c_{anchor}:

For each dependency $d_{anchor,i} \in \delta(c_{anchor})$ of the application anchor we create one variable $v_{anchor,i}$ with the domain $D_{anchor,i}$. Thereby, we initialize the domain of each variable $v_{anchor,i}$ with the union of the pseudo value -1 and the set of contracts that can be used to satisfy the dependency, i.e. $D_{anchor,i} = \{-1\} \cup \{c_j \mid c_j \in C - \{c_{anchor}\} \wedge \mu(d_{anchor,i}, \pi(c_j))\}$. Now, we perform this procedure recursively for each contract which is part of one of the domains. So for each dependency $d_{k,i} \in \delta(c_k)$ of the contract c_k, we create the variable $v_{k,i}$ with the domain $D_{k,i} = \{-1\} \cup \{c_j \mid c_j \in (C - P) \wedge \mu(d_{k,i}, \pi(c_j))\}$ where P is the set of contracts on the path from c_k to the application anchor c_{anchor}. The resulting set of variables and domains form the sets for the Constraint Satisfaction Problem. The only thing that remains to be done now is to define the set of constraints. To represent the structural constraints, we add the following constraint to each variable $v_{anchor,i}$ of the anchor:

(1) $v_{anchor,i} \neq -1$

Constraint one ensures that the dependencies of the application anchor must be resolved properly by some contract. Furthermore, for each variable v_i that has introduced the variables $v_{k,l}$ due to a match of the demand $d_j \in \delta(c_i)$ and the provision $p_k = \pi(c_k)$ of the contract c_k we create the following constraints:

(2) $v_i = c_k \Rightarrow v_{k,l} \neq -1$

(3) $v_i \neq c_k \Rightarrow v_{k,l} = -1$

These constraints ensure (2) that all dependencies are resolved if the associated contract is used in the configuration and (3) that the dependencies are not resolved if the contract is not used. Finally, we add constraints that are defined as follows. Let V_i be the subset of the variables whose value assignment $\chi(v_i)$ represents a contract on computer m_i then:

(4) $\alpha(m_i) \leq \sum_{i=1}^{|V_i|} \rho(\chi(v_i))$

The constraint ensures that (4) the available resources of every computer that participates in the configuration are not overloaded.

3.2.3 Configuration with Asynchronous Backtracking

Using the presented mapping of configuration to constraint satisfaction it is possible to use any approach for solving constraint satisfaction problems in order to solve configuration problems. Thus, it is possible to base a configuration algorithm on any backtracking algorithm that has been described previously. From the set of possible algorithms, we use asynchronous backtracking as basis for automatic configuration. While this might seem like a rather arbitrary design decision, there are three main reasons for using asynchronous backtracking instead of other algorithms.

First and foremost, asynchronous backtracking is a parallel and complete algorithm and thus, in contrast to sequential algorithms, it can fulfill the requirements on completeness and distribution. Secondly, asynchronous backtracking is a dependency-directed backtracking algorithm which ensures that it avoids repetitive failures of backtracking algorithms that perform lower amounts of constraint propagation. Thus, it can fulfill the requirements on efficiency. Thirdly, since it is solely based on look back techniques, it does not unfold the search space unnecessarily.

The third characteristic is specifically important, since automatic configuration requires only a partial assignment of variables. If higher degrees of consistency are enforced, e.g. using forward checking, the constraint propagation procedures will lead to tests on variables that may not even be part of the solution. Thus, look forward techniques will introduce considerable overheads. These overheads can increase drastically with the number of available contracts and thus, they contradict the requirement on optimism. As a result, by relying on asynchronous backtracking, we are using the highest amount of constraint propagation that is reasonable for automatic configuration.

In the following, we first introduce the relevant details of asynchronous backtracking. Thereafter, we describe the necessary extensions that are required to utilize asynchronous backtracking as basis for automatic configuration and we discuss optimizations on the rather naïve mapping presented in the previous section. Then, we present the resulting algorithm in detail and we discuss how resilience can be achieved. Finally, we present further optimizations that reduce the communication overhead.

3.2.3.1 Asynchronous Backtracking

Before we can discuss further considerations that are specific for asynchronous backtracking, we describe the original algorithm as presented in (Yokoo, Durfee, Ishida, & Kuwabara, 1998). For this description, it is important to mention that the original algorithm assumes that each agent is responsible for exactly one variable. However, as explained earlier, this assumption can be relaxed effortlessly.

A beneficial characteristic of asynchronous backtracking is that it only poses comparatively weak requirements on the communication between different agents. Specifically, it does not assume a certain timing of messages. However, it requires that messages are never lost and that multiple messages that are sent from one computer to another are received in the same order they are sent. While the assumptions on communication can be met rather easily, the assumptions on the overall constraint graph are more problematic. In order to operate properly, asynchronous backtracking requires a total priority ordering between variables and it requires the establishment of unidirectional communication links between variables that share a constraint. These communication links needs to be established from the variable with the higher priority to the variable with the lower priority.

After this structure is established, the algorithm operates as follows. The algorithm starts in parallel by letting each agent assign some value to its variable in such a way that it does not conflict with its known constraints. The agents with higher priority variables send their current assignment to linked agents with lower priority variables as soon as they change it. The lower priority agents, in turn, evaluate the constraints that they share with higher priority agents. If an agent receives an assignment it records the assignment and it chooses its own value in such a way that it does not conflict with its constraints. If the agent needs to change the assignment, it sends its new assignment to all linked agents.

If an agent detects that it cannot assign a value to its variable in such a way that does not conflict with a constraint, it creates a nogood. This nogood contains the set of assignments from higher priority agents that cause the conflict. The agent picks the assignment with the lowest priority and sends the nogood to the agent that created this assignment. If this agent receives the nogood it checks whether the nogood is still valid. This check is done by comparing the value assignments of variables that it has recorded already with the variable assignments contained in the nogood. The nogood is still valid, if all the recorded variable assignments and the variable assignments contained in the nogood match. If the nogood is not valid, the agent can simply ignore it, since the nogood may be a result of a temporary inconsistency between the local views of the agent. If it is still valid, the agent establishes new links from all agents to it that participate in the constraint and that have not been linked so far. Then it records the nogood as new constraint and it tries to assign a new value to its variable that adheres to all locally known constraints.

The newly created links are needed to ensure that the agent can keep track of changes to assignments of conflicting variables. Thus, they basically ensure the validity of the invariant that all agents that share a constraint are linked to each other. Note that since the agent that detects a new constraint will send this constraint to the agent with the lowest priority, new links will always be created from higher priority agents to lower priority agents. This, together with the fact that the initial links also pointed from higher priority agents to lower priority agents, ensures that the constraint network will always be a directed acyclic graph.

The algorithm terminates unsuccessfully when some agent cannot assign a valid value to its variable and there are no higher priority agents that could change their assignments in order to resolve the conflict. If a valid assignment has been found, the algorithm will eventually stop creating new messages. Since this is a stable global predicate, it can be detected using a distributed snapshot algorithm.

3.2.3.2 Mapping

As mentioned previously, it is possible to utilize any algorithm for solving a constraint satisfaction problem using the mapping procedure described in Section 3.2.2. To apply asynchronous backtracking, we would have to create the initial constraint graph using the mapping procedure and we would have to establish an arbitrary total ordering between the variables. Using this total ordering, we would have to construct the directed communication links accordingly.

Yet, this simple approach has the following two drawbacks. First, the previously introduced mapping would cause considerable communication overhead that can be avoided easily and secondly, it would require a considerable amount of time to completely unfold the search space. To avoid the first problem, we can optimize the problem mapping by considering the underlying meaning of constraints. To avoid the second problem, we can unfold the search space during configuration. Both optimizations are essentially a result of the specific problem structure of the configuration problem.

To motivate the first problem consider that asynchronous backtracking creates links between variables that share a constraint. These links are then used to exchange value assignments and nogoods. Furthermore, consider that the problem mapping constructs *n* variables for each contract that has *n* dependencies. In order to ensure that the dependencies of the contract are resolved if and only if the contract is used in the configuration, the mapping algorithm established constraints between each variable of a dependency and the variable that "uses" the contract. This causes the creation of *n* communication links. However, if the "using" variable is the variable with the higher priority, the value communicated over these links will always be the same. As indicated on the left side of Figure 19, this is especially problematic in cases where the links are pointing to variables on a different computer since the resulting communication will be remote.

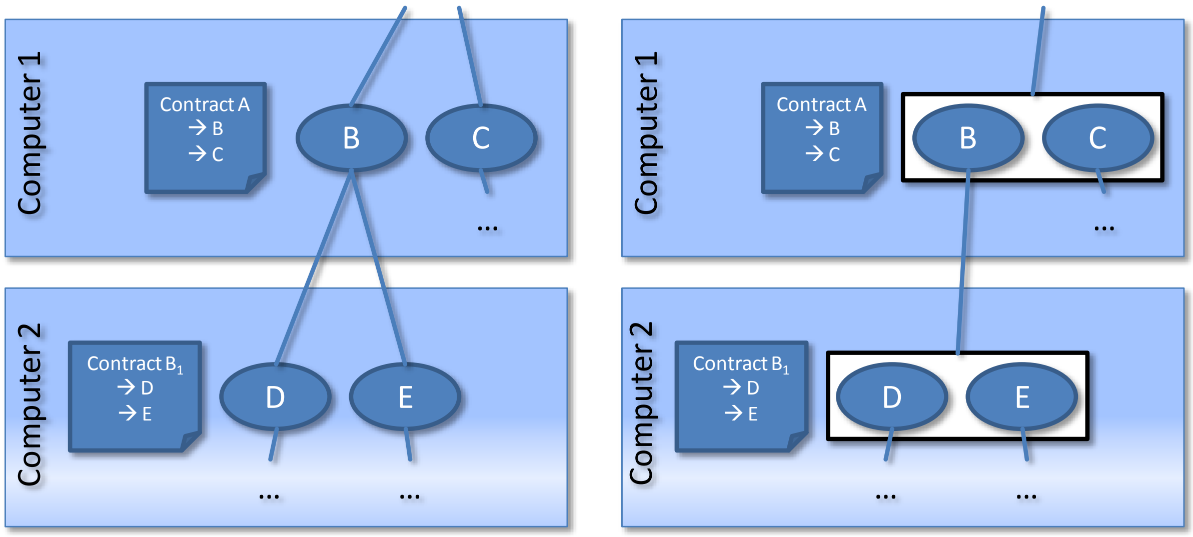


Figure 19 – Avoiding Unnecessary Communication Links

We can avoid such cases by combining the variables that represent the same contract as shown on the right side of Figure 19. In addition to reducing the number of initial communication links, the aggregation of variables also has a potential to reduce the number of communication links created during the configuration. Furthermore, the resulting multi-dimensional variable will also reduce the space overhead, since the local knowledge of individual variables can be shared. Finally, we can use the multi-dimensional variable to reduce the number of validations that need to be performed to enforce the resource constraints, since the resource constraints can be validated once for the multi-dimensional variable.

To motivate the second problem consider that automatic configuration requires a partial solution for the overall constraint satisfaction problem. As a result, completely unfolding the search space may be unnecessary for many problem instances. This is specifically problematic and costly if some potential structural options can never be used due to resource constraints or if the problem instance is not hard to solve, e.g. if it can be solved without backtracking. Thus, a complete unfolding of the search space would conflict with the requirement on optimism.

In order to avoid the complete unfolding, we can virtually assign the pseudo value to all variables that have not been constructed. Then, we can gradually unfold the search space by constructing new variables only if they are needed as part of the solution. This on-demand construction of variables is correct since the underlying constraints enforce that all variables whose contract is not used to satisfy a certain dependency should be set to the pseudo value. However, constructing the variables on demand necessarily restricts the ordering of variables to one which is compatible with the creation sequence.

As a result, the on-demand creation of variables introduces a partial order over the variables in which a variable of a contract that is used to resolve a dependency necessarily has a lower priority than the variable that represents this dependency. Any total order that we introduce for variables must not conflict with this partial order. The remaining degree of freedom can be utilized to define an arbitrary order over the unordered elements. However, in order to achieve a high degree of parallelism during the configuration, it is desirable to avoid synchronization by constructing the ordering in such a way that new elements can be introduced locally. To do this, we construct the ordering on the basis of the path of the variable in the tree that represents the configuration. That way, we can assign a globally unique id to each variable which can be used to perform comparisons.

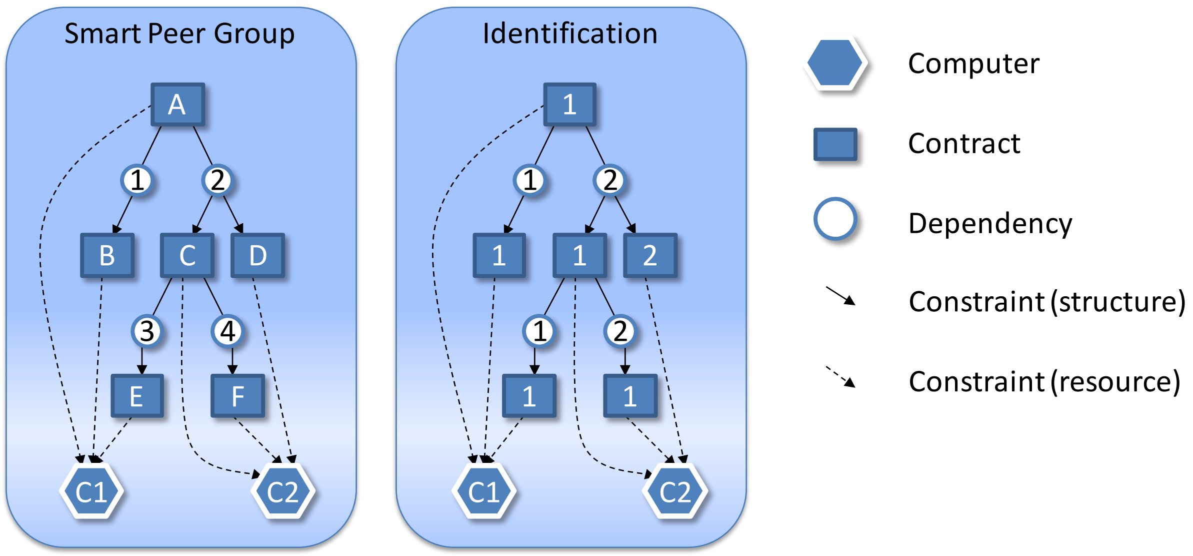


Figure 20 – Mapping Optimizations

To demonstrate the extensions and optimizations described above, consider the search space shown on the left hand side of Figure 20. The search space consists of the computers *C1* and *C2* which host components with the contracts *A* to *F* and the dependencies *1* to *4*. The contracts *A*, *B* and *E* reside on computer *C1* and the contracts *C*, *D* and *F* reside on computer *C2*. The contract *B* can be used to resolve the dependency 1. The contract *C* and *D* can be used to resolve the dependency 2. If contract *C* is used to resolve dependency *2*, it introduces two additional dependencies (*3* and *4*), which can be resolved using the contracts *E* and *F*.

Since we combine the variables of contracts into one multi-dimensional variable, we introduce only one directed (remote) communication link from the variable of dependency *2* to the variables introduced by contract *C*. If dependency *2* is resolved using contract *C*, we can perform the associated resource validation once and we can simultaneously enforce that both dependencies *3* and *4* are resolved. The same can be done for all other variables in any other problem instance. As discussed previously, this optimization is possible due to the specific structure of the configuration problem.

Structurally valid solutions for the problem instance shown above consist of the contracts *A*, *B*, *C*, *E*, *F* or *A*, *B*, *C*, *D*. If the both solutions would also be valid with respect to the resource constraints, it would not be necessary to unfold the complete search space for solving the problem instance. If the contract *B* and *C* are used to resolve the dependencies of the application anchor, there is no need to unfold the variables that might have been introduced by contract *D*. Similarly, if the contracts *B* and *D* would have been used to resolve them, there is no need to unfold the dependencies introduced by contract *C*.

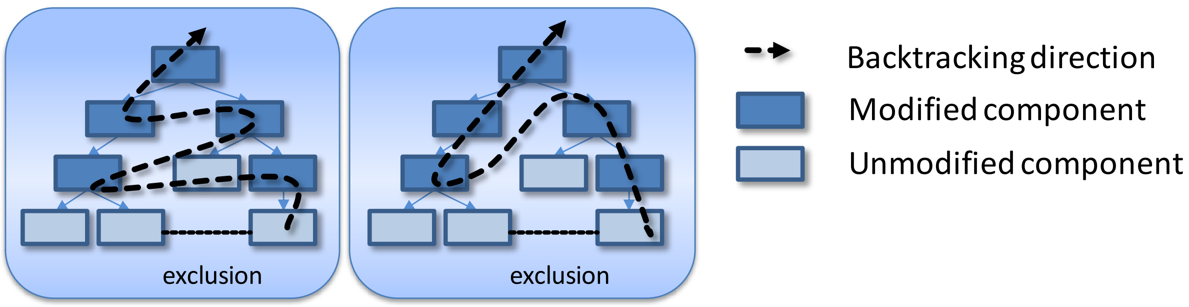


Figure 21 – Backtracking Strategies

In order to establish a total order between variables, we assign identifiers to each variable and each value of the domain of the variable. In order to avoid synchronization for the assignment of the identifiers, they must be constructed locally. Using the local identifiers, we can create globally unique identifiers by concatenating them along the path to the variable. Note that the concatenation requires that each variable knows its place within the tree. Thus, upon the first usage of a contract, each contract needs to be supplied with the identifier assigned by its parent. The parent can easily create this identifier locally by concatenating its own local identifier with a unique identifier for the dependency and the value of that dimension under which the specific contract is selected. In the example shown in Figure 20, we can use (*1, 2, 1, 1, 1*) to identify the variables of contract *E* and (*1, 1, 1*) to identify the variables of contract *B* and so on.

These globally unique identifiers can now be used to perform comparisons. In order to adhere to the partial ordering introduced due to the on-demand creation of variables, we must ensure that all identifiers that are a prefix of another identifier have higher priority, e.g. (*1, 2, 1, 1, 1* < *1, 2, 1*). Apart from that we can define an arbitrary order using the length of the identifiers and their individual values. It is noteworthy that the chosen order defines the backtracking strategy in asynchronous backtracking, since variables with higher priority are changed later. Thus, if we decide to make the length of the identifier the most significant operation during comparison, we end up with the backtracking strategy shown on the left side of Figure 21. If we decide to make the values of the identifiers the most significant operation during comparison, we end up with the backtracking strategy shown on the right side of Figure 21. For the evaluation, we use the strategy that reconfigures contracts on lower levels first, i.e. precedence on the length. The rationale for this decision is that switching between contracts at lower levels reduces the communication overhead due to lower number of recursively required contracts. However, this is a heuristic and there are cases where this leads to higher overhead.

3.2.3.3 Algorithm

In the following, we provide an overview of the resulting algorithm that utilizes the optimized mapping introduced in the previous section. For the sake of clarity, we only present a simplified version that configures only one application at a time. It should be clear that support for multiple applications can be added easily by adding a unique application identifier to the identifier of variables. In addition, we also postpone the discussion of further optimizations, the extensions to achieve resilience as well as the protocol used to detect the termination. Since these extensions and optimizations are crosscutting concerns, we present them in detail in subsequent sections.

To describe the algorithm, we follow the overall structure of the description of asynchronous backtracking given in (Yokoo, Durfee, Ishida, & Kuwabara, 1998). Thus, we model the algorithm as reactive process that responds to incoming message in its *Receive_** procedures. The main difference between the algorithm shown below and the original asynchronous backtracking algorithm is the algorithms capability to deal with multiple variables and the mechanisms to dynamically create the variables on demand. Furthermore, we introduce a resource reservation and validation procedure to avoid the explicit representation of resource constraints.

As in the original version of asynchronous backtracking, the algorithm uses three types of message namely update, backtrack and link. Update messages send the value assignment from a high priority variable to a linked low priority variable. The initial links reflect the dependencies between different contracts, but during its execution, asynchronous backtracking may generate additional links in response to backtracking messages. Backtracking messages report a nogood, i.e. an assignment to a set of high priority variables that prohibit the assignment of a valid value in some low priority variable. Thus, backtracking message are always sent into the opposite direction of update messages, i.e. from low to high priority variable. Finally, link messages inform some variable that it needs to create an additional link to some previously unlinked variable. In contrast to update and backtracking messages, which are sent asynchronously, link messages are exchanged synchronously to ensure that the links are created whenever a backtracking message is processed. Since the link establishment is trivial, we omit the discussion of the *Receive_Link* procedure for link messages.

In order to support multiple variables on the same computer, the algorithm maintains a so-called configuration object for each contract that has been used during the configuration process. The configuration object represents the local knowledge about the contract. The local knowledge consists of the multi-dimensional variable that represents the dependencies of the contract, a set of constraints that has been derived during the execution of the algorithm and a so-called view which contains the assignments of variables with higher priority that are linked. Furthermore, it contains the domain for each dimension of the multi-dimensional variable which is given by the set of contracts that can be used to resolve the corresponding dependency. In addition, the configuration object also stores a flag that denotes whether the resources for the contract have been reserved successfully. Finally, the object also needs to store information about the links that have been established already in order to avoid their duplicate creation.

```
Receive_Update(Identifier, Contract, Value)
  Configuration_Object = Get_Configuration_Object(Identifier)
  If (Configuration_Object == Null)
    Configuration_Object=Create_Configuration_Object(Identifier, ontract)
  Add_Assignment(Configuration_Object, Value)
  Validate_Constraints(Configuration_Object)
```

Algorithm 1 – Receive Update Procedure

The previously introduced information held by the configuration object is sufficient to discuss the overall algorithm. However, it should be noted that in a real implementation the configuration object also has to store information about the computers that host the contract and factories that provide them. Furthermore, the update messages that are used to initialize a variable must contain additional information about the new contract and its factory as well as about the set of contracts that have caused its creation in order to prevent cycles in the configuration of an application. Similarly, the link and backtracking messages must contain the computer identifier of all variables contained in message since the targeted variable may not know the mapping between variables and computers. However, in order to minimize the space consumption, it is possible to store the mapping information once per computer.

Since each computer can host multiple contracts, the algorithm must be capable of uniquely identifying them. To globally identify a contract and its position within the application configuration, we use the generated identifiers discussed in Section 3.2.3.2. Thus, we represent the application anchor with the empty identifier {}, the first contract that can be used to resolve the first dependency of the anchor is identified by {(0) [0]}. The second instance for this dependency is identified by {(0) [1]} and so on. Thus, identifiers are arbitrarily long sequences of pairs, where the first index of a pair denotes the dependency and the second index denotes the contract used to satisfy this dependency. As the algorithm allows the dynamic creation of new variables, we can simply start the algorithm by sending an initial update message for the application anchor. This message will then eventually create all recursively required variables.

When the algorithm receives an update message (Algorithm 1), it must first retrieve the corresponding configuration object (Line 2). If the configuration object does not exist (Line 3), it needs to be created. To create the configuration object, the algorithm needs to initialize a multi-dimensional variable for the corresponding contract and it needs to query the computers for contracts that can be used to resolve each individual dependency. These contracts will then from the domain for the dimensions (Line 4). Thereafter, it performs basically the same steps as in the original version of asynchronous backtracking, i.e. it adds the received value to the local knowledge of the configuration object and it validates the constraints. If the constraints are not met, the algorithm will either change the assignment of the variable held by the configuration object and it will inform the linked variables about this change or it will generate a backtracking message.

```
 1: Validate_Constraints(Configuration_Object)
 2:   If (! Is_Valid(Configuration_Object))
 3:     If (Can_Create_Assignment(Configuration_Object))
 4:      If(Is_Used(Configuration_Object)&!Is_Reserved(Configuration_Object))
 5:         If (Can_Reserve(Configuration_Object))
 6:           Reserve(Configuration_Object)
 7:           Assignment = Create_Assignment(Configuration_Object)
 8:           Set_Assignment(Configuration_Object, Assignment)
 9:         Else
10:           Trigger_Backtracking(Configuration_Object)
11:           Return
12:      If(!Is_Used(Configuration_Object)&Is_Reserved(Configuration_Object))
13:         Free(Configuration_Object)
14:         Assignment = Create_Assignment(Configuration_Object)
15:         Set_Assignment(Configuration_Object, Assignment)
16:    Else
17:       Trigger_Backtracking(Configuration_Object)
18:       Return
19:   Send_Update(Configuration_Object) // for all dimensions to all links
```

Algorithm 2 – Validate Constraints Procedure

To validate the constraints of a configuration object (Algorithm 2), the algorithm needs to compare the current assignment and the local view on the problem, i.e. the assignments of linked variables, with the known constraints. These constraints can either be initial constraints, i.e. resource and structural constraints, or derived constraints that have been created from nogoods sent by other variables. In order to ensure that any configuration does not consume more than the available resources, the algorithm performs resource reservations. Thus, the validity check (Line 1) returns true if the current assignment does not conflict with any initial or derived constraint and if the resources have been reserved or freed accordingly. If the check succeeds, the algorithm simply reinforces the current value assignment (Line 19). If the check fails, the algorithm checks whether it is possible to assign a consistent value to the variable (Line 3). If that is not possible, it performs backtracking (Line 16-18). If it is possible, it tries to free or reserve the resources according to the knowledge stored in the configuration object (Line 4-15), i.e. if the contract is used, the resources need to be reserved, if the contract is not used, the resources need to be freed. During this procedure there can only be conflicts in cases where the contract is used and the resources cannot be reserved. If this happens, the algorithm needs to perform backtracking (Line 10-11). Otherwise, the algorithm can simply assign the value to variable and reinforce the assignment (Line 6-8 or Line 13-15 and Line 19).

```
1:  Trigger_Backtracking(Configuration_Object)
2:   If (Is_Anchor(Configuration_Object))
3:     // Unsuccessful termination
4:   Else
5:     Conflict_Set = Create_Minimum_Conflict_Set(Configuration_Object)
6:     Minimum_Assignment = Get_Minimum_Assignment(Conflict_Set)
7:     Remove_Assignment(Configuration_Object, Minimum_Assignment)
8:     If (!Is_Used(Configuration_Object)&Is_Reserved(Configuration_Object))
9:       Free(Configuration_Object)
10:    Send_Backtracking(Minimum_Assignment, Conflict_Set)
```

Algorithm 3 – Trigger Backtracking Procedure

Asynchronous backtracking terminates unsuccessfully, when a certain conflict can never be resolved. Due to the specific problem structure of automatic configuration, this case occurs if and only if the application anchor triggers a backtracking step. The rationale for this is twofold. First, only those contracts that are used within a configuration can cause conflicts. Second, all contracts except for the contract of the application anchor can request that they are not used by issuing a corresponding nogood. Thus, before a backtracking step is about to be triggered, the algorithm should check whether the backtracking is issued by the configuration object of the anchor (Line 2). If it is the anchor, the algorithm terminates unsuccessfully (Line 3). If it is not the anchor, the algorithm needs to compute the set of variables assignments that cause the conflict (Line 5). The simplest way to compute the conflict set is to use all assignments of variables that are linked to the configuration object since the combination of these variables make a valid assignment impossible. However, in order to avoid repetitive failures, the conflict set should be minimal. Computing this set is a non-trivial task. However, if the conflict was not caused by a lack of resources, the set can be reduced by using only those variables that are part of one or more constraints that restrict the domain. Similarly, if the conflict is caused by a lack of resources, the set can be reduced by using only those assignments that also require the limited resource. If the set has been computed, the algorithm can remove the assignment of the lowest variable contained in the conflict set (Line 7). Thereafter, it may release the resource reservation in cases where the lowest variable was the variable of the parent (Line 8-9) and it can send the backtracking message to the lowest variable (Line 10).

```
1: Receive_Backtracking(Identifier, Conflict_Set)
2:   Configuration_Object = Get_Configuration_Object(Identifier)
3:   Create_Links(Configuration_Object, Conflict_Set)
4:   If (! Is_Outdated(Configuration_Object, Conflict_Set)
5:     Add_Constraint(Configuration_Object, Conflict_Set)
6:     Validate_Constraints(Configuration_Object)
7:   Else
8:     Send_Update(Configuration_Object) // for all dimensions to all links
```

Algorithm 4 – Receive Backtracking Procedure

The steps that need to be taken whenever a backtracking message is received are almost identical to asynchronous backtracking (Algorithm 4). The first thing that needs to be done is to retrieve the configuration object that is addressed by the conflict set (Line 2). Thereafter, the algorithm generates the links to all variables of the conflict set that have not been linked so far (Line 3). Note that the creation of these links is done synchronously. After the links have been established, the algorithm determines whether the conflict is still valid (Line 4). According to the asynchronous backtracking algorithm, a conflict is still valid, if the variable assignments contained in the conflict set do not conflict with the local knowledge contained in the configuration object. If the conflict is still valid, the algorithm records the conflict set as a new constraint (Line 5) and it revalidates the assignments of the configuration object (Line 6). If the conflict is outdated, the algorithm simply reinforces the current value assignment.

3.2.3.4 Correctness

Since this algorithm does not modify the logic of asynchronous backtracking, the proof of correctness follows the argumentation provided in (Yokoo, Durfee, Ishida, & Kuwabara, 1998). Since all configuration objects create either update or backtracking messages in response to changes, algorithm cannot reach a stable state as long as some variables have an invalid assignment. Even in cases where a backtracking message is ignored due to outdated information in the conflict set, the algorithm ensures that all variables will eventually have the same consistent local knowledge by reinforcing the variable assignment across all links.

Due to the total ordering of variables and the construction of the directed links from high to low priority variables, the initial constraint graph cannot contain cycles. Furthermore, since the backtracking messages are always sent to the lowest priority variable contained in the conflict set, further links are always created from high to low priority variables. Thus, the constraint graph will always remain acyclic. This ensures that the continuous reinforcements will stop eventually if all variables have assigned a valid value. If the constraint satisfaction problem is over-constrained, i.e. if it does not have a solution, the algorithm will eventually create an empty conflict set. Due to the specific structure such an empty conflict set can only be generated at the application anchor. The reason for this is that all other contracts may always as their parent to change the dependency.

3.2.3.5 Optimizations

The mapping described in Section 3.2.3.2 already reduces the number of messages required for configuration significantly when compared to the basic mapping described in Section 3.2.2. However, the on-demand creation of variables only considers that variables that have not been used do not require updates. It is possible to apply the same idea also to variables that have been created but that are currently no longer used.

To motivate this consider the constraints (2) and (3) introduced in Section 3.2.2. These constraints essentially state that a contract that is used to resolve a dependency must be resolved properly and a contract that is not used to resolve a dependency must not be resolved. However, when asynchronous backtracking decides to select a different contract to resolve a variable it will send an update to all contracts that have been used previously to resolve the variable. This will cause unnecessary overhead for variables whose domain contains at least three different contracts. The reason for this is that at least one of the three contracts knew already that it was not used and sending the update to this contract does not lead to changes.

As a result, it is possible to restrict the update messages to the contracts that actually need to be informed about a change and in cases where a variable does not change its value assignment at all, we can avoid sending any updates. As a result, we can compute the set of links that require a certain update as the set of links that are linked to changed variable minus the set of links that point to variables that are not affected by the change because they are not used within the configuration. These links do not have to be updated since a change to the variable assignment will never lead to changes.

Yet, if the update messages are restricted to links that truly require a variable assignment, additional care needs to be taken during backtracking. The reason for this is that a backtracking message that is silently dropped due to a temporary inconsistency will require an unconditional reinforcement of the variable assignment to resolve the inconsistency. While this is not a problem for a single backtracking message it will become a problem in cases where the reception of a backtracking message leads to another backtracking message. To understand this consider that the procedure that creates the conflict set automatically removes the lowest variable assignment from the local knowledge from the configuration object that detected the conflict. If this is happens multiple times due to a chain of backtracking messages which is eventually dropped due to an inconsistency, the reverse path of the backtracking messages needs to be updated to restore the removed values.

To ensure that this is done correctly, it is possible to record the variable identifiers of the senders of a backtracking message in cases where a backtracking message is accepted. Due to the links created by asynchronous backtracking, the variable of the sender of the backtracking message will always be linked to the variable that receives the backtracking message already. Thus, we can simply mark the link as a link that needs to be updated unconditionally upon the next reception of an update message. Using these modifications, it is possible to drastically reduce the number of update messages since they will be sent only to those variables that actually require updates.

3.2.3.6 Resilience

As stated in Section 3.1.4, an algorithm that performs automatic configuration must be capable of dealing with fluctuations that occur during its execution. Such fluctuations might be the result of the unavailability of local resources or the mobility of computers. In general, these fluctuations are typically hard to predict and in pervasive systems, they may be frequent. As examples consider a user that removes some USB camera from its computer by simply unplugging it or a moving user who carries a set of mobile computers.

Failure Model

Besides from being hard to predict, fluctuations resulting from mobility and failures are also hard to detect since pervasive systems essentially resemble asynchronous systems. This can be easily explained by the fact that the underlying networking technologies often do not guarantee an upper bound on the message delivery delay. As a consequence, it is not possible to reliably detect the unavailability of a computer by exchanging messages. In practice, we can therefore only detect such failures heuristically, for example, by securing message delivery with acknowledgements that must arrive within a reasonably large timeframe. If an acknowledgement does not arrive within the timeframe we must assume that the message has not been delivered due to a computer failure or a network partitioning. Of course, it is possible and desirable to apply more complex protocols, for example, by dynamically estimating the message round-trip time to deal with network congestion and by introducing retransmissions to overcome unreliable communication channels.

However, independent from the concrete implementation such protocols can only determine the unavailability of a computer heuristically since there may be cases where the network or the receiver are simply too slow. Other failures such as messages that are modified during transmission can usually be prevented by means of checksums or forward error correction. Furthermore, since we assume that the computers in a pervasive system are cooperative, we can safely argue that they will not modify messages on purpose. As a consequence, we can assume that all failures that may occur are detectable by means of missing acknowledgements. However, since asynchronous backtracking does not continuously communicate with all computers, detecting failures requires some form of periodic beaconing to ensure that a computer that has not received a message for some time is still available.

Our prototype implementation described in Chapter 5 introduces such a mechanism by means of leases. This lease mechanism ensures that all computers are notified in cases where a previously available computer becomes unavailable due to mobility or failures. Note that in the case of a network partitioning the lease mechanism also ensures that the computers which are no longer able to contact the coordinating computer are able to remove their orphaned state.

In contrast to changes in computer availability, the unavailability of a previously available resource can be detected easily due to the fact that each resource is managed locally by one computer. Given that the computer is able to detect changes, a reduction in resource availability can be handled directly locally or it can be propagated if necessary. Thereby, it is noteworthy that reductions in resource availability only need to be handled in cases where the change conflicts with a reservation. In other cases, it can be ignored silently since the resource is currently not needed. If it is needed at some later point in time, the resource reservation procedure ensures that the resulting conflict is handled properly.

Failure Handling

The simplest way to deal with fluctuations is to restart the complete configuration algorithm. Yet, this approach may obviously introduce a high overhead since all constraints that have been discovered so far need to be rediscovered again. Fortunately, asynchronous backtracking does not impose timing constraints on the reception of messages. This allows us to implement a much more efficient failure handling for both types of fluctuations in a relatively straight-forward manner.

- **Local resources**: In order to deal with the unavailability of local resources, we can simply compute the conflict sets of resource reservations for the resource or the resources that are no longer available in a sufficient quantity. Using the conflict sets, we can create corresponding backtracking messages that we inject from the configuration object with the lowest priority contained in the conflict set. The generation of these backtracking messages will not endanger the correctness of the overall algorithm since the same backtracking message would have been generated by the algorithm in cases where the resource reservation would have failed at an earlier point in time.
- **Remote computers**: To deal with the unavailability of computers we apply a less lightweight procedure. Whenever the unavailability of some computer is detected, we

stop the execution of the algorithm. Thereafter, we iterate over all configuration objects and we add constraints that prohibit variable assignments in the domain of the variable of the configuration objects that would select contracts on the unavailable computer. Thereafter, we deselect all configuration objects and we free the associated resource reservations. In addition, we remove all links to configuration objects on the unavailable computer. Finally, we restart the algorithm by selecting the configuration object that represents the application core.

Note that our extensions for resilience are solely targeted at fluctuations that result in the unavailability of local resources and remote computers. Thus, these extensions essentially reduce the search space monotonically. Extensions that would allow the usage of newly available computers and resources would be more complicated. The main reason for this is that newly available resources and computers might lead to new options in the search space. As a result, they could invalidate some of the derived constraints that have been discovered and propagated already. In order to eliminate the wrong constraints, we would have to analyze the effect of the additions using a distributed algorithm.

Correctness

As explained previously, the correctness of the failure handling procedure for local resources is obvious since the same message sequence could have been produced by the normal algorithm execution if the unavailable resource would not have been available in the first place and the backtracking message would have been deferred. As a consequence, this procedure must be correct, given that the original algorithm was correct.

The correctness of the failure handling procedure for remote computers is more complicated. The first step of stopping the algorithm ensures that the algorithm execution does not interfere with the failure handling procedure. The second step, namely the introduction of additional constraints ensures that a future execution of the algorithm will not attempt to form a configuration that contains configuration objects on the unavailable computer. The last step resets the state that has been created by the algorithm to enable a consistent restart. However, instead of removing all state the procedure only deselects all configuration objects, frees all resources and removes unnecessary broken links.

The resulting state is mostly corresponds to the initial state of the algorithm (no configuration objects selected and no resources selected) with the exception that there are more propagated constraints and additional links that have been created in response to the constraint propagation. In cases where the propagated constraints do not contain configuration objects on the unavailable computer, the unavailability of the computer does not affect them. The same holds true for the links that have been created by the propagation of these constraints. In other cases, a newly introduced constraint that disallows the selection of the configuration object on the unavailable computer supersedes them. Thus, the removal of the broken link does not affect the correctness since the associated configuration object cannot be used anymore and thus, there is also no need to inform it about changes to ensure completeness.

As a consequence, the remaining conflicts and links represent a consistent intermediate set of conflicts and links that could have been produced by the normal algorithm execution if the configuration objects on the unavailable computer could have never been selected in the first place. However, since the current value assignment of variables may be conflicting with the newly introduced constraints, it is necessary to rerun all of them. By deselecting all configuration objects and freeing all resources before restarting the algorithm, we effectively ensure this. The consecutive executions of the constraint validation procedures will then also consider the newly introduced constraints. This in turn ensures that the unavailable computer is effectively excluded from the configuration and thus, the resulting configuration will be valid.

3.2.3.7 Termination

Asynchronous backtracking terminates unsuccessfully if an empty constraint set is generated during the execution, i.e. if there is no further choice that can be reconsidered in order to resolve an unsatisfied constraint. Due to the specific structure of the configuration problem, such an empty set can only be generated by the configuration object of the application anchor. All other configuration objects can always ask their parents to reconfigure themselves in such a way that they are no longer used. Thus, an unsuccessful run will be recognized by the computer that hosts the configuration object of the application anchor.

The successful termination of the algorithm is achieved if all participating computers stopped generating new messages and all messages have been processed. Therefore, the successful termination represents a stable predicate on the state of the distributed system that can be detected using distributed snapshots (Chandy & Lamport, 1985). Due to the practical importance of this problem class for distributed computations, this class is widely known as distributed termination (Francez, 1980) and there are various algorithms to solve it, e.g. (Dijkstra & Scholten, 1980), (Tel & Mattern, 1993). The taxonomy presented in (Matocha & Camp, 1998) alone compares 35 different algorithms. It classifies the algorithms based on the type of algorithm, e.g. wave-based, credit-based, etc., on the assumptions on the network structure, e.g. tree-based, cycle-based, etc., and on the communication channel, e.g. FIFO, etc., and on the symmetry of the algorithm, among others.

From a conceptual point of view, we could use any of these algorithms to detect the successful termination of the configuration algorithm. Yet, in order to satisfy the requirement on resilience, the termination detection algorithm itself must be resilient. Since the failures that may occur during automatic configuration may be a result of mobility, we must assume that they can be permanent only. Thus, termination detection algorithms that support the recovery from transient failures such as (Tseng & Tan, 2001), for example, cannot be used. Instead, the termination detection algorithm for automatic configuration needs to be able to deal with permanent failures. An example for such a termination detection algorithm is presented in (Lai & Wu, 1995). Given a set of *n* processes whose termination should be detected, the algorithm is capable of dealing with *n-1* faulty processes. To do this, the algorithm extends the wave-based mechanism presented in (Dijkstra & Scholten, 1980). However, since some of the extensions are not necessary for automatic configuration, we decided not to rely on this algorithm. An example for such an unnecessary extension is the capability of dynamically switching the coordinator in cases where it fails. This extension is not required for automatic configuration, since if this computer would fail, the whole configuration process could be aborted as the computer that hosts the application anchor would no longer be available.

To come up with a more specialized and simpler solution, we extend the credit-based termination detection algorithm presented in (Mattern, 1989). The basic idea behind this algorithm is the distribution and recovery of so-called credits whose construction maintains the invariant that the sum of all credits corresponds to a fixed known value, e.g. 1. When the basic computation, i.e. the configuration algorithm, starts, the coordinator, i.e. the computer that hosts the application anchor, will be equipped with a credit that corresponds to 1. Every time the basic computation transmits messages to other computers, the algorithm attaches a fragment of the credit to the message. Every time a message is received, the computer detaches the credit form the message and stores it. The stored credit fragments can then be used to equip further messages with credits. As soon as a computer detects the local termination, it returns the credits to the coordinator. If the credits contained at the application anchor add up to the known fixed value, i.e. 1, the termination algorithm can safely assume that all participating computers have terminated locally and that there are no further messages.

Since the participating computers may become unavailable during the configuration of an application, the algorithm described previously may lose the credits, e.g., that are stored on the computer that is no longer available. To cope with lost credits, we can try to determine the amount of missing credits. Doing that would cause considerable effort since we need to ensure that we have collected all remaining credits including the ones that may still be in transit. Due to the fact that we also need to restart the configuration algorithm in response to an unavailable computer, we can simplify this significantly by synchronizing the participating computers and restarting the termination detection protocol as well. To minimize the overhead of restarting the configuration, we use the following strategy.

In addition to credits, each computer and each message is equipped with an epoch value. If the epoch value is started at 0, the epoch essentially denotes the number of computers that have failed during the configuration. As soon as a computer detects that another computer is no longer available, it signals this to the coordinator, i.e. the computer that configures the application anchor. This may result in two outcomes. If the computer that detects the failure has left the smart peer group, it can no longer communicate with the coordinator. Thus, it can simply remove all state associated with the configuration algorithm. If the computer that detects the failure is still part of the smart peer group, it will be able to contact the coordinator and thus, it simply waits. In response to the signal, the coordinator will send a failure notification to all computers that are still participating in the configuration. The failure notification contains the identifier of the computer that has failed, i.e. the one that has left the smart peer group. In response to that, the computers stop their computation, increase their epoch value and remove all messages and all credits. Since there may be additional messages in transit the computers will simply ignore all further messages that have a lower epoch value. Finally, each computer needs to perform the corrective actions described in the previous section and it needs to signal the completion of these actions to the coordinator. After the actions have been performed on all remaining computers, the coordinator resets its credit to the original value, e.g. 1, and starts the configuration again.

Although, the previously described approach suffices to deal with failures of computers, the utilization of a credit-based termination detection algorithm complicates the resilience to fluctuating resource availability. To clarify this, consider that the fluctuating resource availability may cause the detection of a new conflict at any point in time. Yet, in the formalization of the termination detection problem a process may only become active in response to a message. Moreover, the correctness of the credit-based termination detection algorithm depends upon this restriction. If a computer becomes active spontaneously because a resource has become unavailable, the algorithm may detect the global termination incorrectly. However, incorrect termination detection may only occur in cases where the local termination has already been signaled to the coordinator. Thus, reacting to a resource fluctuation is unproblematic as long as the affected computer is still in possession of some credits. In cases where the computer has already sent its credits to the coordinator additional precautions need to be taken. To deal with such cases, we introduce a credit request message that retrieves a credit from coordinator.

Clearly, since resource fluctuations and connectivity fluctuations may occur at any point in time, no termination detection algorithm can guarantee that the detection of a successful termination also results in a successful configuration of an application. As counter examples consider a disconnection or a resource fluctuation that is detected at the same instant of time at which the termination detection algorithm detects the global termination. However, the previously described approach reduces the gap during which a fluctuation can have this negative impact to an absolute minimum and thus, is reduces the probability of such cases as far as possible. If the computed configuration cannot be executed, it would be possible to perform the previously described corrective actions again. Yet, due to the low likelihood of this situation, we decided not to implement this. Instead, in our implementation, we restart the algorithm from scratch.

A good and desirable characteristic of the previously introduced termination detection algorithm is its low termination detection delay. If a computer returns its credits immediately after it has detected the local termination, the detection delay is optimal. In the worst case, it corresponds to the latency introduced by the network for transmitting a single credit recovery message. However, since a computer may become active again due to the reception of another message from the basic computation, there is a trade-off between optimizing the detection latency and reducing the number of credit recovery messages. In the worst case, the immediate transmissions of recovery messages can double the number of messages of the basic computation. As an example consider a case where the local termination is detected after every single message that has to be processed. Although it is theoretically always possible to construct a scenario in which the algorithm exhibits this pathologic behavior, we can largely avoid it in practice by deferring the transmission of credit recovery messages for a certain amount of time. While this increases the termination detection delay and thus, it also increases the overall configuration delay, we found that the reduction of recovery messages that results even from small delays makes it worthwhile to rely on this strategy. In fact, for some scenarios, the reduced number of recovery messages also leads to a reduction of the overall discovery delay despite the additional delay introduced by deferring the recovery messages.

3.2.3.8 Integration

To integrate the previously described extensions for resilience and to detect the successful termination, we first extend the algorithm described in Section 3.2.3.3 by wrapping the *Receive_** and *Send_** procedures of the configuration algorithm using the procedures shown in Algorithm 5 and Algorithm 6 and we introduce the timer procedure and the recovery procedure shown in Algorithm 7 and Algorithm 8, respectively. For the sake of clarity, we assume that the procedures are executed sequentially and thus, we omit the necessary synchronization.

```
Receive_Message(Message)
  If (Message.GetEpoch() != Computer.Epoch) return;
  Computer.Termination_Timer.Stop();
  Computer.Processing = True;
  Computer.Credits.Add(Message.Get_Credit());
  Do_Receive_Message(Message);
  Computer.Processing = False;
  Computer.Termination_Timer.Start(TIMEOUT_VALUE);
```

Algorithm 5 – Receive Message Wrapper Procedure

As explained earlier, the wrapper for the receive procedure (see Algorithm 5) filters out messages that do not correspond to the current epoch of the computer (Line 2) and it collects the credits that are attached to the configuration algorithm messages (Line 5). Furthermore, in order to initiate the recovery of credits later on, it controls a timer to collect the credits (Line 3 and 8). In order to perform the local termination detection on the coordinator, the procedure additionally toggles a flag that denotes ongoing local computations (Line 4 and 7).

```
1:  Send_Message(Message)
2:    Message.setEpoch(Computer.Epoch);
3:    Message.setCredit(Computer.Credits.Split());
4:    Do_Send_Message(Message);
```

Algorithm 6 – Send Message Wrapper Procedure

Similarly, the wrapper for the send procedures (see Algorithm 6), attaches the epoch (Line 2) and a fragment of the credit (Line 3) to each message before it is transmitted (Line 4).

```
1:  Termination_Timer_Expired ()
2:    Send_Recover_Credits(Computer.Epoch, Computer.Credits);
3:    Computer.Credits.Clear();
```

Algorithm 7 – Termination Timer Procedure

The expiration procedure for the termination timer (see Algorithm 7) that is running on every computer is also very simple. Upon expiration, each computer just transfers its locally stored credits to the coordinator (Line 2) and removes them (Line 3). Analogue to messages of the configuration algorithm, the transmission of credits must also contain the epoch.

```
1:  Receive_Recover_Credits (Epoch, Credits)
2:    If (Computer.Epoch != Epoch) return;
3:    Computer.Credits.Add(Credits);
4:    If (! Computer.Processing && Computer.Credits.Is_Complete())
5:      // terminate successfully
```

Algorithm 8 – Credit Recovery Procedure

When a set of credits arrives at the coordinator (Algorithm 8), the coordinator uses the epoch to filter outdated messages (Line 2). Thereafter, it adds the credits to the local credits (Line 3) and it determines whether the local computation has been terminated and whether all credits have been recovered. If both conditions hold, the local termination has been detected. In all other cases, the basic computation is still running.

```
1:  Detect_Failure (Identifier)
2:    Send_Notify_Failure(Identifier) // to coordinator
```

Algorithm 9 – Detect Failure Procedure

The previous set of procedures suffices to implement the basic termination detection protocol. In order to deal with connectivity fluctuations and disconnections, we add the procedure shown in Algorithm 9. The purpose of this procedure is to signal all detected failures to the coordinator (Line 2). To do this, the computer that detects a communication failure sends the identifier of the failing computer to the coordinator.

```
1:  Receive_Notify_Failure (Identifier)
2:    If (! Is_Participant(Identifier)) return;
3:    Stop_Computation();
4:    Computer.Epoche += 1;
5:    Computer.Credits.Clear();
6:    Reset_Configuration_Objects(Identifier);
7:    Record_Removed_Participant(Identifier);
8:    If (Computer.Is_Coordinator())
9:      Send_Notify_Failure(Identifier); // to all remaining participants
10:     Computer.Credits = new Credit(); // reset the credit to initial value
11:     Create_Initial_Message(…);     // restart the configuration algorithm
```

Algorithm 10 – Receive Failure Notification Procedure

As shown in Algorithm 10, the coordinator uses this to distribute the failure message consistently to all remaining computers that participate in the configuration process (Line 9). In response to that, all computers terminate their current computations (Line 3), they increase their current epoch (Line 4) and they clear their credits (Line 5). Furthermore, they perform the corrective actions for the removed computer as described in the previous section (Line 6). In addition to that, the coordinator creates a new credit (Line 10) and restarts the configuration algorithm in the new epoch with the corresponding message (Line 11). As we pointed out earlier, the implementation of this procedure also requires an adequate synchronization which is omitted for the sake of clarity. Since it is possible that two different computers detect a failure of one computer in the same epoch, the coordinator also needs to filter out duplicate notifications (Line 2). To do this, it may either keep track of the removed computers or it can keep track of the computers that are still included (Line 7). A duplicate failure notification in two different epochs can never occur since the participating computers will never use the removed computer again. To guarantee this, they also need to keep track of the removed participants.

```
1:  Detect_Resource_Fluctuation ()
2:    If (Reservations_Valid()) return;   // ignore if reservations are valid
3:    If (Computer.Credits.Is_Cleared()) // retrieve credit if none available
4:       Credits.Add(Send_Retrieve_Credit());
5:    Computer.Termination_Timer.Stop();   // perform corrective actions
6:    Computer.Processing = True;
7:    Reset_Affected_Reservations();
8:    Computer.Processing = False;
9:    Computer.Termination_Timer.Start(TIMEOUT_VALUE);
```

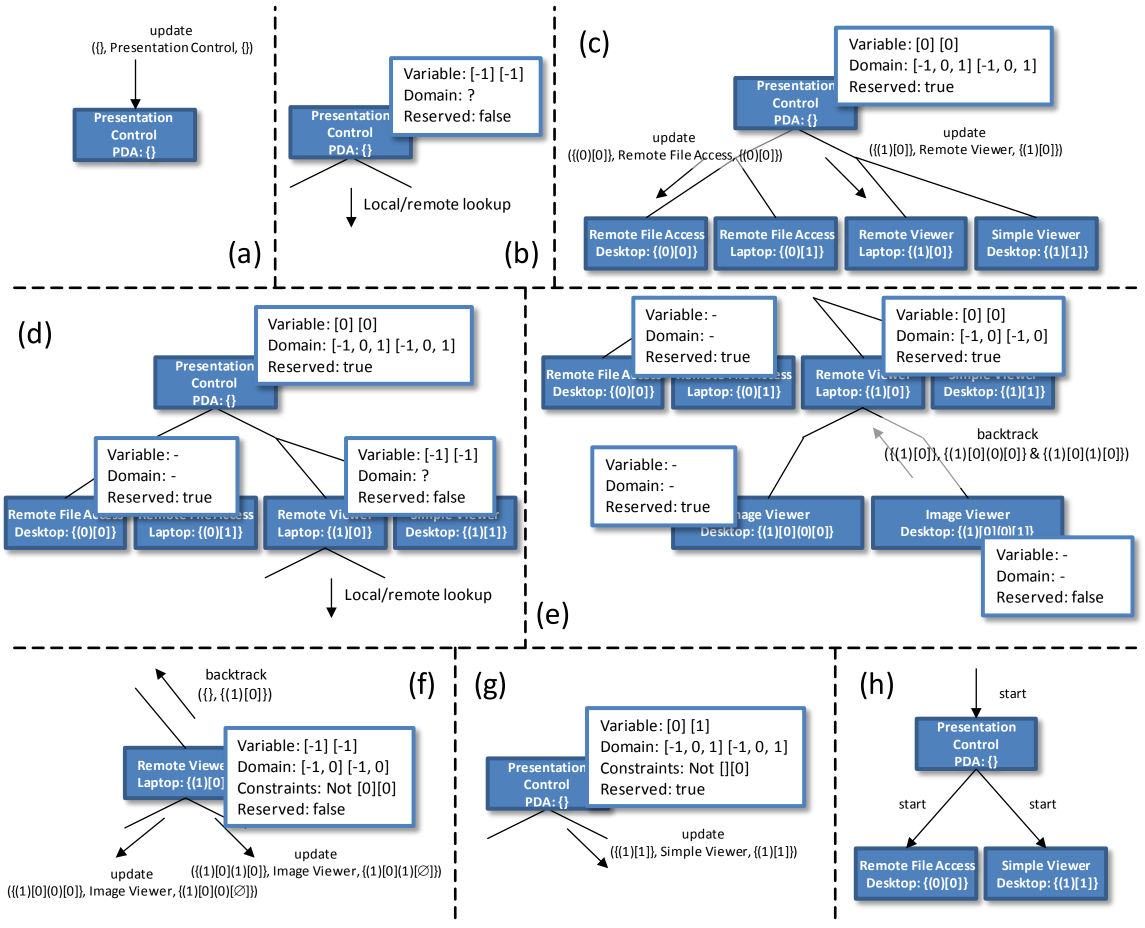
Algorithm 11 – Receive Failure Notification Procedure

Finally, to deal with resource fluctuations, we add the procedure shown in Algorithm 11. To avoid unnecessary overhead, the procedure first checks whether the resource fluctuation affects the current resource reservations (Line 2). If not it simply returns. If some resource reservations are affected, the procedure ensures that there are some credits available, if not it retrieves some from the coordinator (Line 3 and 4). Thereafter, it simply performs the corrective actions (Line 7) which might result in new backtracking messages that will be sent to other computers. To ensure that the procedure does not impact the overall termination detection algorithm it is wrapped in the same statements that are used by the wrapper procedure for the reception of messages (Line 5, 6, 8 and 9).

3.2.3.9 Example

To describe the configuration process performed by the algorithm in a more dynamic manner, we will use the exemplary smart peer group and the presentation application shown in Figure 15. When the presentation application needs to be configured, the algorithm is initiated by calling the *Receive_Update* procedure with *{}*, an identifier that locally identifies the contract of the *Presentation Control* and the value *{}*. This signals that an anchor should be started (a).

Since this is the first time that the configuration algorithm sees an update for *{}*, it creates a configuration object for the contract. Using the contract, it determines that *Presentation Control* has two dependencies, thus it creates a two-dimensional variable (*[Input]*), (*[Output]*). To determine the domain of the variable, i.e. possible options to satisfy the dependencies, the algorithm performs local and remote lookups (b). Thereby, the algorithm discovers the following options: *File System (desktop) {((0)[0])}*, *Remote File Access (laptop) {((0)[1])}*, *Remote Viewer (laptop) {((1)[0])}* and *Simple Viewer (desktop) {((1)[1])}*. Thus, the domain for the variable is *([-1, 0, 1]), ([-1, 0, 1])*. For the new variable, the initial assignment is *([-1]), ([-1])*.



(a)
update
({}, Presentation Control, {})
Presentation Control PDA: {}
(b)
Presentation Control PDA: {}
Variable: [-1] [-1]
Domain: ?
Reserved: false
Local/remote lookup
(c)
Presentation Control PDA: {}
Variable: [0] [0]
Domain: [-1, 0, 1] [-1, 0, 1]
Reserved: true
update
({(0)[0]}, Remote File Access, {(0)[0]})
update
({(1)[0]}, Remote Viewer, {(1)[0]})
Remote File Access Desktop: {(0)[0]}
Remote File Access Laptop: {(0)[1]}
Remote Viewer Laptop: {(1)[0]}
Simple Viewer Desktop: {(1)[1]}
(d)
Presentation Control PDA: {}
Variable: [0] [0]
Domain: [-1, 0, 1] [-1, 0, 1]
Reserved: true
Variable: -
Domain: -
Reserved: true
Variable: [-1] [-1]
Domain: ?
Reserved: false
Remote File Access Desktop: {(0)[0]}
Remote File Access Laptop: {(0)[1]}
Remote Viewer Laptop: {(1)[0]}
Simple Viewer Desktop: {(1)[1]}
Local/remote lookup
(e)
Variable: -
Domain: -
Reserved: true
Variable: [0] [0]
Domain: [-1, 0] [-1, 0]
Reserved: true
Remote File Access Desktop: {(0)[0]}
Remote File Access Laptop: {(0)[1]}
Remote Viewer Laptop: {(1)[0]}
Simple Viewer Desktop: {(1)[1]}
backtrack
({(1)[0]}, {(1)[0](0)[0]} & {(1)[0](1)[0]})
Variable: -
Domain: -
Reserved: true
Image Viewer Desktop: {(1)[0](0)[0]}
Image Viewer Desktop: {(1)[0](0)[1]}
Variable: -
Domain: -
Reserved: false
(f)
backtrack
({}, {(1)[0]})
Remote Viewer Laptop: {(1)[0]}
Variable: [-1] [-1]
Domain: [-1, 0] [-1, 0]
Constraints: Not [0][0]
Reserved: false
update
({(1)[0](0)[0]}, Image Viewer, {(1)[0](0)[∅]})
update
({(1)[0](1)[0]}, Image Viewer, {(1)[0](1)[∅]})
(g)
Presentation Control PDA: {}
Variable: [0] [1]
Domain: [-1, 0, 1] [-1, 0, 1]
Constraints: Not [][0]
Reserved: true
update
({(1)[1]}, Simple Viewer, {(1)[1]})
(h)
start
Presentation Control PDA: {}
start
start
Remote File Access Desktop: {(0)[0]}
Simple Viewer Desktop: {(1)[1]}

Figure 22 – Configuration Process

The algorithm continues to add the value *{}* to the local knowledge which states that the contract bound to the configuration object is required. Thereafter, the algorithm calls the

Validate_Constraints procedure and determines that the current assignment *([-1]), ([-1])* is not valid, since the instance is used according to the local knowledge. Note that this is a result of the built-in constraints presented in 3.2.2. Next, the algorithm determines a valid assignment *([0]), ([0])* and reserves the resources. The reservation finishes successfully and the algorithm continues to send parallel update messages to the *Remote File Access {((0)[0])}* and the *Remote Viewer {((1)[0])}* (c).

When the update message for the *Remote File Access* arrives, the algorithm creates the configuration object. Since the contract does not introduce further dependencies, the configuration object will simply contain a zero-dimensional variable and the algorithm will not perform any further lookups. After the configuration object has been created, the algorithm adds the value to the local knowledge and it checks the constraints. Thereby, it performs the resource reservation successfully and it stops without sending further messages (d).

In response to the update for the *Remote Viewer*, the algorithm creates a new two-dimensional variable with the domain *([-1, 0]), ([-1, 0])*. After checking the constraints and successfully completing the resource reservation, the algorithm sends two updates, both to an *Image Viewer* with the IDs *{((1)[0](0)[0])}* and *{((1)[0](1)[0])}*, respectively.

The first update message creates a new configuration object and finishes successfully. The second update fails due to a lack of resources. Thus, the *Trigger_Backtracking* procedure determines that the minimum conflicting sets consist of exactly one set of component instances that contains both instances of the *Image Viewer* (e). Note that although the *Remote File Access* is also configured on the desktop, its identifier will not be added to the conflict set since it has nothing to do with the shortage on displays. Furthermore, the algorithm does not need to add the complete path to the anchor to the constraint as it can be gradually generated whenever a conflict is escalated. Following the traversal strategy, *{((1)[0](1)[0])}* is picked as the smallest identifier and a backtrack message is sent to is parent. Additionally, the instance is deactivated and all potentially reserved resources.

When the backtracking message arrives at the *Remote Viewer*, the algorithm will determine whether it has to create any new links. Since both identifiers contained in the conflict set are local variables, no new link must be created. Therefore, the algorithm continues to add a mutual exclusion constraint between *{((1)[0](0)[0])}* and *{((1)[0](1)[0])}* to the local knowledge. In cases where added conflicts are not conflicts between linked instances, the addition of new links between the assigning instance and the instance that recorded the constraint is necessary to ensure that the constraint evaluation always considers all relevant variable assignments of the present situation.

Since the *Remote Viewer* cannot create a valid assignment, it creates a backtracking message that contains its own identifier and sends it to its parent. Thereafter, the *Remote Viewer* is deactivated and its constraints are checked again. Thereby, the algorithm releases all resources, assigns *([-1]), ([-1])* and creates updates that will eventually release previously bound instances (f).

When the *Presentation Control* receives the backtracking message, it adds the constraint that the *Remote Viewer* can never be started and assigns another value for the *Output* dependency. It selects the *Simple Viewer {((1)[1])}* and it creates an update (g). When the update arrives, the *Simple Viewer* will be reserved and the algorithm stops.

When the algorithm succeeds, the application must still be started. Therefore, a parallel traversal of the tree-structure starting from the application anchor is sufficient. This will not result in conflicts, since each configuration object has already reserved the resources for the chosen bindings (h). After the application has been started, all configuration objects that have been created can be removed.

3.3 Discussion

This chapter has introduced and formalized the configuration problem introduced by PCOM applications. The formalization allows us to utilize various existing algorithmic solutions to the problem. Based on the underlying system model of a smart peer group, we have identified the requirements on distribution and resilience. Furthermore, we have derived the requirements on completeness, efficiency and optimism from the overall vision of providing a seamless and distraction-free user experience.

We resolve the resulting conflict between efficiency and completeness in favor of completeness in order to avoid user frustration from false negatives. As a consequence, we can rule out incomplete algorithmic solutions and we focus our discussion of possible approaches on existing complete algorithms. Due to the requirement on distribution and efficiency, we can also rule out non-distributed algorithms and algorithms that are not capable of utilizing the inherent parallelism of pervasive systems. Finally, in order to satisfy the requirement of optimism, we cannot rely on algorithms that are unfolding the search space as this would result in higher configuration delays in scenarios that are comparatively easy to solve.

On the basis of this rationale, we select the asynchronous backtracking as underlying algorithm to solve the configuration problem. In order to apply this algorithm, we must provide a suitable mapping between the configuration problem and a Distributed Constraint Satisfaction Problem. To create this mapping, we represent dependencies with the potential candidates to resolve them as variables and domains, respectively. The structural constrains and the resource constraints, introduced by the application and the system model, are defining the constraints of the Distributed Constraint Satisfaction Problem. Since automatic configuration must only find a partial assignment of the corresponding Distributed Constraint Satisfaction Problem, we introduce a pseudo-value in the domain of each variable. This value is assigned to variables that are not required in the configuration. To ensure that the pseudo and non-pseudo values are assigned as needed, we introduce an additional set of constraints.

In order to allow the on-the-fly construction of the mapping during the configuration, we assign the pseudo value virtually to all variables of the configuration problem that have not been created. Furthermore, we introduce an ordering between variables that can be constructed dynamically with local computations. The potentially high number of variables resulting from the mapping may lead to a significant communication overhead. This problem is exacerbated by the fact that only a subset of them is required. In order to reduce the communication overhead, we optimize the number of communication links between variables by combining the ones that are always transmitting the same piece of information. Furthermore, since we know the semantics of the constraints that represent the application model, we can also avoid the transmission of messages that do not lead to changes. The ability of constructing the mapping on-the-fly is a necessary precondition to achieve the overall goal of optimism. The optimizations to reduce the communication overhead improve the efficiency of the asynchronous backtracking algorithm when applied to the configuration problem.

As a last step, we introduce a credit-based termination detection protocol to detect the successful termination of the algorithm and we make modifications to deal with the unavailability of computers during the execution of the algorithm. The key idea thereby is to dynamically introduce additional constraints for unavailable computers. These constraints ensure that the components on unavailable computer are not – and can no longer be – used as part of the configuration. Since asynchronous backtracking does not impose restrictions on the point in time when constraints are detected and propagated, these modifications can be introduced in a straight-forward manner without affecting the correctness. However, due to the fact that we are relying on a credit-based termination detection protocol, we also need to deal with lost credits that may be in transit or stored on an unavailable computer. To do this, we restart the algorithm every time a computer becomes unavailable. In order to differentiate the messages created before and after a restart, we introduce an epoch that needs to be transmitted with each message. Before a restart, the epoch is increased monotonically in a consistent manner on all remaining computers of the smart peer group. As a consequence, it can be used easily to drop outdated messages.

In summary, we get an integrated approach for automatic configuration that is capable of fulfilling the requirements on distribution, resilience, completeness and optimism by design. In the next chapter, we show how this approach can be extended to automatic adaptation. Thereafter, we discuss how this approach can be integrated into the component system presented in the previous chapter. In Chapter 6, we evaluate how well the approach fulfills the last remaining requirement, namely efficiency. Due to the nature of the problem, an approach that achieves completeness cannot be efficient enough for every possible scenario. However, as our simulations and experiments indicate, the approach works well in many typical scenarios and it is often preferable over incomplete approaches. Finally in Chapter 7, we discuss how other system software deals with the problem of automatic configuration and in Chapter 8, we describe possible future extensions.

4 Automatic Adaptation

This chapter introduces and formalizes the problem of adapting a PCOM application. It discusses the overall problem complexity and derives the requirements on approaches for automatic adaptation. As we will show in this chapter, automatic adaptation can be seen as an optimization problem on top of automatic configuration (Handte, Herrmann, Schiele, Becker, & Rothermel, Automatic Reactive Adaptation of Pervasive Applications, 2007). As a result, we mainly focus on the discussion of the differences to automatic configuration and we refer to the previous chapter for a detailed description of the commonalities. Thereafter, we present our approach for automatic adaptation. To motivate the approach, we briefly introduce and classify possible solutions. Thereafter, we describe the high-level rationale for the optimization heuristics as well as the relevant implementation details. The basic goal is to perform the optimization without increasing the communication overhead of the configuration algorithm. Towards this end, we incorporate a greedy heuristic into the configuration algorithm that operates on a specifically constructed value and variable ordering. After discussing the algorithm, we close the chapter with a summary.

4.1 The Adaptation Problem

Automatic adaptation denotes the task of modifying an existing and usually invalid composition of components in such a way that it can be executed as an application. Consequently, the composition resulting from automatic adaptation is also subject to the same structural and resource constraints as automatic configuration. Thus, automatic configuration can be seen as a sub problem of automatic adaptation and unless the original composition is not taken into account at all, they are not equivalent. As we show later on, this also increases the overall complexity.

A primary reason for taking the original configuration into account is usually the desire to minimize the user distraction resulting from an adaptation. A more detailed look at this goal enables us to classify the sources for this distraction into two basic categories. The first category entails modifications during the adaptation of an application that are perceivable by the user. The second category entails the decreased responsiveness of the application during the course of an adaptation.

For an interactive application, for instance, perceivable changes are usually modifications of components that are responsible for providing a part of the user interface. For a component-based distributed application, this implies that replacing a component that provides some service in the background may go unnoticed while replacing another one may immediately affect the perceivable representation of the application. On a more abstract level, we can thus conclude that modifying or replacing some components may be less distractive than others.

Besides that, the adaptation process itself may be distracting as it interferes with the execution of the application which may, for instance, increase its response time. Such effects are partly a result of the fact that the adaptation is performed reactively, i.e. at a point in time when the application can no longer be executed and adaptation is required. However, even for proactive adaptation it can be very challenging to avoid all effects. This can be attributed to the fact that the adaptation process itself competes with the application for shared resources and that some delays, e.g. for switching from one configuration to another, can solely be minimized but not nullified, in general.

For reactive adaptation, we can classify the sources for delays into two broad categories. First, the algorithm that computes the new configuration requires some time to finish its task. Secondly, switching from one configuration to another configuration also requires some time, e.g. to transfer the application-specific state from one configuration to the other. In the following, we refer these two time intervals as search and reconfiguration latency, respectively. Unfortunately, both latencies cannot be minimized independently – in general. Since the reconfiguration latency may critically depend on the characteristics of the old and the new configuration, finding a new configuration that causes a minimal reconfiguration latency increases the search latency. Furthermore, due to the fact that finding a single configuration is an NP-complete problem, the absolute minimization of the reconfiguration latency can easily lead to search latencies that exceed the savings with respect to reconfiguration latency by far. Thus, in order to optimize the overall latency, it is necessary to trade-off the search and reconfiguration latency in an adequate manner.

In order to motivate our approach on solving the problem of automatic adaptation, we first discuss an example using the application introduced in the previous chapter. Thereafter, we formalize the overall problem in the most general form. On the basis of the problem formalization, we briefly outline the resulting problem complexity and we derive the requirements for approaches for automatic adaptation.

4.1.1 Example

Figure 23 shows an exemplary smart peer group similar to the smart peer group shown in Figure 15. The group consists of four computers, i.e. a personal digital assistant (PDA), a laptop, a desktop and a server, that provide a set of resources and components for the distributed presentation application introduced previously. In contrast to the original example, the *Simple Viewer* component in this example can support two different parameterizations. To simplify the description, we refer to these parameterizations as *Fast* and *Slow*, however, this does not necessarily imply that the *Fast* parameterization is "better" than the *Slow*. From a system's point of view, both parameterizations can be used to resolve the *Output* dependency. The technically important difference between the parameterizations is the utilization of *CPU* resources. The *Fast* parameterization requires 100 units of *CPU* whereas the *Slow* parameterization requires only 50 units.

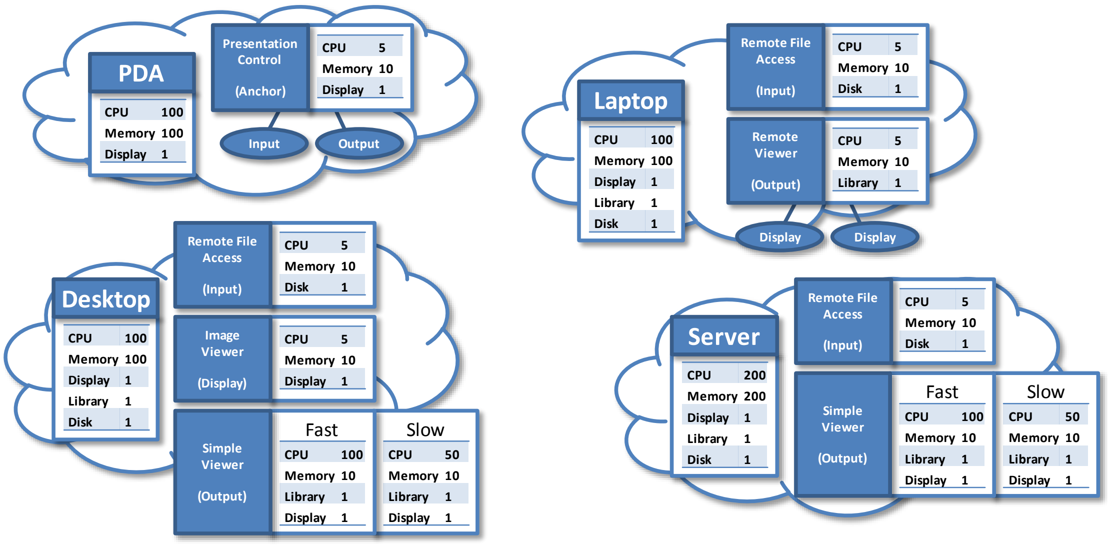


Figure 23 – Extended Exemplary Smart Peer Group

Given these components with the corresponding parameterizations, there are a number of valid configurations that can be used to execute the application. These configurations differ depending on the choices for the *Input* dependency, i.e. the location of the *Remote File Access*, and the *Output* dependency, i.e. the parameterization and the location of the *Simple Viewer*. Without considering the resource constraints, there are five possible options for *Output* and three possible options for *Input*. Thus, there are fifteen structurally valid configurations. However, due to a lack of *Display* resources on the desktop, it is not possible to use the *Remote Viewer* in any configuration. Furthermore, due to a lack of *CPU* resources on the desktop, it is not possible to use the *Remote File Access* on the desktop together with the *Fast* parameterization of the *Simple Viewer* on the same computer. As a result, the valid configurations in this example sum up to eleven.

To continue, assume that initially the application has been configured with the *Presentation Control* on the PDA, the *Remote File Access* on the laptop and the *Simple Viewer* with the *Fast* parameterization on the desktop. After this configuration has been started successfully, the laptop leaves the smart peer group, for instance, because the user decided to turn off the laptop to save the remaining battery power. As shown in Figure 24, the initial configuration will become invalid due to the unavailability of the *Remote File Access* induced by the unavailability of the laptop. As a result, the now invalid configuration needs to be adapted in order to continue the execution of the application.

Given the remaining three computers, i.e. the PDA, the desktop and the server, with their components and their possible parameterizations, there are multiple ways of adapting the configuration in such a way that it becomes valid again. Figure 25 shows three adaptations resulting in the configurations *A*, *B*, and *C* that exhibit significant conceptual differences. Depending on the properties of the smart peer group and the implementation of the individual components, these may also result in vastly different reconfiguration latencies and user distraction.

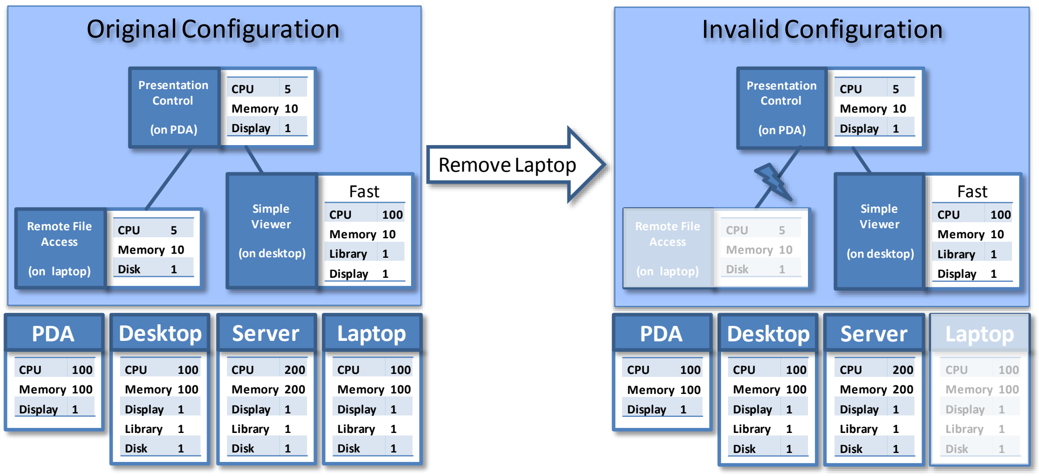


Figure 24 – Effects of Unavailable Laptop

Configuration *A* and *B* can be derived from the original configuration shown in Figure 24 by using the *Remote File Access* component on the desktop to resolve the unresolved *Input* dependency. Yet, if this is the only change to the configurations, they will not be valid due to a lack of *CPU* resources on the desktop. As a consequence, the utilization of the *Remote File Access* component on the desktop necessarily implies a change to the configuration of the *Output* dependency as well. In configuration *A*, this is reflected by replacing the *Fast* parameterization of the *Simple Viewer* on the desktop with the *Fast* parameterization of the *Simple Viewer* on the server. In configuration *B*, the *Simple Viewer* on the desktop is adapted by changing its parameterization to *Slow*. Due to the switch from the *Fast* parameterization to the *Slow* parameterization, the *Simple Viewer* requires only 50 units of *CPU* resources. The freed resources can then be used by the *Remote File Access* component. In configuration *C*, this shortage of *CPU* resources on the desktop is avoided altogether by using the *Remote File Access* component on the server instead of on the desktop. As a consequence, it is not necessary to change the configuration of the *Output* dependency.

The actual cost in terms of resulting user distraction for each of these configurations may vary depending on the characteristics of the smart peer group and the component implementations. Due to the nature of reactive adaptation, the distraction resulting from the replaced *Input* dependency cannot be avoided. Yet, whether it is less distractive to use the *Remote File Access* on the server or on the desktop depends on the difference between the reconfiguration latencies. If the reconfiguration latency for the *Remote File Access* on the server is lower than the reconfiguration latency for the desktop, this reconfiguration is clearly favorable as it does not require further changes to the configuration of the *Output* dependency. If the reconfiguration latency for the *Remote File Access* on the server is higher, then it might make sense to consider the configurations *A* and *B* as well. When comparing the costs of these configurations, it is not sufficient to solely compare the reconfiguration latencies. The reason for this is the (avoidable) structural change implied by configuration *A* which may cause additional user distraction because the user interface of the application is no longer displayed on the desktop. Thus, to determine the optimal configuration it is necessary to jointly compare all factors causing user distraction.

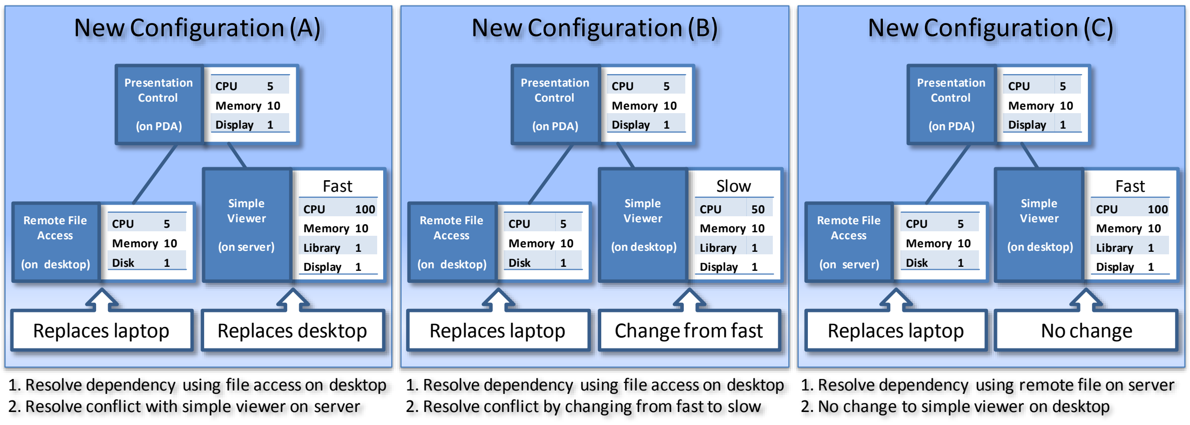

New Configuration (A)
Presentation Control (on PDA)
CPU 5
Memory 10
Display 1
Remote File Access (on desktop)
CPU 5
Memory 10
Disk 1
Simple Viewer (on server)
Fast
CPU 100
Memory 10
Library 1
Display 1
Replaces laptop
Replaces desktop
1. Resolve dependency using file access on desktop
2. Resolve conflict with simple viewer on server
New Configuration (B)
Presentation Control (on PDA)
CPU 5
Memory 10
Display 1
Remote File Access (on desktop)
CPU 5
Memory 10
Disk 1
Simple Viewer (on desktop)
Slow
CPU 50
Memory 10
Library 1
Display 1
Replaces laptop
Change from fast
1. Resolve dependency using file access on desktop
2. Resolve conflict by changing from fast to slow
New Configuration (C)
Presentation Control (on PDA)
CPU 5
Memory 10
Display 1
Remote File Access (on server)
CPU 5
Memory 10
Disk 1
Simple Viewer (on desktop)
Fast
CPU 100
Memory 10
Library 1
Display 1
Replaces laptop
No change
1. Resolve dependency using remote file on server
2. No change to simple viewer on desktop

Figure 25 – Possible Adaptations

Besides from demonstrating the interrelation of the different sources of user distraction, the example also demonstrates two further key challenges of automatic adaptation. First, it shows that the resulting user distraction cannot be captured independently for each sub-tree of the configuration. Instead, the distraction resulting from an adaptation is given by the total distraction resulting from all effects on all sub-trees of the application. Secondly, it shows that due to resource conflicts it is not possible to minimize the distraction independently within the sub-trees of an application. As an example consider the configurations *A* and *B*. There the choice of using the *Remote File Access* on the desktop makes it impossible to keep the configuration of the *Output* dependency as it is.

Finally, it should also be immediately apparent that minimizing the user distraction in terms of replaced user interface components or reconfiguration time can easily increase the search latency. Clearly, the search latency can be minimized by computing only one configuration but the resulting configuration can introduce an intolerably high distraction with respect to the remaining factors of distraction. To handle this, we define the reconfiguration latency and the user distraction resulting from replaced components as optimization goals during the adaptation and we incorporate the minimization of the search latency as an overall design goal for the adaptation algorithm.

4.1.2 Formalization

In order to formalize the adaptation problem, we can reuse most definitions that we introduced for the configuration problem. To avoid the duplication of all definitions, we would like to refer to Section 3.1.2 for detailed descriptions. However, to make the following description readable, we remind that we model a configuration as a tree of instances $G = (E, V)$ with its corresponding mapping to contracts $\chi : V \to C$.The contracts of instances sharing directed edges adhere to the Boolean matching function $\mu(d, p) : D \times P \to \{true, false\}$ between component demands $\delta(c_i) = \{d_{i,1}, ..., d_{i,n}\} \subset (D \cup \{\})$ and component provisions $\pi(c_i) = p_i \in P$. In order for $G = (E, V)$ to represent a valid configuration, we furthermore require that each component demand is met by exactly one provision of one instance and that the smart peer group is able to satisfy the resource requirements of all contracts for all instances of the configuration simultaneously – see conditions one to five in Section 3.1.2.

In the most general form, the adaptation problem can be seen as a configuration problem in which we search for the best configuration, i.e. the one that introduces the least cost, as opposed to anyone. As a result, we require some global function ψ_{global} that determines the cost of a given configuration by mapping it to some integer value in N_0, for example. The input parameters of this global cost function depend on the cost factors that should be considered. As indicated by the example in the previous section, the cost factors can depend on the changes that are made to instances as well as changes that are made to contracts. These changes can be evaluated using the original tree of instances $G^{old} = (E^{old}, V^{old})$ with its mapping to contracts $\chi^{old} : V^{old} \to C^{old}$ and their new pendants, i.e. G^{new} and χ^{new}. Thus, for a specific problem we require the global cost function to map the old configuration and all potential new configurations to some non-negative costs.

Contrary to this definition, it is usually possible and desirable to represent the global cost function as an aggregation of local costs. In addition to simplifying the formulation of the global cost function, the availability of a local cost function facilitates the development of local heuristics that, for instance, try to avoid seemingly bad options – that is, options that are introducing high costs. To define such a local cost function, we can base the computation of the global costs on the costs for individual modifications. In the subsequent section, we describe a cost model that allows the local approximation of the resulting costs on the basis of changes to the path of an instance to the parent.

This model essentially allows us to assign costs to the individual edges of G^{new}. Using these local costs of edges in E^{new}, we can then compute the global costs by summing up the costs of the individual edges, i.e. ψ_{global} is then given by $\sum_{i=1}^{|E^{new}|} \psi_{local}(e_i \in E^{new})$ where ψ_{local} depends only on parts of G^{old}, χ^{old}, G^{new} and χ^{new}. On the basis of the local cost function, we can then define the adaptation problem as global minimization problem over the search space given by the corresponding configuration problem. That is, given the smart peer group $M = \{m_1,...,m_l\}$ with the existing and potentially invalid configuration $G^{old} = (E^{old}, V^{old})$, $\chi^{old} : V^{old} \to C^{old}$ find a tree $G^{new} = (E^{new}, V^{new})$ with a function that maps instance to contracts $\chi^{new} : V^{new} \to C^{new}$ such that G^{new}, χ^{new} resembles a valid configuration – according to the conditions one to five in Section 3.1.2. Thereby, minimize the following term:

(6) $\sum_{i=1}^{|E^{new}|} \psi_{local}(e_i \in E^{new})$

4.1.3 Complexity

Since the adaptation problem is essentially an extended version of the configuration problem, it should be immediately clear that the complexity of the adaptation problem can only be equal or higher than the complexity of the configuration problem. The question whether the problem exhibits a strictly higher complexity depends on the characteristics of the local cost function. Clearly, if the local cost function is constant and zero, i.e. it assigns zero costs to all options, both configuration and adaptation exhibit the same complexity. However, for non-trivial cost functions, the overall problem of adaptation is strictly more complex since the problem lies no longer in NP.

To follow this argumentation, consider that if the adaptation problem would lie in NP, it must be possible to determine an algorithm that validates a solution in polynomial time. So given a non-deterministically “guessed” solution, we need to validate whether the solution is correct. To do this, we need to determine whether the solution is correct with respect to the conditions one to five stated in Section 3.1.2. As discussed previously, it is possible to derive an algorithm that does this in polynomial time.

Yet, according to condition six stated in Section 4.1.2, we also need to determine that the solution is optimal, i.e. it needs to exhibit minimum costs. Although there are specific cases, e.g. cases where the costs are exactly zero, where we can guarantee optimality, we cannot derive an algorithm that is capable of dealing with all cases. To clarify this, consider that validating the optimality of a result requires the comparison with other solutions, in general. Since finding a single solution lies already in NP, performing the computation of configuration as well as the comparison cannot lie in P.

4.1.4 Requirements

Based on the formalization and the complexity analysis of the adaptation problem, we can derive the requirements on solving this problem automatically. For the sake of the following argumentation, it is important to stress that we are focusing on reactive adaptation, i.e. adaptation in cases where the currently executed configuration has become invalid already. Thus, analogous to the configuration problem described in the previous chapter, the user cannot use the application unless its configuration has been adapted successfully. Due to this reason, the overall requirements of automatic configuration remain valid for automatic adaptation as well (see Section 3.1.4).

Specifically, this means that we must also require completeness in the sense that an algorithm for automatic adaptation should be able to find a new configuration, if it exists. Otherwise, the user cannot continue to use the application. Undoubtedly, this could easily frustrate users. Furthermore, since the work of the user is interrupted until the new configuration has been found, we must also demand efficiency. Apart from that the overall scenario for automatic adaptation corresponds to the one of automatic configuration. Thus, we must keep the requirements for optimism, distribution as well as resilience. However, due to the fact that automatic adaptation should minimize the user distraction, we need to extend the requirements of automatic configuration accordingly.

Optimality

Under the assumption that all valid configurations are equally well suited for adaptation, the search for a new configuration corresponds to the configuration problem. However, when an application needs to be adapted, there is still an invalid configuration available. Clearly, this configuration might lack some necessary component instances or it might require more than the available resources. Yet, if the adaptation algorithm does not take this configuration into account, the resulting configuration might differ significantly from the original one. As a result, even small changes that can be easily fixed, e.g. by changing the parameterization of a single component, can cause significant changes.

Besides from introducing a higher delay by an unnecessarily increased reconfiguration latency, this can also lead to changes that are immediately perceivable by the user and which can therefore, increase the distraction resulting from adaptation. Of course, due to the fact that we are focusing on reactive adaptation, some distraction cannot be avoided. If the level of distraction is modeled in terms of a suitable cost function, it becomes apparent that approaches for automatic adaptation should try to minimize the resulting costs and ideally, they should be able to find the optimal solution.

Unfortunately, due to the complexity of the resulting problem for a non-trivial cost function, determining the optimal solution conflicts with the goal of efficiency. In fact, even seemingly small problems can easily cause intolerably high adaptation delays since the corresponding optimization algorithm needs to compute and evaluate multiple configurations. Thus, minimizing the perceived user distraction requires a deliberate trade-off between the decreased distraction due to configurations with lower costs and the increased distraction for higher search delays. In essence, this means that algorithms for automatic adaptation need to make use of adequate optimization heuristics. In the next section, we present such a heuristic and we show how it can be integrated into the configuration algorithm described in the previous section.

4.2 Approach

To minimize the overall user distraction resulting from adaptation, we first define a basic cost model that approximates the user distraction in terms of reconfiguration latency and replaced components. The cost model allows us to express the problem of minimizing the user distraction mathematically as minimization problem. The proposed cost model captures all relevant details of the original (invalid) configuration but in order to avoid a high overhead during its initialization, it abstracts from all cost factors that depend on the properties of the new configuration. While this may result in an imprecise estimation, it ensures that the necessary computations do not conflict with the requirement of optimism. Specifically, the initialization of the model does not require the complete or even partial unfolding of the search space. All necessary computations can be performed within a single traversal of the available parts of the original configuration which must be done anyway.

Even with such a simple cost model for approximating the user distraction, the resulting optimization problem is NP hard. As a consequence, it is technically possible to reuse existing approaches for constraint optimization to tackle this problem. However, due to the computational complexity of the problem, we cannot target a complete or a bounded-error optimization since this can easily lead to intolerable search latencies. Similarly, we cannot perform the optimization within an a priory fixed bound since a meaningful bound will vary depending on the preferences of the user, the type of application and last but not least the characteristics of the smart peer group. To clarify this, we briefly discuss the benefits and limitations of a possible optimization approaches in the next subsection.

On the basis of this discussion, we propose a light-weight optimization strategy that is a combination of two greedy heuristics. As we will show, these heuristics can be incorporated easily in the previously described configuration algorithm. The basic idea behind both heuristics is the targeted utilization of the available degrees of freedom – that is the undefined value ordering and the partially defined variable ordering during configuration. As we will show in the following sections, simple yet effective value and variable orderings can be constructed on-the-fly from the cost captured by the cost model and their integration requires only insignificant amounts of additional ordering information that can be attached to existing messages.

Since both heuristics are greedy, they are obviously sensitive to the chosen starting point of the computation. In order to mitigate this, it is possible to execute the overall algorithm multiple times using a randomized starting point in cases where the resulting cost is comparatively high and the already experienced search latency is low. Such an incremental approach has the benefit of quickly producing a valid configuration while still providing a way of improving the previously found configurations. However, the evaluation of the overall approach suggests that in many scenarios the initial solution is sufficiently good already. Thus, iterations are only required in some cases. In these cases, it is often possible to find a solution that is close to the optimum with a small number of iterations.

4.2.1 Constraint Optimization

As indicated previously, it is possible to interpret the adaptation problem as a general Constraint Optimization Problem. This interpretation is quite natural considering the fact that we have already applied a constraint satisfaction technique to solve the configuration problem. Just like adaptation can be seen as a generalization of configuration, Constraint Optimization Problems can be viewed as a generalization of Constraint Satisfaction Problem and there are a set of equivalent ways of formalizing Constraint Optimization Problems.

Given the formalization of the adaptation problem described above, an obvious way would be to model it as a classical Constraint Satisfaction Problem that has a number of soft constraints with associated weights in addition to the hard constraints of the satisfaction problem. The goal is to find a solution that satisfies the hard constraints and that minimizes the sum of the weights of soft constraints that are not met. In this formulation the cost for different alternatives during adaptation are represented as soft constraints whereas the constraints that define a valid configuration can be expressed as hard constraints.

Under the assumption that the weights of the soft constraints are finite, it is simple to remove the differentiation between soft and hard constraints by assigning infinite weights to hard constraints. After the best solution has been determined it is simple to check its validity by determining whether it results in finite costs. If the best solution does not have finite costs, the underlying Constraint Satisfaction Problem is over-constrained and the returned solution does not represent a valid configuration. Otherwise, the returned solution is one of the best solutions, i.e. it is one of the configurations that exhibit the lowest adaptation costs.

Of course, it is possible to formulate a distributed variant of constraint optimization in order to model the fact that the constraints and variables are distributed among a set of agents. Similarly to Distributed Constraint Satisfaction Problems, there exists a considerable body of knowledge on solutions for Distributed Constraint Optimization Problems. In the following, we provide a rough overview over some of the existing solutions. Thereby, we first introduce a classification of the design space for optimization techniques and we use this classification to sort the pointers to different algorithms. However, since we are not relying of the implementation of these algorithms, we only describe their main ideas and we refer to the original publications for a more detailed description.

4.2.1.1 Optimization Technique Design Space

Existing optimization techniques can be classified on the basis of the guarantees that they provide with respect to their solutions. The two extremes are guaranteed optimality and no guarantees at all. In between it is possible to revise techniques that guarantee optimality with respect to a certain bound. For some problems, such bounds may be useful since a bounded optimization can result in significantly less computational effort than the complete optimization. However, the applicability of such techniques depends on the availability of a suitable bound and on the type of problem. Although, techniques that provide guaranteed or bounded optimality are theoretically appealing, they often induce a computational complexity that makes them prohibitively expensive. As a result, it is often not possible to rely on them in practice. Thus, many problems need to be solved by some heuristic that makes a suitable trade-off between computational effort and typical solution quality.

Besides classifying optimization techniques on the basis of their guarantees, it is also possible to classify individual algorithms depending on their characteristics, e.g. sequential vs. parallel, centralized vs. distributed, required amount of preprocessing, etc. Since automatic adaptation inherits the requirements of automatic configuration it should be clear that we are mostly interested in solutions that are parallel, distributed and require only little preprocessing. As a consequence, the following description of optimization techniques focuses on those techniques that have led to one or more algorithms with these characteristics. Albeit, we already mentioned that our approach is heuristic, we also describe some complete and bounded optimization techniques. This allows us to indicate why we are not relying on them as basis for automatic adaptation.

4.2.1.2 Complete Optimization

Probably the simplest complete optimization technique is the enumeration of all solutions with their respective costs. If this technique is applied, the best solution can be picked from the set of solutions as the one with the lowest costs. Obviously, this technique cannot be applied to continuous optimization problems or problems that do not exhibit a finite set of solutions. Fortunately, automatic configuration and adaptation of PCOM applications are both discrete and finite due to the fact that each component factory provides only a discrete set of contracts and that the application model does not allow cycles. However, for many practical problems – including adaptation – the enumeration of all solutions is costly. This can either be a result of the fact that the number of solutions is high or that computing them is expensive. As a consequence, the computational effort for enumerating all possible solutions can be considered to be prohibitive for automatic configuration.

The efficiency of complete enumeration can be improved by aborting the computation of a solution as soon as it is possible to show that the current partial solution cannot be part of the best. This technique is usually called branch and bound. The idea is to keep track of an (over-) estimated upper bound for the costs of the optimal solution. Initially, this bound is set to infinity. Thereafter, the solutions are computed sequentially. During the computation of the solutions, the costs for the current partial solution are estimated whenever the partial solution is expanded. The estimation must ensure that the costs are never overestimated. If this condition can be met, the computation of the current solution can be aborted as soon as the estimated costs for the partial solution are higher than the bound. The rationale for this is that there is at least one solution that exhibits lower costs. If the current partial solution can be expanded to a solution with lower costs than the costs estimated by the current bound, the bound can be adjusted – that is decreased – accordingly. As a result, the initially overestimated bound will eventually become exact.

The basic idea of branch and bound dates back to the 1960s and the first algorithm has been formulated in 1965 (Dakin, 1965). Since then there have been a good deal of improvements on the basic scheme that resulted in a variety of algorithms. An early example for an algorithm in the domain of distributed constraint optimization is the synchronous branch and bound algorithm described in (Hirayama & Yokoo, 1997). As indicated by its name, this algorithm simply translates the idea of branch and bound into a distributed setting by means of a predetermined static variable ordering. As a result, it does not provide any speedup since it does not support parallelism. The no commitment branch and bound algorithm presented in (Chechetka & Sycara, 2006) improves on this by allowing parallel computations in unrelated parts of the search space. Finally, the branch and bound ADOPT algorithm described in (Yeoh, Felner, & Koenig, 2008) resembles a completely asynchronous version of the branch and bound strategy.

Branch and bound drops a partial solution as soon as it is possible to prove that all possible completions cannot be better than the best solution found so far. To do this, it relies on an overestimated upper bound of the optimal costs of all solutions. Besides that it is also possible to use a lower bound underestimation over all remaining partial solutions to ensure that they cannot be better. This technique is usually called best first search. The key idea for this is to continuously (over)estimate the minimum costs of the current partial solution and to underestimate the minimum costs of all other solutions. As soon as it turns out that there might be another assignment for the current partial solution that *could* have lower costs, the current partial solution is dropped in favor of the seemingly better one. During the process of expanding a changing set of partial solutions, the estimations are gradually improved due to the fact that more or larger partial solutions are explored. If the process completes, it is clear that the current complete solution must have the lowest cost since the underestimated costs of all other solutions are already known to be higher.

Similarly to the branch and bound technique, there are asynchronous versions of the best first optimization technique that have been specifically developed to solve Distributed Constraint Optimization Problems. ADOPT (Modi, Shen, Tambe, & Yokoo, 2003), (Modi, Shen, Tambe, & Yokoo, 2005), for instance, is a polynomial space algorithm that enables asynchronous best first search. The communication structure in ADOPT is comparable to asynchronous backtracking, i.e. it organizes the constraint network as a depth first search tree with directed links that reflect constraints and it sends variable values across the links. However, instead of sending backtracking messages in case of a conflict, ADOPT continuously exchanges cost messages that contain a partial cost estimate. Within the tree these partial cost estimates are aggregated in a bottom up fashion and eventually they are transferred to the root as complete cost information. In order to eliminate inconsistent cost aggregations, cost message contain context information that is similar to the nogoods in asynchronous backtracking. As soon as a potentially sub-optimal variable assignment is revealed by means of a partial cost estimate, the variables are changed accordingly. Thus, the final solution will eventually be optimal. The basic scheme used in ADOPT can also be extended with so-called valued nogoods as done by ADOPT-ng (Silaghi & Yokoo, 2006). This enables ADOPT to adapt the original communication graph as done in asynchronous backtracking which can improve the speed of convergence.

4.2.1.3 Bounded Optimization

Even with advanced techniques that cutoff irrelevant parts of the search space such as branch and bound or best first search, the computational effort for finding the optimal solution is often too high. In addition, in many practical problems, it is not necessary to find the optimal solution. Instead, it suffices to find a solution that is close to the optimum. This relaxation can be exploited with optimization techniques that guarantee optimality with respect to a certain bound. Such bounds can either be expressed relative to the optimal solution in which case the distance between the produced solution and the optimal solution is guaranteed to be at most ε. As a result, these techniques are commonly referred to as ε- approximations or bounded error approximations. The other alternative for a bound is to express it absolute a priori, i.e. independent from the optimal solution.

A practical advantage of bounded optimization techniques is that for many problems finding a good solution or one that is close to the optimum is much simpler – in terms of computational effort – than finding the optimal solution. Or to put it in other words, after a good solution has been found, a lot of computational effort is required to achieve the last (minor) improvements. A potential disadvantage of bounded techniques is that it may be hard to define a useful bound. Due to their very nature, ε- approximations require an efficient way to estimate a good lower bound for the optimal solution. In addition, it is necessary to define a suitable ε, i.e. one that is sufficiently small but avoids the high effort for the complete optimization. A similar argument can be made about absolute bounds that are defined a priori. There the fixed bound may be too restrictive or simply not relevant. This is often problematic in cases where the cost of the optimal solution exhibits major fluctuations across different problem instances.

Bounded error and fixed bound approximations for Distributed Constraint Optimization Problems can be derived from complete optimization techniques. For instance, it is possible to derive a bounded error approximation from the best first search performed by the ADOPT algorithm (Modi, Shen, Tambe, & Yokoo, 2005). The basic idea behind this is rather straight-forward. Instead of switching directly to the partial solution that might exhibit lower costs, the algorithm simply defers switching until the seemingly best solution has a distance of ε. Thus, the first solution found by the algorithm has a distance of at most ε from the best possible solution. Similarly, it is rather simple to derive a fixed bound search from branch and bound techniques. To do this, one may simply stop the enumeration of further solutions, after the first solution with acceptable cost has been found.

Another way to perform this type of optimization is to apply constraint satisfaction techniques iteratively as described in (Hirayama & Yokoo, 2000). The main idea thereby is to consider the optimization problem as a sequence of gradually relaxed satisfaction problems. The sequence can for instance be created by gradually removing sets of constraints from the original problem. If the relaxation removes constraints systematically on the basis of increasing cost, the solution to the first (least relaxed) problem represents an optimal solution of the corresponding optimization problem. Clearly, this approach suffers from the potential combinatorial explosion of possible relaxations. However, bounded approximations greatly reduce the number of combinations since many (i.e. weak) relaxations do not have to be considered and others (i.e. too strong relaxations) can be ruled out as well. Interestingly, it is also possible to reverse the search order. That is to move from most relaxed to least relaxed problem. The key idea is that if all problems with a certain cost cannot be solved, all problem instances with lower cost (that is less relaxed problem) cannot be solved as well since they would only be more constrained.

4.2.1.4 Heuristic Optimization

Although, bounded optimization techniques can drastically reduce the computational effort for optimization in practical scenarios, in many cases they are still too expensive. To reduce the effort for optimization even further, it is necessary to rely on heuristic techniques. However, heuristics do not provide guarantees with respect to the quality of their solutions. From a theoretical point of view, many heuristic techniques can even return solutions that are arbitrarily bad with respect to quality. Yet, good heuristics usually take problem-specific domain knowledge into account to ensure that the produced solutions are acceptable for typical problem instances. As a consequence, good heuristics are often preferable over techniques with provable properties, because they provide good results for the relevant problem instances while causing a considerably lower computational effort.

From a high level point of view, heuristics can be classified into two broad categories. The first category entails constructive heuristics that try to come up with a good solution during the initial construction. Typical examples are greedy algorithms. Given a set of choices during the construction of a solution, a greedy algorithm may simply pick the one with the seemingly lowest cost without revisiting the choice again. Obviously, such techniques do not result in optimal solutions in cases where globally optimal choices cannot be made independently. However, due to their simplicity and low computational complexity, greedy algorithms are frequently applied to problems where there is only a low interdependency between individual choices. A simple way to improve greedy algorithms, is to increase their “field of view” so that they also consider the effects on (some) other choices as well but this increases the computational effort for making the next “best” choice.

The second category entails iterative heuristics that try to improve upon a solution gradually. Typical examples are algorithms that are based on local search. Starting from one solution, they construct and analyze a (hopefully) small set of neighboring solutions. If one of the neighboring solutions is better, they use this solution to compute the next set of neighboring solutions. This process is repeated until the neighboring set does not contain a better solution, in which case the current solution is returned as the seemingly best. Although this technique is fairly simple, it can produce good results. Yet in general, it suffers from two potential deficiencies. First, the set of neighboring solutions can be quite large. If this is the case, choosing the next solution from the set can be quite costly. Secondly, since the next solution must be strictly better than the current solution, the overall approach is only capable of finding local optima. Depending on the type of problem, there may be many such local optima and some of them may be significantly worse than the global optimum.

To overcome the problem of large sets of neighboring solutions, it is possible to reduce the set heuristically. An example for this approach is the so-called fast local search described in (Tsang & Voudouris, 1995). The basic idea is to heuristically deactivate some ways of creating neighboring solutions if they have not created better solutions in the past. If it turns out that a similar way of producing neighbors has succeeded, the deactivated production rules are activated again. The selection procedure of how production rules are activated or deactivated depends on domain knowledge. However, the general idea has been applied successfully to a number of different problems, including travelling salesman and resource scheduling.

To overcome the problem of getting trapped in local minima, there are two alternative approaches. First, one may simply run the overall search using multiple starting solutions. This approach can even lead to (provable) optimal results if it is possible to select the starting points in such a way that they cover all local optima. However, for non-trivial problems it is often hard or even impossible to perform this selection properly. Alternatively, one may modify the strictly greedy behavior of the iterations by integrating some form of noise (Selman, Kautz, & Cohen, 1994). Noise can be integrated randomly, for instance, by allowing the search algorithm to pick a seemingly worse solution or by introducing a non-neighboring solution in the set of neighboring solutions from time to time.

Instead of relying on the power of randomization, it is also possible to use a more systematic approach such as guided local search (Tsang & Voudouris, 1999), for example. The basic idea of Guided Local Search is to modify the cost function every time a local minimum is found by means of local search. The modification of the cost function is done in such a way that unwanted features of the locally optimal solution are penalized increasingly. So after the cost function has been modified, the solution might no longer be optimal since some of its unwanted features are now resulting in high costs. This strategy effectively directs the local search to different areas of the search space and thus, it can often avoid hangs in local minima.

Alternatively, it is also possible to utilize techniques that are inspired by natural phenomena. A frequently used example is simulated annealing, e.g. (Martin & Otto, 1993). A survey of applications of Simulated Annealing can be found in (Koulamas, Antony, & Jean, 1994). Simulated Annealing mimics the behavior of a solid that evolves to thermal equilibrium. The basic idea of its application to optimization problems is to perform local search that may sometimes pick a solution that does not reflect the optimal neighbor. The probability for picking such a suboptimal solution is based on a slowly decreasing temperature value – the higher the temperature, the higher the chance of selecting a suboptimal solution. Although, this technique is based on randomization, it is possible to show that it converges with high probability, given that the temperature decreases slowly enough.

Apart from simulated annealing, there are other optimization techniques that are based on natural phenomena. Although these techniques are iterative, they do not use local search directly. A primary example is genetically inspired optimization techniques that mimic evolution, e.g. (Affenzeller & Mayrhofer, 2002). The idea is to start off with a set of solutions – a so-called population – that are manipulated by means of genetic operators such as mutation or cross-over that are found in biological systems. Newly created solutions are evaluated by means of a fitness function. If the new solutions are fit enough they will survive and continue to evolve, if they do not withstand the fitness test, they will die. Genetic techniques can also be executed in parallel (Rivera, 2001) to speed up the optimization. Unfortunately, the overall approach also mimics some unwanted properties of evolution – that is, for many problems it requires a considerable amount of time and resources.

4.2.2 Cost Model

Independent from the chosen optimization technique, it is necessary to formulate a cost model to capture the quality of alternative solutions. As a minimum requirement for reactive adaptation, the cost model must be able to measure the total cost for each valid configuration that can be used to adapt a given invalid configuration. Although the cost may also be expressed in terms of an ordinal scale that simply provides an ordering, usually it is possible to measure the costs on the basis of an absolution scale.

In addition, it is frequently desirable to compute the total cost on the basis of local costs that result from individual decisions made during the configuration. Many of the above described techniques depend on the availability of a model that can be computed incrementally. As an example consider branch and bound that relies on cost estimates for partial solutions in order to reduce the search in areas of the search space that are provably irrelevant. Other examples are constructive heuristics that rely on local cost estimates for individual options to perform their greedy decisions.

In the following, we derive a cost model that approximates the cost of adapting PCOM applications. The proposed model supports the incremental computation of costs via aggregation and thus, it allows the utilization of a broad range of optimization techniques. Although, this model itself is problem-specific by its very nature, the proposed approach for optimization depends only on a subset of its properties. Thus, the optimization heuristics presented in the next section can be applied to other models as well, given that the main requirements are fulfilled.

4.2.2.1 Cost Factors

In order to understand the design rationale of our cost model, it is useful to revisit the possible sources of adaptation cost. As indicated previously, the overall adaptation cost can be broadly classified into cost resulting from search and cost resulting from modifications to the original configuration. Both types of cost cannot be minimized independently. Thus, we aim at minimizing the cost for search by means of a suitable – i.e. a light-weight – optimization technique. Consequently, the cost model itself does not have to capture the cost for search. Instead, the model can concentrate on the remaining cost factors, i.e. the cost for modifications.

The cost for modifications can, in turn, be classified depending on their sources. On the one hand, there are non-technical sources such as the distraction resulting from a modification to a component that provides a part of the user interface. Such sources cannot be captured without additional help from the component developer, since the actual cost depends on the functionality provided by the component. As a consequence, we enable the application developer to specify this manually per component instance. On the other hand, there are technical sources such as the delay introduced by switching from one configuration to another. These sources can be approximated automatically since the relevant factors are known. As a result, the cost captured by the model may consist of a combination of cost factors that are specified manually by the component developer and cost factors that are approximated automatically by the component system.

To enable the incremental estimation of costs, it is necessary to determine the total cost on the basis of the individual modifications that need to be performed. In general, we can distinguish the following two basic cases. First, the parameterization of one component instance may change and secondly, one component instance may be replaced by another component instance. When comparing the two modifications, it becomes clear that the cost for changing a parameterization can be neglected. This is an immediate result of the fact that the modification of a parameterization does not affect the structure of the application and it can always be handled locally. Of course, a modification of a parameterization may in turn lead to further modifications. However, these modifications induce further costs that should be aggregated by the model when looking at the overall costs. In contrast to modifications of parameterizations, replacing a component instance with another component instance will usually affect two computers that need to coordinate in order to avoid a loss of application-specific state. Moreover, these changes may cause visible effects and thus, they may introduce considerable costs. As a result, our model focuses on capturing the costs induced by replaced component instance.

The cost factors for replacing a component instance can be classified in factors that depend on the existing configuration and cost factors that depend on the new configuration. Cost factors of the existing configuration are the delay induced by storing the state of a component instance and by transferring this state to the parent instance. The cost factors that depend on the new configuration are the delay induced by transferring the state from the parent to the new instance and the delay induced by restoring the state. To simplify this, we can assume that the costs incurred by the transfers are independent from the original and the new configuration. Furthermore, we can also assume that they are directly proportional to the amount of the state held by the component instance. These simplifications are valid since the transfer time is usually bounded by the bandwidth of the network. Furthermore, we can neglect the delay induced for storing and restoring the state, since these costs are small in practice, e.g. in most cases the component instances will directly transfer its internal fields.

A potential complication arises from the mechanisms of the component system. In order to keep the application-specific state consistent without requiring more complex coordination, PCOM always replaces complete sub-trees. Thus, the state stored by the original component instance must enable the new component instance to restore the state of its children as well. Consequently, it is possible to approximate the state that needs to be stored for a component instance as the sum of the states of the original sub-tree spanned by it. However, it is noteworthy to point out that this is a course-grained estimation since the amount of state that needs to be transferred during restoration may depend on the new configuration as well. Thus, the real costs may be higher or lower.

To clarify this, consider that it is the deliberate goal of the component model to enable a high compositional flexibility. Thus, it is possible that a component instance *A* that does not require any further component instance may be replaced with a component instance *B* that requires two further component instances *C* and *D*. If component *A* is replaced, the overall state of *A* must be transferred to *B* which will then use this state to initialize *C* and *D*. Thus, the overall amount of state that needs to be transferred during restoration will be higher than the overall amount of state that needs to be transferred to store the state. Similarly, if *B*, *C* and *D* would be replaced by component instance *A*, the actual amount of state that needs to be transferred during restoration might be lower.

Yet, in order to account for these differences in the cost model, it would be necessary to compute the structural differences between the original and the new configuration during the adaptation. Due to the fact that some structurally valid configurations may not result in valid configurations due to resource constraints, this computation cannot be done in advance. As a result, including these factors in the cost model could significantly increase the costs incurred by search since comparing the cost of two alternatives would require the computation of both. As a consequence, we do not to model these differences. This limits the quality of the approximation but it greatly simplifies the computation of the cost model.

4.2.2.2 Formalization

To formalize the resulting cost model, we first need to define the notion of a replaced component instance. Since PCOM replaces complete sub-trees, the notion of replacement does not simply correspond to the removed component instances. Instead the replaced component instances are solely the removed instances whose ancestors have not changed. In other words, we can define the actually replaced instances as the set of the topmost instances of a configuration that have been removed.

We reuse the definition of the tree of component instances $G=(E,V)$ introduced previously to formalize the configuration and the adaptation problem, respectively. Remember that this tree $G=(E,V)$ consists of the nodes $V=\{v_1,...,v_n\}$ and the directed edges $E=\{e_1,...,e_{n-1}\}$ where v_i represents a component instance and $e_i=(v_j,v_k)$ represents a dependency of the component instance v_j on the component instance v_k. Based on this, we can define the parent relation $P(v_i)$ of component instance v_i as the set that contains the instances depending on v_i, i.e. $P(v_i):=\{v_j \mid (v_j,v_i)\in E\}$. Since $G=(E,V)$ represents a tree, the set returned by $P(v_i)$ is empty for the root and it contains one component instance in all remaining cases. Intuitively, we can define the ancestor relation $A(v_i)$ as the transitive closure of $P(v_i)$, i.e. $A(v_i):=P^*(v_i)$.

Given two trees of instances $G_{old}=(E_{old},V_{old})$ and $G_{new}=(E_{new},V_{new})$ representing the old configuration and the new configuration of an adaptation, we can easily compute the set of removed component instances as $V_{removed}=V_{old}-V_{new}$. Furthermore, we can compute the set of replaced instances as the set of topmost removed instances (whose ancestors have not been removed), i.e. $V_{replaced}=\{v_i \mid v_i\in V_{removed}\wedge\neg\exists v_j:(v_j\in V_{removed}\wedge v_j\in A(v_i))\}$.

The cost resulting from a replacement of a single component instance v_i can be computed locally as $C_{local}(v)$ from the amount of state $T(v)$ carried locally by the component instance and the amount of user distraction $D(v)$ caused by replacing the instance. Since these two factors are experienced independently upon replacement, we can consider them to be additive, i.e. $C_{local}(v) = T(v) + D(v)$. Clearly, in order to perform this computation it is necessary to express the user distraction relative to the amount of state, e.g. replacing the component instance is as distracting as transferring a certain amount of state, or vice versa. Alternatively, one may consider these factors as independent dimensions, but eventually they need to be combined somehow to support programmatic comparisons and decisions. Since the replacement of a single component instance always replaces the complete sub-tree, it is necessary to compute the total cost as the sum of the local costs of all component instances that are removed due to the replacement. To do this, we can first define the set of descendants $S(v_i)$ of a component instance v_i as the set of component instances that have v_i as an ancestor, i.e. $S(v_i) := \{v_j \mid v_j \in V \wedge v_i \in A(v_j)\}$. Now, given the set of descendants $S(v) = \{v_1, \ldots, v_s\}$ of a concrete component instance v we can compute the total cost $C_{total}(v)$ for replacing v as $C_{total}(v) = C_{local}(v) + \sum_1^s C_{local}(v_i)$.

Finally, we can extend the computation to span the whole set of replaced components by aggregating the total costs for all of them. This raises the question of how the total costs for individual replacements should be aggregated. For the costs that represent user distraction in terms of replaced component instances, it is quite clear that the costs are additive, e.g. replacing two user interface component instances is worse than one. However, the costs for state restorations are not necessarily behaving additively, e.g. replacing two component instances with the same amount of state will not necessarily double the reconfiguration latency. To clarify this, consider that the replaced component instances represent different parts of the tree. Thus, it is technically possible to replace them in parallel. As a consequence, one might argue to aggregate them as the maximum of all experienced individual replacement costs. However, this is also not a precise estimate in cases where the set of computers that participates in the configuration overlaps. As a consequence, we sum up the individual replacement costs in our model. So if the set of replaced components $V_{replaced}$ consists of the component instances $\{v_1, \ldots, v_r\}$, we get the total adaptation cost C for all replaced components as $C = \sum_1^r C_{total}(v_i)$. It is noteworthy that due to this simplification, it is also possible to express the total cost of an adaptation on the basis of the removed components, i.e. without considering the replacements at all. To do this, we can simply add up the individual local costs $C_{local}(v)$ of removed component instances $V_{removed}$.

4.2.2.3 Discussion

As discussed, the cost model described above introduces a number of simplifications that may impact the quality of the estimate. Undoubtedly, it is possible to introduce other models that provide better estimates. However, a more detailed cost model will also increase the effort that is required to compute estimates. To clarify this, reconsider the three main simplifications:

First, the automatically computed parts of the model are solely based on the amount of state held by a component instance. Thus, the model neglects the differences of storing and restoring the state for individual computers. In order to improve the estimate, we could, for instance, introduce weights to model the differences in storing time. However, since the time will depend on dynamic factors such as the CPU utilization, they might fluctuate during the adaptation. As a consequence, the more detailed information may become outdated at runtime. Similarly, in order to improve the estimate with respect to restoration time, we would have to take the new configuration into account. Thus, we would have to compute new configurations in order to precisely estimate the resulting delays. Although this is possible in general, the induced computational and communication effort can easily nullify the potential gains.

Secondly, the cost model sums up the costs that occur in different sub-trees. As explained earlier, this estimate may be imprecise since it may be possible to replace component instances in parallel. However, if the sets of computers that host the affected component instances overlap, the assumption of a parallel replacement is invalidated. Similarly, if the computers are communicating using a shared network, the assumption will break as soon as the amount of state to transfer exceeds the available bandwidth. Thus, in order to get a more detailed estimate, the cost model would have to account for the dynamically changing network conditions and it would have to base its computation on the details of the new configuration as well. As discussed earlier, it is questionable whether the achievable gains in estimation quality would justify the additional effort.

Thirdly, the model does not capture any costs for parameterizations. As a consequence all parameterizations are considered to be equivalent. In contrast to the previously described simplifications, this simplification is not a result of a performance optimization. Instead, it would be perfectly viable to model the cost of different parameterizations. However, since we assume that parameterizations reflect modes of operation that provide similar functionality, we argue that switching between different parameterizations will be a comparatively lightweight operation in general. As a result, there is no technical need to model the differences with respect to cost. If this assumption would not be valid, it would be easy to introduce the associated costs as local costs. Similarly, it would be possible to extend the optimization approach described in the subsequent sections.

Even though the proposed cost model is simple, it provides us with valuable information at a very low overhead. Yet, due to the reasons discussed above the costs cannot be mapped directly into a delay. Moreover, since the total costs also include other factors, i.e. the user distraction resulting from replacements, it is hard to interpret the value without further knowledge. As discussed previously, this could be mitigated by considering different cost factors separately. However, any approach will eventually define a mapping between the factors to reason about individual options. In summary, the total cost information as defined by the cost model enables guided replacement decisions during adaptation since the available cost information allows us to order component instances on the basis of their cost.

In order to determine the cost information for each component instance, it is sufficient to perform a single traversal of all component instances of the original configuration. This traversal is required in order to perform the aggregation of costs for sub-trees. The traversal can be parallelized and it is possible to reuse existing messages that are required to trigger the transitions in the component lifecycle. After finishing the traversal, each component instance knows its own total costs and the total costs of the children that are incurred when the corresponding instance is replaced. As we will explain in the following sections, it may be necessary to distribute the information even further. To do this, it is possible to piggy-back the information on other existing messages.

4.2.3 Adaptation as Constraint Optimization

On the basis of the cost model presented in the previous section, we can interpret adaptation as optimization problem in which we want to minimize the cost (Handte, Herrmann, Schiele, Becker, & Rothermel, Automatic Reactive Adaptation of Pervasive Applications, 2007). Since we want to ensure that the new configuration resulting from adaptation is valid, we can reuse the transformation described in Section 3.2.2. Thus, the variables of the optimization problem correspond to individual dependencies of contracts. The domains are defined by the contracts that can be used to resolve the corresponding dependency. The constraints reflect the structural and the resource constraints.

In addition, we must use the original (invalid) configuration and the concrete values of the cost model to introduce soft constraints with penalty values. The soft constraints can be broken but if they are broken, the resulting cost will occur. The constraints can be created gradually, by traversing through all options in the search space starting from the application anchor. Given a reused instance v_i of the original configuration with a dependency that is resolved by an instance v_j, we assign $C_{total}(v_j)$ to a single constraint for all options of the corresponding variable that would cause the reuse of v_j. These options are essentially all contracts that are created by the factory that issued the contract of v_j. The goal is then to find a solution that does not break any hard constraint while minimizing the sum of the penalty values for broken soft constraints.

As an example, consider the exemplary scenario introduced in Section 4.1.1 and three possible adaptations shown in Figure 25. The original (invalid) configuration consists of two remaining components, i.e. the *Presentation Control* and the *Simple Viewer*. Now assume that the costs C_{local} of the *Simple Viewer* are given by $C_{local}(S)$ and the costs of the *Presentation Control* are given by $C_{local}(P)$. Since the *Simple Viewer* does not have any further dependencies, its total cost $C_{total}(S)$ corresponds to the local costs $C_{local}(S)$. Furthermore, the total cost of the *Presentation Control* $C_{total}(P)$ can then be computed as $C_{local}(P)+C_{local}(S)$. Note that the total cost of the *Presentation Control* does not contain any costs for the *Remote File Access*, since this component instance is no longer available. As a consequence, it is not considered by the cost model which is reasonable since it needs to be replaced in any case. On the basis of these costs, we can introduce soft constraints using the procedure sketched above. This results in soft constraints for reusing the *Presentation Control* and reusing the *Simple Viewer*. If we evaluate the costs for the possible configurations shown in Figure 25, we can assign the cost $C_{total}(S)$ to the configuration *A* as it replaces the *Simple Viewer*. Furthermore, we can assign the cost of 0 to the configuration *B* and *C* since both configurations reuse the *Simple Viewer* and the *Presentation Control*. The differences between configuration *B* and *C* do not affect the cost captured in the cost model and thus, they are considered to be equivalent.

4.2.4 Adaptation with Heuristic Optimization

The mapping outlined above allows us to apply any optimization technique for constraint optimization. However, since our goal is to minimize both, the search latency and the cost captured by the cost model, we do not apply a complete or bounded optimization technique. As explained earlier, both types of techniques may cause high overheads with respect to search. To minimize the search overhead, we propose the utilization of heuristic optimization as basis for adaptation. However, the decision to rely on heuristic search still leaves a considerable degree of freedom ranging from constructive techniques over iterative techniques to techniques that are inspired by natural phenomena.

To reduce the effort for search as far as possible, we propose to apply a constructive technique as primary basis for adaptation. In contrast to other heuristic optimizations that are based on iterations, a constructive optimization technique ensures that the initially constructed solution is likely to exhibit low costs already. Most other techniques do not exhibit this characteristic. Iterative techniques, for instance, start at some random starting point which may be arbitrarily bad. Thus, they exhibit an initial overhead for computing the starting point. This starting point is then subject to multiple iterations in order to improve the cost of the final solution. Clearly, every time a better solution has been found, it is possible to use this configuration for adaptation. However, since the computation of a single configuration may already be a time-consuming task, computing multiple configurations may not be a viable approach for many scenarios. A similar argument can be made about other approaches such as genetically inspired optimization, for example.

Of course, applying a constructive optimization technique is also not free of limitations. Depending on the concrete technique, constructing an optimized solution may be a computationally demanding process. For instance, if the construction process frequently needs to inspect the potential completions for the current partial solution, the overall process may lead to significant overheads when compared to a non-optimizing approach for finding a configuration. Alternatively, if the technique is very lightweight, it may result in a reduced or even insufficient solution quality. Iterative techniques based on local search, for instance, ensure that the final solution represents a local minimum. Although, this minimum may be distant from the global minimum, there is at least a guarantee for local optimality. Achieving even such weak guarantees via construction may be complicated.

In order to combine the benefits of constructive and iterative optimization techniques, we propose to extend the basic constructive optimization with an iterative scheme that applies randomization to vary the constructed solutions. The randomization naturally fills the gap resulting from seemingly identical choices that cannot be distinguished by a greedy heuristic. On the basis of this randomization, it is then possible to compute different solutions using the constructive optimization technique. Thus, it is possible to rerun the construction multiple times and to store the best solution. As a result, one may use the best configuration that has been found so far at any point in time. This enables us to continue the optimization in cases where a solution has been found quickly but exhibits comparatively high costs. Thus, it is possible to dynamically balance the benefits and limitations of constructive and iterative techniques.

In the following, we describe the details of the constructive optimization technique. Thereby, we discuss possible ways of introducing randomization. Since the iterative part of the optimization process is straight-forward, we do not discuss any details on this. The basic principle is simply to execute the constructive part multiple times and to store the best solution together with the cost. In Section 6.3, we discuss the effects of randomization and iteration on different scenarios.

Since any optimization technique needs to ensure that the resulting solutions are representing valid configurations, we integrate the heuristics directly into the configuration algorithm presented in the previous chapter. As a consequence, it is important to briefly revisit the basic principles of asynchronous backtracking. However, in order to avoid duplicate descriptions, we solely describe the most important concepts and we refer to Section 3.2.3.1 for more details.

4.2.4.1 Optimization Heuristics

For our optimization heuristics, we do not want to change the basic principles of asynchronous backtracking. This allows us to avoid additional effort for optimization and at the same time, it enables us to use the same algorithm for the initial configuration and adaptation. As a consequence, we need to integrate the construction heuristics in such a way that they do not change the communication flow and that they do not introduce significant computations. Besides from ensuring minimal effort, this also ensures that the important characteristics of our configuration algorithm such as completeness, resilience, distribution, etc. remain unchanged. Since reactive automatic adaptation inherits the requirements of automatic configuration, this approach ensures by design that all requirements except for the additional requirement on optimality can be met effortlessly.

As basic premises, it is necessary to remember that many backtracking algorithms already introduce a significant degree of freedom that may be used for optimization purposes. Specifically, many backtracking algorithms can support arbitrary value and variable orderings, i.e. they rely on the existence of an ordering but they are agnostic to the concrete implementation. It is a well established fact that the value and variable orderings may have significant impact on the performance of backtracking algorithms. For example, a value ordering heuristics such as min-conflict may significantly reduce the number of backtracking steps in practice, since it helps spreading the search through different parts of the space quickly. Similarly, it is possible to improve the performance of backtracking by modifying the variable ordering. The usual goal during reordering is to define the ordering in such a way that the most constraining variables are assigned first. Since it is hard to determine an optimal variable ordering, some backtracking algorithms even reorder variables dynamically at runtime, e.g. (Yokoo, 1995).

Instead of using value and variable ordering heuristics to improve the search latency, we propose to use them to optimize the constructed solution with respect to the resulting costs which reduces the reconfiguration latency. To do this, we use the information provided by the cost model, specifically the individual values for C_{total}, to define a partial value and variable ordering that guides the individual decisions made during value and variable selection. Thus, when the configuration algorithm is about to resolve a dependency, it will prefer options with lower costs over options with higher costs. Similarly, when the configuration algorithm needs to backtrack, it will first try to change the parts of the configuration that result in low costs.

The basic assumption for both orderings is that the total costs for replacing a component instance and its potential children are known and can be accessed locally on the corresponding computer. As discussed earlier, this can be achieved by traversing the configuration once. Apart from this, we only require information that is already stored on each computer of the smart peer group. Specifically, we solely require that each computer knows its own component instances and the component instances that are directly used by them. This assumption is not limiting since the local component instances need to know the used component instances anyway to interact with them.

On the basis of this information, we can dynamically construct the necessary orderings on-the-fly. The on-the-fly constructability of the ordering is an important precondition in order to ensure that the characteristics of the configuration algorithm are not changed. To clarify this, remember that it is a primary goal of the mapping of configuration to constraint satisfaction to enable the on-the-fly construction. This allows the algorithm to unfold only those parts of the search space that need to be unfolded. This, in turn, is particularly important since the basic mapping can easily result in a high number of variables, even for a comparatively low number of components.

In the following, we discuss how both heuristics can be integrated into the configuration algorithm. The result of this integration is a configuration algorithm that is capable of optimizing costs, i.e. an algorithm to solve the adaptation problem. It is noteworthy to point out that if the cost information is not available it is still possible to use the same algorithm to automatically compute a configuration. To do this, we can simply think of all costs being zero. As a consequence, the adaptation algorithm can be used for configuration as well.

4.2.4.2 Value Ordering Heuristic

The idea behind the value ordering heuristic is rather obvious. Starting from the application anchor, the configuration algorithm recursively resolves dependencies using some matching contract. Algorithmically, this is done by creating a variable for each dependency. Thereby, the domain of the variable is initialized according to the amount of contracts that can be used to resolve the corresponding dependency, i.e. with the exception of the pseudo value each value represents a certain contract. Without any value ordering, the configuration algorithm simply uses the first possible value that does not conflict with a known constraint. However, in many cases there will be multiple possible assignments and they may cause different costs. Thus, the value ordering should ensure that the algorithm first tries the values that cause no cost before it switches to values that introduce costs.

Intuitively, the overall value ordering should be considered whenever the value of the variable needs to be changed, e.g. due to a newly discovered resource constraint. As a consequence, it makes sense to determine the ordering once, i.e. at the point in time when the configuration algorithm creates the variable. The ordering can then be stored in the corresponding configuration object, e.g. by ordering the mapping between values and contracts according to the costs.

In order to compute the ordering, each configuration object needs three pieces of information. First, it needs to know whether the configuration object itself represents a contract that can be used to parameterize an existing component instance. If this is the case it needs to know which component instances are bound to each of the dependencies. Finally, it needs to know which contracts would allow the reuse of the potentially bound component instance.

Since the first piece of information also depends on the value selections for the parent variables, the information whether a configuration object allows the reuse cannot be determined locally. However, the parent configuration object can simply pass this information along recursively when the value for the corresponding variable is selected. The second piece of information can be gathered locally by inspecting the existing configuration and the third piece of information can be determined by comparing the factory of the potentially bound component instance with the factory that created the contracts returned by a remote query.

```
 1: Initialize_Configuration_Object (Contract, Is_Reuse, …)
 2:   ConfigurationObject Config = new ConfiguratonObject(Contract, Reuse, …)
 3: …
 4:   Dependency[] Deps = Contract.getDependencies()
 5:   For (int i = 0; i < Deps.length; i++)
 6:      // initialize one variable for each dependency
 7:      Contract[] Opts = PerformQuery(Deps[i])
 8:      Variable Var = ConfigurationObject.CreateVariable(Deps[i])
 9:      Contract BndCnt = GetCurrentConfiguraton(Deps[i])
10:      FactoryID BndFacID = (BndCnt==Null)? Null : BndCnt.getFactoryID()
11:      For (int j = 0; j < Opts.length; j++)
12:         FactoryID FacID = Opts[j].getFactoryID()
13:         Var.AddOption(Opts[j], (Is_Reuse && FacID == BndFacID))
14:…
```

Algorithm 12 – Variable Ordering Initialization

The relevant piece pseudo code for the initialization of the configuration object is shown as Algorithm 12. When a configuration object is used for the first time, the parent of the configuration object will send a Boolean flag that indicates whether the component instance represented by the configuration object can be reused. Furthermore, the parent will also pass the corresponding contract represented by the configuration object (Line 1).

In order to initialize the variables of the configuration object (Line 6-13), we step through the individual dependencies specified by the contract (Line 4). In a simplified version, we perform a set of remote queries to determine the potential contracts that can be used to resolve the dependency (Line 5). Thereafter, we can create the variable that represents the dependency (Line 7). Furthermore, we retrieve the globally unique identifier of the factory for the currently bound instance, if any (Line 8-9). This process can be done locally by inspecting the existing configuration.

Finally, we can iterate over the contracts that can be used to resolve the dependency and we add them as an option to the variable (Line 11-13). If combined with the pseudo value, the options will form the domain of the variable. The second parameter of the method specifies whether the corresponding option would allow the reuse of a component instance. A reuse is possible, if the configuration object already reflects a reused component instance and if a potentially existing instance is created by the same factory as the contract that corresponds to the option (Line 13).

When the configuration object needs to assign a value to a variable, it can now first check the options that have been added with the Boolean flag set to true and only if these options cannot be used, it considers the options that have been added as false. Furthermore, if a value is used for the first time, it can add the Boolean flag as part of the corresponding update message of the configuration algorithm. This ensures that the receiving system can perform the initialization correctly. Together these changes complete the variable ordering heuristic.

4.2.4.3 Variable Ordering Heuristic

Besides from preferring variable assignments with lower costs during value selection, we can also prefer variable assignments with lower cost during backtracking. To do this, we introduce a variable ordering heuristic that reflects the cost for replacements. The variable ordering heuristic makes use of the fact that the ordering between the variables of unrelated paths of the tree can be arbitrary. The general idea is that during backtracking, the algorithm should first change the variables that select instances causing low costs before it changes variables that select instances with high costs. Note that if two assignments conflict, the original algorithm changes the assignment of the variable with the lower priority first. Thus, by defining the priority on the basis of the total costs C_{total} of a potentially bound component instance for each variable, we can achieve the desired behavior.

While the value ordering heuristic can be integrated in a straight-forward manner, the integration of the variable ordering heuristic causes a subtle problem. To clarify this, consider that the correctness of asynchronous backtracking is based on the fact that the variable ordering is static. Thus, if a variable has assignments that can either reuse or replace an existing component instance, we could either assign the priority C_{total} to reflect a reused instance or 0 to reflect a replaced instance, but not both.

Both options are suboptimal: If we assign the priority 0 to the variable, we do not get the desired ordering. As a consequence, we should rather assign a priority of C_{total}. However, if the algorithm already detected that the existing instance cannot be used, e.g. due to a resource conflict, changing the new instance does not increase the cost. Thus, if a newly bound instance conflicts with other parts of the application, it is more cost-efficient to change it before even more parts of the application are replaced.

An example for such a variable can be seen in Figure 26. The left side of the figure shows a fragment of the existing configuration that consists of two instances *1* and *2* created by the components *A* and *B*, respectively. The right side of the figure shows an exemplary search space during adaptation. Contract *A* is selected when its parent variable is set to the value *X*. Under the assumption that the usage of contract A supports the reuse of component instance *1*, selecting contract *B* would reuse component instance *2*. Selecting contract *C* instead would require the creation of a new instance of type *C* and the restoration of the state of instance *2* in the new instance. As discussed, we can therefore either assign the priority of 0 to reflect the selection of contract *C* or we could assign the priority C_{total} of instance *1*.

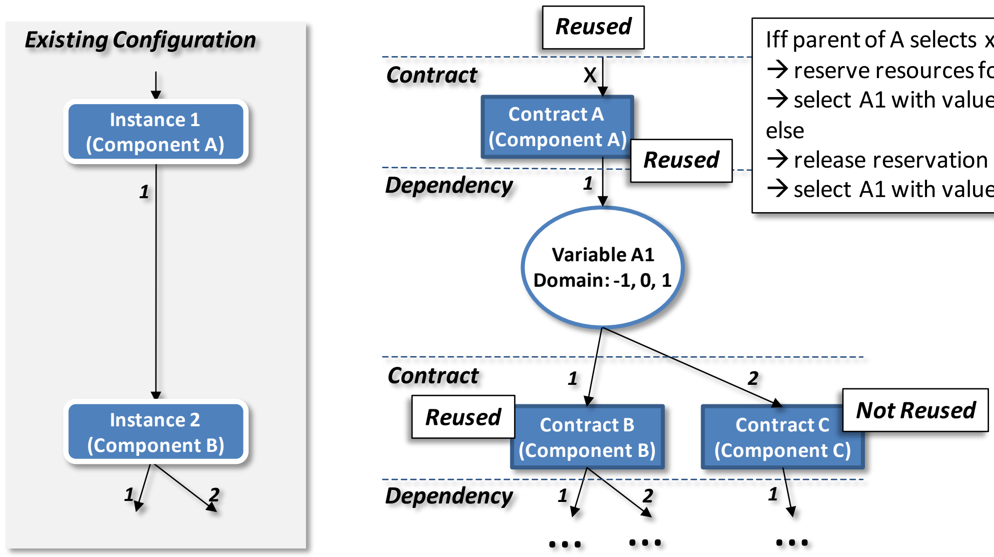


Figure 26 – Subtle Mapping Problem

To avoid this problem without further modifications to the basic principles of the configuration algorithm, we split such variables into two variables to assign individual priorities for each of them. To do this, we partition the possible value assignments into assignments that allow reuse and assignments that do not allow reuse. Note that this classification is already done to implement the value ordering heuristic (see Algorithm 12). However, since both variables are representing the same dependency, we also need to ensure that the algorithm does never select a contract for both variables. This can be done by introducing an additional pseudo value (-2) and a constraint that enforces this.

As an example for this, consider the exemplary search space shown in Figure 26. When the algorithm initializes the domain of *A1*, it detects that there are assignments that can reuse the existing instance (*A1*=0) and assignments that would replace the instance (*A1*=1). Thus, it splits the variable and creates a new pseudo variable *PsA1*. The domain of the variable *A1* consists of the assignments that reuse the instance, including the pseudo values -1 (i.e. sub-tree not required) and -2 (i.e. reuse not possible but sub-tree required).

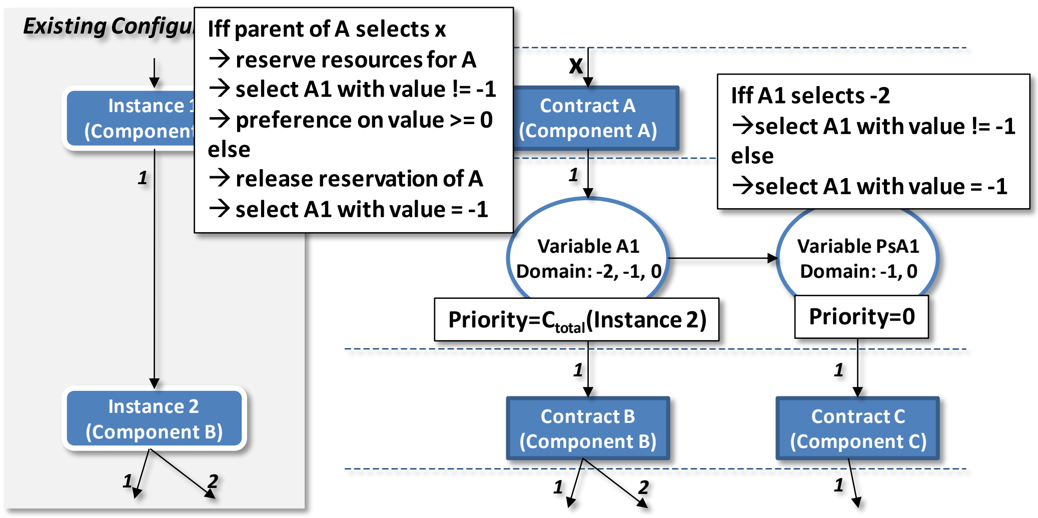

Existing Configu
Instance 1 (Component
Instance 2 (Component B)
Iff parent of A selects x
→ reserve resources for A
→ select A1 with value != -1
→ preference on value >= 0
else
→ release reservation of A
→ select A1 with value = -1
x
Contract A (Component A)
Iff A1 selects -2
→select A1 with value != -1
else
→select A1 with value = -1
Variable A1 Domain: -2, -1, 0
Variable PsA1 Domain: -1, 0
Priority=C_total(Instance 2)
Priority=0
Contract B (Component B)
Contract C (Component C)
1
2

Figure 27 – Extended Mapping

The domain of the variable *PsA1* is constructed from the assignments that do not reuse the instance and the pseudo value -1 (sub-tree not required or reuse possible). The algorithm adds a built-in constraint that models the fact that *PsA1* must only select a value if *A1* must be assigned but cannot reuse an existing instance (*A1*=-2). Since *PsA1* now shares a constraint with *A1*, the configuration algorithm needs to add a link from *A1* to *PsA1*. As a final step, the algorithm must assign priorities to the variables. This can now be done statically. The priority of *A1* is initialized with C_{total} of instance *1* since changing a selection (>=0) could add these costs. The priority of *PsA1* is 0 since changing a selection (>=0) will never add costs. The result of this procedure is shown in Figure 27.

In order to consider the priorities during backtracking, we need to replace the existing ID-based variable ordering. As discussed in the previous chapter, the online mapping performed by the algorithm requires that a parent variable has a higher priority than its children. The definition of C_{total} already ensures that a parent has at least the same C_{total} than its children. Thus, a parent will never have a lower priority. Yet, C_{total} does not establish a total ordering which is required by the algorithm. In order to create a total ordering, we can combine the priority-based partial ordering resulting from C_{total} with the lexicographic ID-based total ordering introduced in the previous chapter. For two variables *A* and *B*, we define the ordering between variables on the basis of the priorities and the identifiers as:

$$A < B \Leftrightarrow (priority(A) < priority(B)) \vee (priority(A) == priority(B) \wedge ID(A) < ID(B))$$

As indicated previously, this approach requires the computation of C_{total} for each component instance which can be done with a single traversal of the existing configuration. The traversal ensures that each computer knows C_{total} of the instances hosted by it. However, the algorithm needs to ensure that C_{total} is always available on each computer that requires it for a comparison.

To clarify which computers need the ordering information, consider that the configuration algorithm uses the ordering solely during backtracking in order to determine the variable that should be changed first. Thus, we can safely conclude that the priority information of a variable solely needs to be known by other variables that may be conflicting at some point in time. Of course, the exact sets of variables that may conflict is only discovered gradually.

A seemingly simple approach to fix this would be to distribute the priority information globally among all computers of the smart peer group. However, this would require a complete unfolding of the search space in order to detect the mapping between the dynamically created variable identifiers and the total cost of the potentially existing instances. Alternatively, we can dynamically distribute only the priority information that is really required for the variables that have been created so far. To do this without introducing additional messages, we may reuse the messages of the algorithm. A simple and effective approach would be to include the priority information for the transmitted identifiers in all messages of the algorithm, i.e. update, link and backtracking messages. Since computers can only detect that they have a conflict with another variable, if they already know the identifier, this approach guarantees that the computer also knows the priority of the variable.

However, if the mapping between the identifiers and priority information is stored on each computer, it is sufficient to include the priority only in link response messages. The reason for this is that backtracking messages can only result directly from resource conflicts or indirectly from a number of other conflicts that together cause a conflict. Since resource conflicts are always a result of local constraints, the priority values for all affected variables are always known already. Note that this is a result of the assumption that each computer knows C_{total} for its hosted instance. Due to the fact that the configuration algorithm creates new links to higher priority variables contained in a backtracking message, before accepting it, we can use the response message to a link request in order to distribute the priority information to this variable. Since it is guaranteed that the algorithm will only send backtracking messages to linked variables, this approach suffices for all cases, i.e. other variables are not known and thus, they cannot cause a conflict. Since the link establish is not done repeatedly, i.e. links are only established once and then reused if similar conflicts appear, this is also minimal.

4.2.4.4 Example

In the following, we outline an exemplary execution of the value and the variable ordering heuristic as part of the configuration algorithm using the scenario introduced in Section 4.1.1. As discussed, the original configuration consists of three component instances, i.e. the Presentation Control on the personal digital assistant, the *Remote File Access* on the laptop, and the *Simple Viewer* on the desktop. The initial configuration becomes invalid because the laptop leaves the smart peer group. As a consequence, the *Input* dependency of the *Presentation Control* is no longer resolved properly because of the induced unavailability of the *Remote File Access*.

After the laptop is no longer available, the component system detects and signals the unresolved dependency either by means of a timeout or due to a failing communication attempt which eventually triggers the adaptation process. As a first step, the component system notifies all remaining component instances and causes the appropriate transition in their lifecycle. This is done by traversing the tree of remaining component instances. At the same time, the system computes the values for the cost model according to the state held by the available component instances and other cost factors specified by the component developer. Finally, the traversal also creates the data structures required to execute the configuration algorithm.

After the traversal of the remaining component instances, the adaptation algorithm starts. The individual steps are depicted in Figure 28. Just like the original configuration algorithm, the adaptation algorithm starts with an update message for the application anchor (a). The update message contains the identifiers for the component instance that have been introduced in the previous chapter. Additionally, the update message also contains a flag that indicates whether an existing instance may be reused. Since the *Presentation Control* is still available, the flag is set to true.

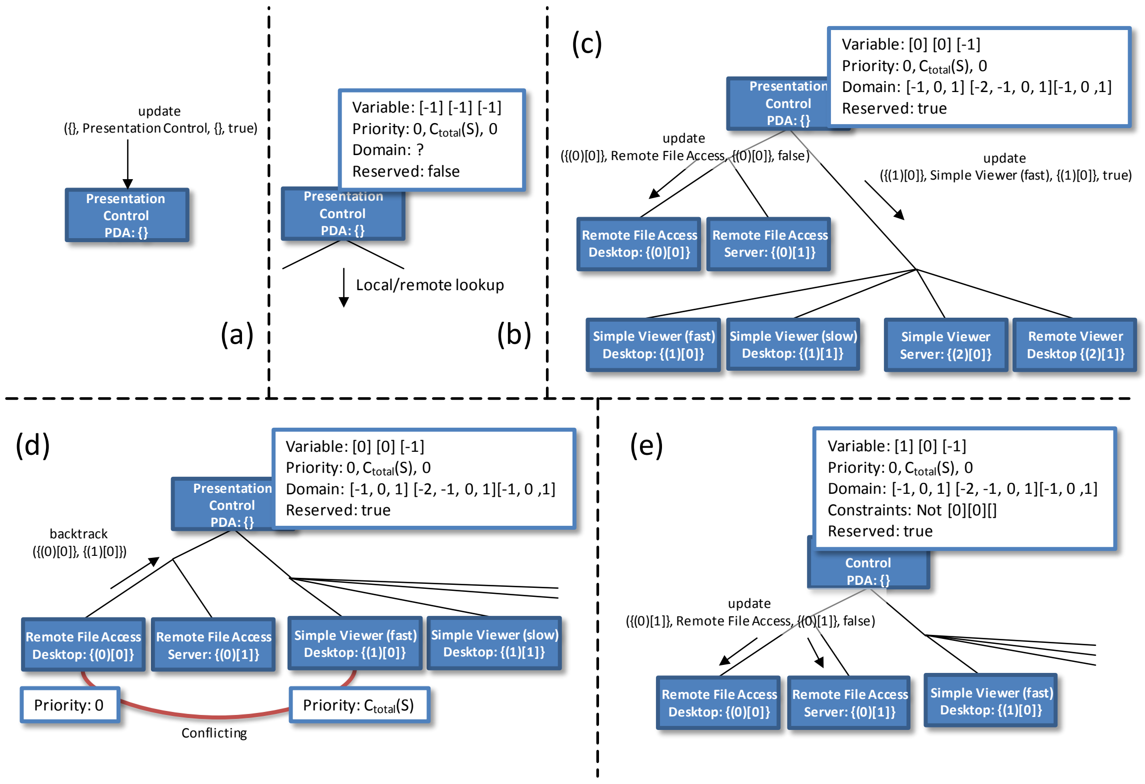

update
({}, Presentation Control, {}, true)
Presentation Control PDA: {}
(a)
Variable: [-1] [-1] [-1]
Priority: 0, C_total(S), 0
Domain: ?
Reserved: false
Presentation Control PDA: {}
Local/remote lookup
(b)
(c)
Variable: [0] [0] [-1]
Priority: 0, C_total(S), 0
Domain: [-1, 0, 1] [-2, -1, 0, 1][-1, 0 ,1]
Reserved: true
Presentation Control PDA: {}
update
({(0)[0]}, Remote File Access, {(0)[0]}, false)
update
({(1)[0]}, Simple Viewer (fast), {(1)[0]}, true)
Remote File Access Desktop: {(0)[0]}
Remote File Access Server: {(0)[1]}
Simple Viewer (fast) Desktop: {(1)[0]}
Simple Viewer (slow) Desktop: {(1)[1]}
Simple Viewer Server: {(2)[0]}
Remote Viewer Desktop {(2)[1]}
(d)
Variable: [0] [0] [-1]
Priority: 0, C_total(S), 0
Domain: [-1, 0, 1] [-2, -1, 0, 1][-1, 0 ,1]
Reserved: true
Presentation Control PDA: {}
backtrack
({(0)[0]}, {(1)[0]})
Remote File Access Desktop: {(0)[0]}
Remote File Access Server: {(0)[1]}
Simple Viewer (fast) Desktop: {(1)[0]}
Simple Viewer (slow) Desktop: {(1)[1]}
Priority: 0
Priority: C_total(S)
Conflicting
(e)
Variable: [1] [0] [-1]
Priority: 0, C_total(S), 0
Domain: [-1, 0, 1] [-2, -1, 0, 1][-1, 0 ,1]
Constraints: Not [0][0][]
Reserved: true
Control PDA: {}
update
({(0)[1]}, Remote File Access, {(0)[1]}, false)
Remote File Access Desktop: {(0)[0]}
Remote File Access Server: {(0)[1]}
Simple Viewer (fast) Desktop: {(1)[0]}

Figure 28 – Adaptation Process

When the update message is processed, the personal digital assistant creates a configuration object with an appropriate set of variables and it initializes their domains by performing a local and remote lookup for matching contracts (b). Note that although the *Presentation Control* has only two dependencies, the configuration object contains three variables. This is a result of the fact that the original configuration still has an instance that is bound to the *Output* dependency, i.e. the *Simple Viewer*, and thus, there is a need to split the variable representing the *Output* dependency. As discussed in the previous subsection, this is necessary to differentiate between contracts that induce the replacement of the instance and contracts that allow the reuse of the instance. Also note that due to the additional built-in constraint discussed previously, the two variables that represent the *Output* dependency are never resolved at the same time. The priority of the variables is set according to the total cost induced by a potential replacement. Thus, the priority of the first variable is 0, since the *Input* dependency is not resolved. Similarly, the priority of the last variable is 0, since this variable represents options that replace the component instance that is still bound to the *Output* dependency. The priority of the second variable is set to the total costs induced by replacing the *Simple Viewer*, since changing the values of this variable might lead to a replacement of the existing sub-tree bound to the *Output* dependency.

After the remote and local lookup has been completed, the domains of all variables are initialized according to the contracts (c). In this scenario, there are two options to resolve the *Input* dependency. Thus, the domain of the first variable is -1, 0, and 1. Furthermore, there are four options to resolve the *Output* dependency. These options can be grouped into options that induce the replacement of the existing component instance and options that support the reuse. The two options that support the reuse are essentially parameterizations of the existing component instance. The other two options are contracts from factories of other components, potentially residing on other computers. As a consequence, the domain of the second variable is -1, 0, 1 and -2 to denote the fact that the third variable can be used as well. Furthermore, the domain of the third variable is -1, 0, and 1. After the initialization is completed, the algorithm resolves the dependencies by assigning values to the variables. According to the mapping between contracts and variable domains, the algorithm selects the *Remote File Access* on the desktop and the *Fast* parameterization of the *Simple Viewer* on the desktop. This is done by assigning 0 to the first and the second variable. Due to the fact that the second variable is not set to -2, the third variable will be set to -1 automatically. Note that this is a result of the built-in constraint that prohibits multiple resolutions of the same dependency by pseudo variables. The modified variable assignment causes new update messages that are sent to the desktop. Just like the initial update message, the new update messages contain the identifier and the flag to indicate the reuse. Since the *Input* dependency is not resolved, the flag is set to false for the update message sent to the *Remote File Access*. However, due to the fact that both, the *Presentation Control* and the *Simple Viewer*, can be reused, the flag is set to true for the remaining update message.

When the update messages are processed by the desktop (d), the desktop creates the necessary configuration objects and performs the usual steps of the configuration algorithm. When the algorithm performs the resource reservation for the second configuration object, it detects the shortage of *CPU* resources and it derives the set of conflicting assignments as the *Fast* parameterization of the *Simple Viewer* and the *Remote File Access*. The original configuration algorithm would now initiate backtracking according to the lexicographical ordering between the variables involved in the conflict. Thus, the *Output* dependency would be the first dependency that would be changed. However, due to the variable ordering heuristic, the adaptation algorithm considers the priorities of the variables as most significant ordering information and it uses the lexicographical ordering information only to break ties. Thus, when the priority of the *Simple Viewer* is higher, the adaptation algorithm will first change the *Input* dependency before changing the *Output* dependency. If we assume for the scenario that the *Simple Viewer* carries at least some state, its total cost will be higher than zero. As a consequence, the backtracking would be performed on the *Input* dependency first. Note that in this example, there is no need to create links during backtracking. However, if there was a need to create a link, it would be necessary to distribute the priority information as discussed in the previous subsection.

Finally, when the backtracking message is processed by the personal digital assistant, the computer records the constraint and tries to assign other values. In this example, it assigns 1 to the first variable and leaves the second and third variable unchanged. This causes two update messages that are sent to the *Remote File Access* on the desktop and on the server. After they have been processed, the algorithm terminates successfully and the new configuration is given by the *Presentation Control* on the personal digital assistant, the *Fast* parameterization of the *Simple Viewer* on the desktop and the *Remote File Access* on the server. In this exemplary scenario, we can see that both, the value ordering and the variable ordering heuristic, can be effective. The value ordering heuristic ensures that the algorithm prefers the parameterizations of existing component instances, i.e. it prefers to use combinations that reuse existing parts of the configuration, as in the case of the *Simple Viewer*. Furthermore, if there are conflicts between variable assignments of non-existing or replaced sub-trees with variable assignments of reused sub-trees, the algorithm first changes the non-existing or replaced sub-trees as in the case of the *Remote File Access*. The combination of these two light-weight heuristics leads to an optimal result in this scenario.

4.2.4.5 Limitations

Although, the exemplary scenario demonstrates nicely that the effectiveness of the proposed value and variable ordering heuristic can be high, the overall effectiveness depends on the properties of the scenario, in general. Since both heuristics are greedy, the distance between the quality of the optimal and the determined solution can be arbitrarily high. The reason for this is the inherent discrepancy between the local cost estimates that serve as the basis for algorithmic decisions and the resulting global costs.

To clarify this, consider that even though, it is never “wrong” to reuse a component instance, different parameterizations might indirectly cause different costs but the value ordering heuristic assigns identical costs to all parameterizations of the same component. As a consequence, the resulting discrepancy between estimated and induced costs may lead to sub-optimal value selections. A simple example would be a component that supports two different parameterizations whose dependencies cannot be resolved by the same component.

Clearly, it is possible to improve the quality of the estimate by broadening its scope. However, to do this, one would have to unfold larger parts of the search space upfront and therefore this approach would conflict with the requirement on optimism. Furthermore, even if the search space would be unfolded completely, the resource constraints might lead to conflicts that have an impact on the costs. Thus, in order to compute the exact costs, one would have to compute multiple configurations, in general.

The variable ordering heuristic exhibits similar limitations. First, it is not guaranteed that changing the parts of the configuration that exhibit lower costs results in a configuration with minimum costs. As an example consider that it might be more cost efficient to replace a single expensive component instance than to replace a number of less expensive component instances. Secondly, the variable ordering does not systematically search through all possible parameterizations. As an example, consider that it might be more cost efficient to change the parameterization of an expensive component instance before replacing a less expensive one. However, since changes to a variable may only occur as result of backtracking, which must consider the variable ordering in order to remain complete, the value of a variable with lower priority will always be changed before the value of a higher priority variable. As a consequence, the heuristic might force the replacement of a component instance that is bound to a lower priority variable before it changes the parameterization of a component instance that is bound to a higher priority variable. Last but not least, the variable ordering heuristic does not create a total ordering as can be seen by the necessity of using the identifiers to break ties. However, the ordering in which ties are broken may have an impact on the resulting costs.

Again, it is possible to mitigate this, e.g. by introducing more variables to support more fine-grained control and by computing more detailed estimates which result in better orderings. However, these approaches require additional computations and thus they increase the overall effort required for search. Since a higher effort cannot be tolerated in scenarios where finding a single configuration consumes significant amounts of time, we propose an alternative strategy in the following. Although, this strategy is not perfect, it does not increase the overhead of the basic algorithm since it can be added on-demand.

4.2.4.6 Randomization

The limitations discussed previously, are a result of the greedy nature of the heuristics and the fact that the estimated local costs may not reflect the actual induced costs precisely. As a result, the overall approach is sensitive to the starting point of the computation. This means that the ordering in which different options are selected may have a significant impact on the solution quality. As a consequence, it is possible to utilize this property of the heuristics to compute multiple solutions with varying costs without changing the underlying algorithm.

To do this, we can introduce randomization at various points. For instance, in order to modify the backtracking order, we may use different schemes to compare identifiers. Alternatively, we can simply reorder the dependencies. However, a simpler way of introducing randomization is to reorder the mapping between variable assignments and contracts. If the algorithm uses a deterministic way of selecting the next option, changing this mapping induces different traversals of the search space. As we show in Chapter 6, this simple randomization strategy yields good results in many scenarios.

Using this approach to randomization, we can revise an iterative search scheme by computing multiple configurations using the previously described constructive heuristics. The basic idea is to compute randomized configurations with their resulting total cost iteratively. Thereby, we store the configuration that exhibits the least cost. Whenever a better configuration has been found, we store it as the new optimum. The iterative computation can be aborted at any point in time – after the first valid configuration has been found – or it can be continued until a sufficiently good configuration has been found.

Of course, it is possible to revise strategies that hand over the decision to the user or that perform trade-offs based on some heuristic. For instance, one may argue that if the adaptation costs of the current solution are low and the computation of the last configuration caused a comparatively high search effort, it is better to adapt the application than to increase the search effort even further. Similarly, if the last configuration has been found quite fast and the current optimal configuration exhibits high adaptation costs, it may be worthwhile to continue the search. Alternatively, one could argue that if the last configuration did not improve the quality, it might not be worthwhile to compute another one.

However, since this tradeoff boils down to a comparison of the user distraction resulting from replaced components and user distraction resulting from increased search effort, we assume that it is subjective in general. As a result, we argue that it should be controlled by the user as opposed to the system. To do this, a user may specify the aggressiveness of the optimization by means of an application preference. This preference can either be expressed as an absolute boundary for costs or it can directly select one of the strategies discussed above. From a system's point of view, the type of strategy does not affect the internal mechanisms, since it only decides whether the next configuration should be computed or whether the current configuration should be used for adaptation.

4.3 Discussion

In this chapter, we have formalized the adaptation problem as an extension to the configuration problem detailed in the previous chapter. The formalization shows that adaptation can be seen as an optimization problem on top of configuration. Based on the formalization, we identify the requirements on adaptation as an extended set of requirements on configuration in which we introduce the additional requirement on optimality. The requirement on optimality is conflicting with the requirement on efficiency. In order to account for this, we aim for a solution that imposes only minimal additional overhead on adaptation when compared to configuration.

To find a suitable solution, we discuss possible optimization techniques and we classify them on the basis of the provided guarantees with respect to the optimization result. From the set of possible techniques, we rely on a constructive heuristic that uses a local cost estimate to perform decisions. In order to gather suitable cost estimates, we propose a cost model. This model captures the costs of different adaptations in an abstract manner. The proposed cost model focuses on costs for replacing individual component instances. Such costs may be composed from different cost factors. The two main cost factors that we are considering are costs for migrating applications-specific state and costs resulting from disruptive changes to the user-experience. Since the cost model represents costs with an abstract value, other cost factors could be considered as well. By adding reasonable simplifying assumptions, it is possible to compute the costs for a concrete problem upfront. Since it is possible to reuse existing messages, the effort for applying the cost model is marginal.

To optimize a configuration, we integrate a greedy value ordering heuristic and a greedy variable ordering heuristic in the configuration algorithm. While the value ordering heuristic can be integrated in a straight-forward manner, the variable ordering heuristic requires additional precautions. Specifically, we need to split variables whose domain can consist of values with different costs. To ensure that the split variables behave properly, we introduce an additional value into the domain and we add specific constraints to ensure that value selection are mutually exclusive. Finally, in order to ensure that the variable ordering can be computed on each computer, we transmit the costs of variables during the link establishment as part of the link request. The resulting constructive optimization algorithm is very light-weight when compared to the configuration algorithm since it requires only marginal computations and it does not cause additional message transmissions. Yet, due to its greedy nature and the characteristics of the underlying problem, it cannot provide guarantees on the quality of the determined solutions. As mitigation, we propose to apply randomization during the value selection in combination with multiple executions. This approach can then be used to introduce various strategies for balancing the efforts of search and reconfiguration at runtime.

Since both overheads, i.e. the overhead for communication and the overhead for computations, can be neglected when compared to the overhead of the original configuration algorithm, the resulting approach naturally fulfills all requirements apart from optimality. In the following chapter, we first discuss how the adaptation algorithm can be integrated into the component system. Thereafter in Chapter 6, we use simulations to verify that the algorithm can fulfill the requirement on optimality in many scenarios. Furthermore, we show that the randomization can be used to balance the efforts for search and reconfiguration in scenarios in which the initial run of the constructive algorithm delivers unsatisfying results. In Chapter 7, we compare the approach for automatic adaptation with other approaches taken by existing system software. Finally in Chapter 8, we describe future research directions to extend the proposed approach on automatic adaptation.

5 Prototype

In order to evaluate the concepts and abstractions introduced in the previous chapters, we have developed a prototypical version of the PCOM component system. On the basis of this implementation, we have developed a prototypical version of the configuration algorithm described in Chapter 3. Besides from enabling automatic configuration, this algorithm also supports automatic adaptation using the extensions detailed in Chapter 4. In the following section, we first outline the architecture of this prototype. Thereafter, we present some implementation details. Due to the size of the implementation, it is not possible to discuss all details in depth. Thus, we focus on the description of the internal structure of the individual parts. Finally, we close the chapter with a discussion.

5.1 Architecture

To motivate the design decisions, we first present the architectural goals that guided our design. Thereafter, we provide an overview over the architectural building blocks and we group them into two layers. Subsequently, we describe the dependencies between the building blocks and the resulting interfaces. Finally, we describe the interaction of the building blocks by walking through an exemplary reconfiguration of an application at runtime.

5.1.1 Design Goals

As mentioned previously, the purpose of our prototypical version of PCOM is primarily the evaluation of the concepts introduced by its component model. As a result, we can derive the following two architectural design goals:

- *Completeness*: The abstractions and the mechanisms discussed previously are tightly integrated and sometimes, they depend heavily on each other. In fact, many abstractions are useless if they are not supported with appropriate mechanisms. As an example consider the different lifecycles of component instances, resource assignments and applications. They are only useful, if the components and resources developed by an application developer can actually depend on the associated guarantees. Thus, to provide a thorough evaluation of the benefits and the limitations of the component model, the abstractions and mechanisms cannot be realized and evaluated on an individual basis. As a result, the architecture must be complete and it must support all abstractions and mechanisms discussed previously.
- *Resilience*: The primary goal of PCOM is to ease the task of developing pervasive applications for smart peer groups. To achieve this goal, the component system tries to hide many complications that arise from the distributed nature of pervasive applications and the dynamic nature of the execution environment. As a consequence, the

component system itself must be highly resilient to failures. Cleary, this resilience cannot be solely achieved by the architecture, e.g. the implementation also needs to be defensive. However, an architecture that defines the interaction between its building blocks in an appropriate manner can greatly simplify the implementation. As a result, we require that the prototypical architecture should ensure that the defined building blocks can function properly in isolation and may fail independently.

Apart from solely ensuring that the concepts can be implemented, we also want to be able to determine the potential overheads introduced by them. Since the evaluation of the overheads should approximate a high-quality implementation, we can derive the following design two design goals:

- *Minimalism*: Although, PCOM can also be used in traditional networked computing environments in principle, its primary application area are smart peer groups that will frequently consist of a number of resource-poor computers. In order to support such environments, the system software must be minimal with respect to resource utilization. Thus, the underlying architecture should be simple and minimal and it should not tradeoff increased flexibility with size in cases where the flexibility is not mandatory.
- *Efficiency*: The same argument that has been made about minimalism can also be made about efficiency. In order to determine realistic estimates of the runtime overheads introduced by PCOM, the architecture should foster an efficient implementation wherever possible. This means that the architecture should not tradeoff flexibility with efficiency in cases where the gains in flexibility are not required to fulfill another design goal.

Finally, we also want to use our prototype component system to evaluate the algorithm for automatic configuration and the adaptation heuristics proposed in the previous chapters. To perform a detailed evaluation, we need to be able to measure their runtime overhead and we need to be able to compare the proposed approaches with other alternatives (Handte, Herrmann, Schiele, & Becker, 2007). Thus, we can derive the final architectural design goal:

- *Pluggable algorithms*: In order to enable the comparison of different approaches towards automatic configuration, the architecture must support the utilization of different configuration algorithms. In order to provide an unbiased basis for the evaluation, the support for different configuration algorithms must not predetermine internal details of the algorithm implementation. Specifically, it should not provide mechanisms that simplify one type of algorithm while complicating another. To do this, the architecture must refrain from specifying specialized interfaces wherever possible and it should offer alternative interfaces in cases where this is not possible.

It should be noted that the five architectural design goals are not completely orthogonal. For example, the design goal of supporting pluggable algorithms conflicts with the goal of creating a minimal architecture. Due to the research-centered nature of our prototype, this conflict cannot be avoided and regarding this point, our prototype provides a higher flexibility at the cost of a non-minimal architecture. However, in a broader context – beyond the scope of this dissertation – support for multiple configuration algorithms can also be used as the basis for adaptive approaches towards automatic configuration. Some more details on this are presented in Chapter 8.

5.1.2 Building Blocks

To simplify the description of our prototype architecture in the following, we abstract from the low-level tasks that are needed in order to enable the remote communication between the computers that are part of a smart peer group. These tasks include discovery of new computers that enter the smart peer group and the detection of computers that have left the smart peer group. It should be noted that these tasks can be performed by any communication middleware for smart peer groups. In fact, in order to implement the architecture, we have used a slightly extended version of the BASE micro-broker and BASE takes care of these issues.

Our architecture splits the responsibilities of the component system that are not directly related to low-level communication into three major functional building blocks. These building blocks are called component container, application manager and assembler. Each of these building blocks is self-contained and they can be deployed separately on the computers of a smart peer group. However, in order to use the functionality of other computers and in order to share functionality with the computers of the smart peer group, a computer must be equipped at least with a component container. Depending on the concrete implementation of the assembler it might be necessary to deploy one as well. For example, the algorithm for automatic configuration detailed in the previous two chapters requires a specific type of assembler. Since this assembler is fully distributed, it must be deployed on each computer of the smart peer group that is equipped with a component container.

As indicated by its name, the core of the architecture is formed by the component container. The component container is responsible for realizing the component and the resource abstractions of PCOM that have been introduced previously. To do this, it provides an application development framework for developers. Furthermore, it enforces the guarantees provided by the application model and it monitors the changes to the environment and signals them. The application manager implements all specific functionality that is needed to support preferences and it is responsible for managing the lifecycle of the application. In addition, it is responsible for initiating the configuration and reconfiguration of an application by selecting and initializing an assembler. The assembler is used to compute a configuration in cases where an application needs to be configured or reconfigured. To do this, the assembler may need to gather information about the available components and resources and about the existing configuration.

The decision to separate the functionality of the assembler from the functionality of the component container is a result of the design goal to simultaneously support alternative approaches towards automatic configuration. The reason for separating the functionality of the application manager from the functionality of the component container is twofold. First and foremost, it simplifies the tasks of the component container significantly by alleviating the need of providing specialized functionality for the root of an application. In fact, in the proposed architecture, the component container is exclusively realizing the concepts of components and resources that are part of a tree and thus, virtually all functionality can be realized through some form of recursion. Secondly, it can also reduce the memory requirements of PCOM since only those computers that are starting applications on behalf of a user need to be equipped with an application manager. This helps in fulfilling the design goal of minimalism.

5.1.3 Layers

In order to explain the dependencies between the individual building blocks, it is necessary to show the overarching concepts of how a configuration and reconfiguration is performed. As shown in Figure 29, the key idea of our prototype architecture is to differentiate explicitly between the execution layer and the configuration layer. The execution layer is formed by the application manager and the component container. The configuration layer is formed by the assembler. To decouple the execution layer from the configuration layer, the architecture uses models that abstract from the internal details of the layers. These models contain sufficient details to configure an application and to execute a configuration.

In principle, there are two different ways how these models could be exchanged between the assembler and the component container. First, they could be exchanged unconditionally and completely and secondly, they could be exchanged partially in an on-demand fashion. Furthermore, there are two possible options for controlling the data exchange, i.e. either the execution layer or the configuration layer could be responsible for controlling it. In order support the efficient implementation of the architectural building blocks, we combine these approaches depending on the purpose and type of the model.

For efficiency reasons, we only exchange those models that are always needed completely and upfront. The other models are retrieved partially and on-demand. Besides from supporting the design goal of efficiency, this also helps in creating a fair basis for the comparison of different approaches towards automatic configuration as they may restrict themselves to retrieving only those parts of the models that are actually required. As result, the models that contain configuration related information are always exchanged upfront and completely and the models that contain environmental information, e.g. the contracts supported by a component instance, etc., are exchanged partially and on-demand.

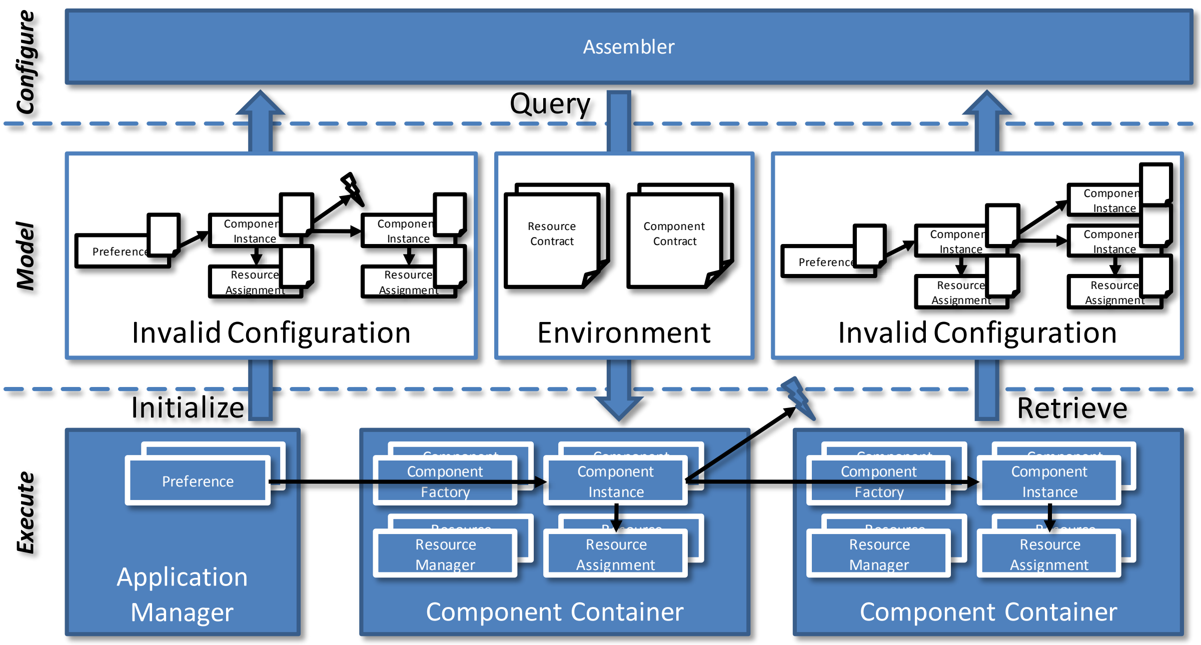


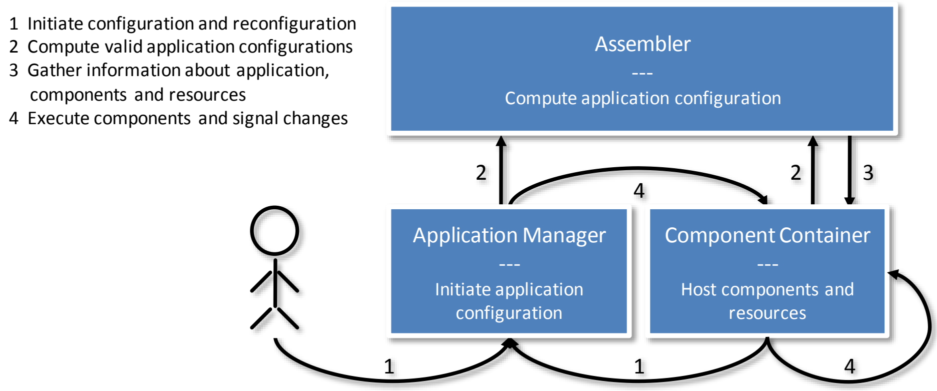
Figure 29 – Architectural Layers

In order to support a resilient implementation, we ensure that the application manager and the component container remain in control over all interactions with the assembler. This allows them to take corrective actions in cases where the assembler becomes unavailable. Thus, the exchange of models that contain configuration information is always initiated and controlled by the execution layer. The exchange of environmental information is controlled by the assembler in order to allow arbitrary retrieval strategies.

In addition, the component container and the application manager are also validating all models retrieved from the assembler in a defensive manner before they are using them to adapt the executed configuration. Clearly, at a first glance, such defensive validations might seem to introduce additional overhead and thus, they might seem to contradict the goal of an efficient implementation. However, this approach ensures that the guarantees defined by the component model can be enforced without relying on the proper functioning of an assembler and this, in turn, greatly simplifies the task and the implementation of the assembler.

5.1.4 Dependencies

Figure 30 depicts the resulting responsibilities of the three building blocks and it shows their interdependencies. The application manager hosts the preferences and ultimately controls the application lifecycle upon user request (1). It relies on the assembler to compute a configuration for the application (2) as soon as it needs to be started. To do this, the application manager and the component container provide the appropriate information about any existing configuration. In addition, the assembler requests additional information about the environment from the component container (3).


1 Initiate configuration and reconfiguration
2 Compute valid application configurations
3 Gather information about application, components and resources
4 Execute components and signal changes
Assembler

Compute application configuration
Application Manager

Initiate application configuration
Component Container

Host components and resources
1
2
3
4

Figure 30 – Architectural Dependencies

After a configuration has been computed, the application manager relies on the component container to host the application anchor (4). Besides from hosting the application anchor, the component container also takes the responsibility to signal changes to application back to the application manager. In order to fulfill this responsibility, the component container depends on other component containers that host components and resources recursively. If an application configuration can no longer be executed, the component container depends on the application manager to initiate a reconfiguration of the application (1). To do this, the application manager needs to interpret the signals received from the component container.

5.1.5 Component Container

As indicated in the architectural overview, the component container is responsible for hosting components and resources. According to the dependencies of other building blocks on the component container, its interfaces can be classified into three categories. These categories are best explained by looking at the roles of the component container using a fragment of an application as shown in Figure 31.

The fragment of the application consists of three component instances *A*, *B*, and *C*. In its contractual demand, component instance A declares a dependency (D_B) that is fulfilled by the contractual provision (P_B) of component instance *B*. Component instance *B*, in turn, contractually demands a component (D_C) of type *C* which can be matched by the provision (P_C) of component instance *C*. For the sake of simplicity, we assume that each component instance is hosted in a different component container called *A*, *B*, and *C* and we look at the application from the perspective of the component container that hosts the component instance *B*.

With respect to the component instance *A*, the component container *B* acts in the role of a *Provider*, i.e. it provides a component instance with a certain provision for the application. This means that it needs to be informed about transitions to the lifecycle of component instance *A*. If component instance *A* is about to be started, stopped or paused the component instance *B* should be notified accordingly by the component container *B*. It is noteworthy that conceptually, the component container *B* also needs to be informed if the component container *A* is no longer part of the same smart peer group. However, due to obvious reasons such a notification cannot be performed explicitly by component container *A*. As a result, monitoring of the availability of component container *A* becomes a conceptual part of the interface required to fulfill the role of a providing component container.

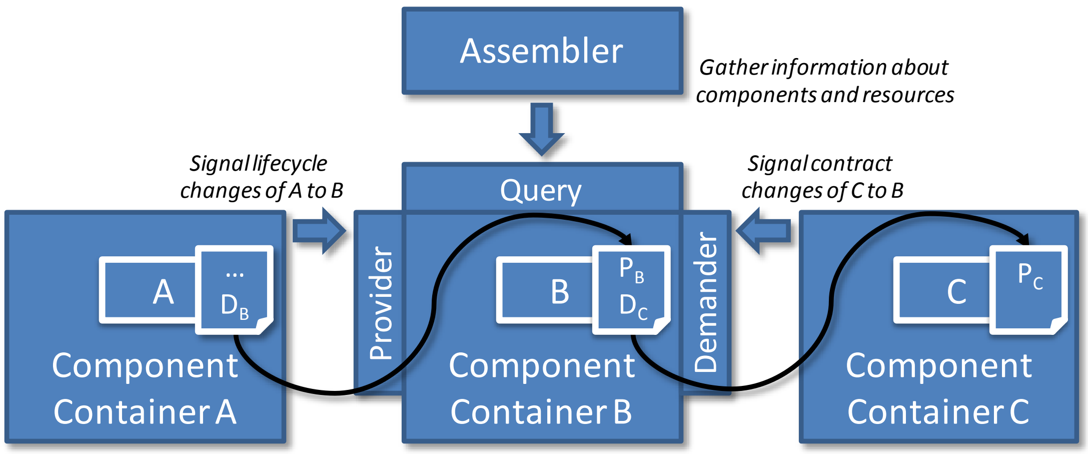

Assembler
Gather information about components and resources
Signal lifecycle changes of A to B
Query
Signal contract changes of C to B
A
…
D_B
Component Container A
Provider
B
P_B
D_C
Component Container B
Demander
C
P_C
Component Container C

Figure 31 – Component Container Interfaces

With respect to the component container *C*, the component container *B* acts in the role of *Demander*, i.e. it demands a component instance with a certain provision. This means that it needs to be informed about changes to the provision of the component instance *B*. Such changes can either be the result of a failure induced by mobility or an unforeseeable change in resource availability. In addition, component container *B* also needs to be informed if component container *C* becomes unavailable. In analogy to the role of a providing component container, the unavailability of component container *C* cannot be signaled explicitly but must be detected by component container *B*. Thus, conceptually the detection of the unavailability of the providing container is also part of this interface of the demanding component container.

As third and final role, component container *B* also acts as a data repository for the assembler, i.e. it needs to enable the assembler to retrieve relevant internal information in cases where the assembler needs to configure or adapt an application. To do this, the component container enables the assembler to query for possible resource and component contracts using a component or resource demand. In order to minimize the number of interactions between the assembler and the component container, the interface also support batch queries for sets of components and resource contracts. Additionally, the assembler may also query for the available amount of resources in order to determine resource conflicts.

5.1.6 Application Manager

The application manager is responsible for storing the user preferences and for managing the overall lifecycle of an application. Thus, the application manager needs to support two types of interfaces. These interfaces are depicted in Figure 32.

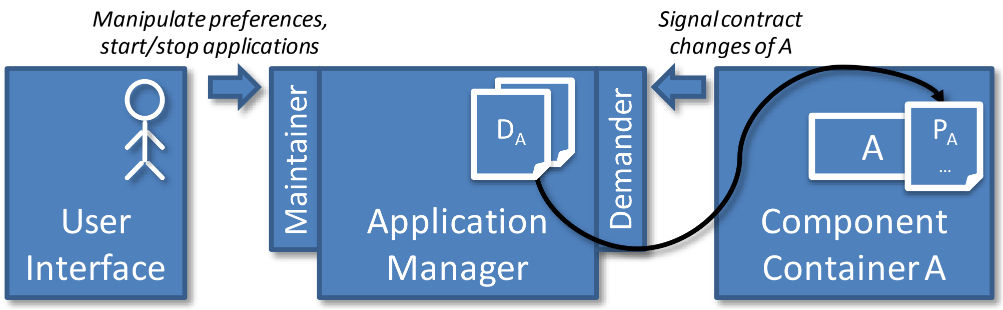


Figure 32 – Application Manager Interfaces

The interface depicted on the left side enables the user to create, modify and remove preferences and that enables the user to start and stop applications using the stored preferences. It is noteworthy, that this interface is the only interface that is geared at supporting interaction with the user. All other interfaces are solely used internally. However, it should be clear that this interface is not accessed directly by a user. Instead, it is usually used indirectly through some user interface. As briefly discussed in the next section, we have implemented a graphical user interface for this purpose.

In contrast to the first interface which is supporting interaction with the user, the second interface is needed during the execution of applications. To motivate the design of this interface, it is noteworthy to point out that there is no conceptual difference between preferences and demands during the execution of an application. Thus, the component container implementation can be simplified by hiding the distinction between the application manager and the component container interface. Since a preference can be interpreted as a demand on a component, we can do this by reusing the demander interface of the container.

5.1.7 Assembler

The assembler is responsible for computing configurations during the initial configuration and during adaptation. As shown in Figure 33, the assembler provides three interfaces for this task. It provides a control interface that can be used to start and stop the configuration of an application. In addition, it provides interfaces to initialize the assembler with an existing configuration and to retrieve a successfully computed configuration from the assembler.

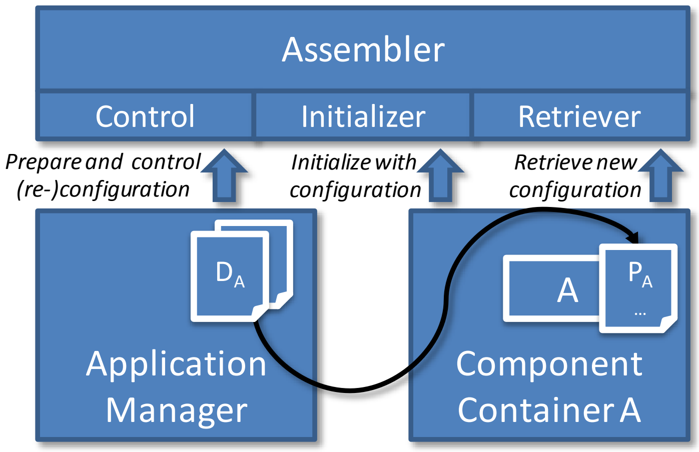

Assembler
Control
Initializer
Retriever
Prepare and control (re-)configuration
Initialize with configuration
Retrieve new configuration
D_A
Application Manager
A
P_A
...
Component Container A

Figure 33 – Assembler Interfaces

The details of these interfaces are explained best by looking at the dynamic cooperation of the component container, the application manager and the assembler. Since we describe the interaction of these architectural components in following in Section 5.1.9, we refer to this section for more details.

5.1.8 Signaling

As indicated previously, the application manager is responsible for managing the overall lifecycle of an application. In order to do that, the application manager must be notified by the containers as soon as the configuration is no longer valid. Since each component containers only knows the component containers that provide or use some component instance, the signaling must be done recursively.

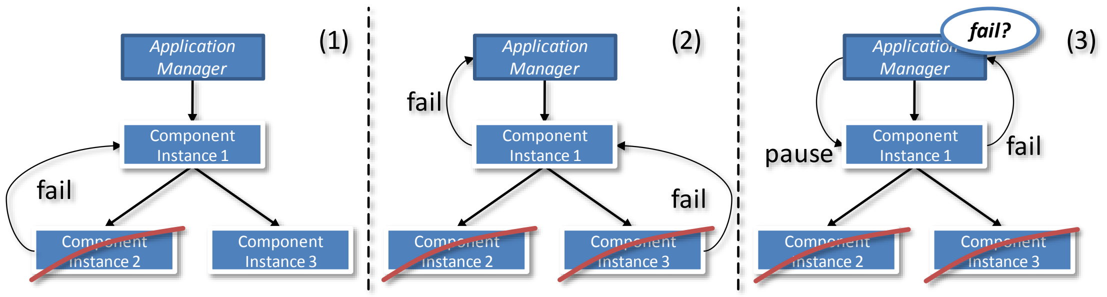


Figure 34 – Ambiguous Signals due to Parallelism

Since applications are usually distributed, changes in different parts of the application might be detected a signaled in parallel. Without further precautions, the resulting parallel signals may result in unnecessary overhead. Figure 34 shows an example for this. The application consists of three component instances. After component instance *2* becomes unavailable, component instance *1* detects this and signals the failure to its parent, the application manager (1). Shortly after the change has been signaled, the unavailability of component instance *3* is detected (2). In response to the detected failure, the application manager signals a transition in the application lifecycle. At the same time, component instance *1* signals the unavailability of component instance *3* to the application manager (3). Due to the fact that the application manager does not know whether the failure signaled by component instance *1* has happened before or after the transition of the lifecycle, it needs to treat the failure. However, in this particular case, treating the failure is not necessary. In extreme cases, e.g. if messages are delayed for long periods or if a irrelevant failure is signaled while switching from one configuration to another, this problem might not only cause duplicate transitions in the application lifecycle, but it might also lead to duplicate adaptations. Since the effort for performing an adaptation is comparatively high, it is necessary to avoid such cases.

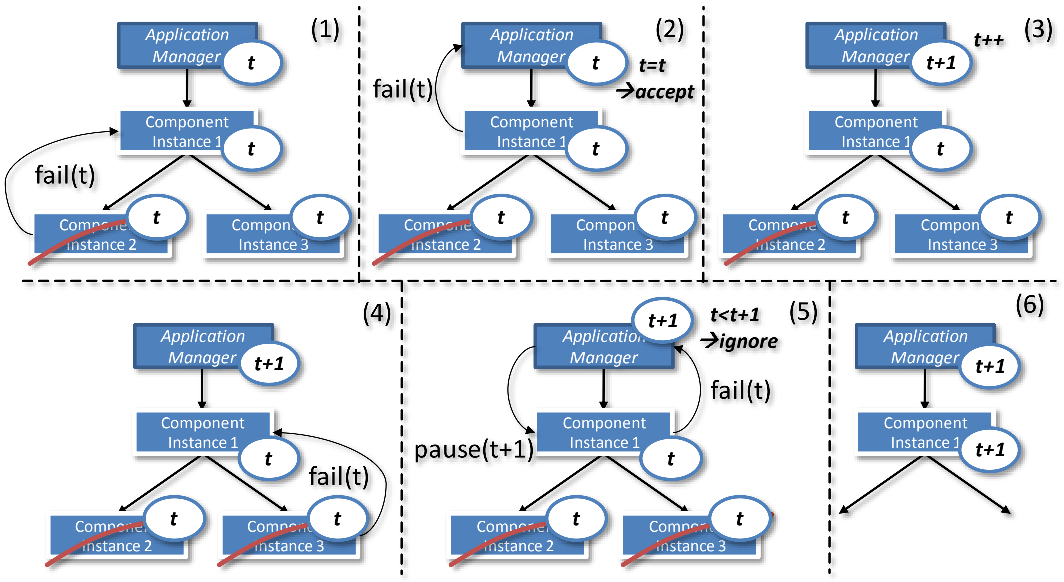


Figure 35 – Disambiguation with Logical Timestamps

To do this, we introduce a logical timestamps to capture the causal relation between transitions in the application lifecycle and the detection of failures. Conceptually, the logical phases are similar to logical clocks (Lamport, 1978). However, instead of capturing the causal relationship between all events, we limit the scope to the relevant subset. To realize this, we keep an integer timestamp for each component. The timestamp is increased for every lifecycle transition. When a failure is detected, the current timestamp is included in the message. By comparing the local timestamp with the timestamp contained in the message, the receiving computer can detect whether the failure has been detected before or after the last transition.

A concrete example is depicted in Figure 34. Again, the application consists of three components from which two will fail. When the application has been started completely and successfully, the timestamp for each component is equal and shows the value *t*. After the first failure is detected by component instance *1* (1), it is signaled to the application manager (2). As described earlier, the signal contains the timestamp associated with component instance *1*. Since the timestamp of the application manager shows the same value, it accepts the message and initiates a lifecycle transition. Thereby, it increases its local timestamp (3). At the same point in time, the unavailability of component instance *3* is detected (4) and signaled using another message to the application manager (5). When the second failure signal arrives, the application manager finds that the timestamp contained in the message has been created before the transition has been signaled. Thus, it can simply drop the message as the initiated lifecycle transition will also handle this problem. When component instance *1* eventually receives and processes the lifecycle transition, it can adjust its local timestamp according to the value contained in the message (6).

Note that although it might seem possible to avoid the transmission of the current timestamp as part of the lifecycle transition messages by incrementing the timestamp, this approach will not work in general. The reason for this is that adaptation can introduce new component instances that need to be initialized with the current timestamp. This can either be done by performing a special initialization during adaptation or by transmitting the timestamp as part of the transition, as shown in the example.

5.1.9 Interaction

In the following, we describe the dynamic cooperation of the three building blocks by walking through the complete adaptation process of an application. To simplify the description, we assume that the assembler implementation is fully distributed. This means that each computer is not only equipped with a component container but also with an assembler that computes the parts of the configuration that are relevant for the computer that hosts it.

Figure 36 depicts the sequence of interactions required to initialize the reconfiguration process for an exemplary application consisting of four components (*A*, *B*, *C*, *D*) running on four computers (*1*, *2*, *3*, *4*) of a smart peer group that needs to be adapted because computer *4* is no longer reachable.

When a component instance used by the application is no longer available the container that hosts the parent component instance detects this and sends a message to the application manager that started the application (1). As explained earlier, this is done by signaling the change recursively on the path to the parent through the demand interface of the component container until the signal arrives at the application manager.

The application manager then selects the assembler that shall compute the configuration and calls a prepare method through the control interface of the assembler (2). In response, the assembler can prepare its internal data structures and if it requires other assembler instances, it can initialize them. Since this example assumes that the assembler is fully distributed, the assembler implementation would usually use the control interface of all other assemblers in the smart peer group to initialize them as well.

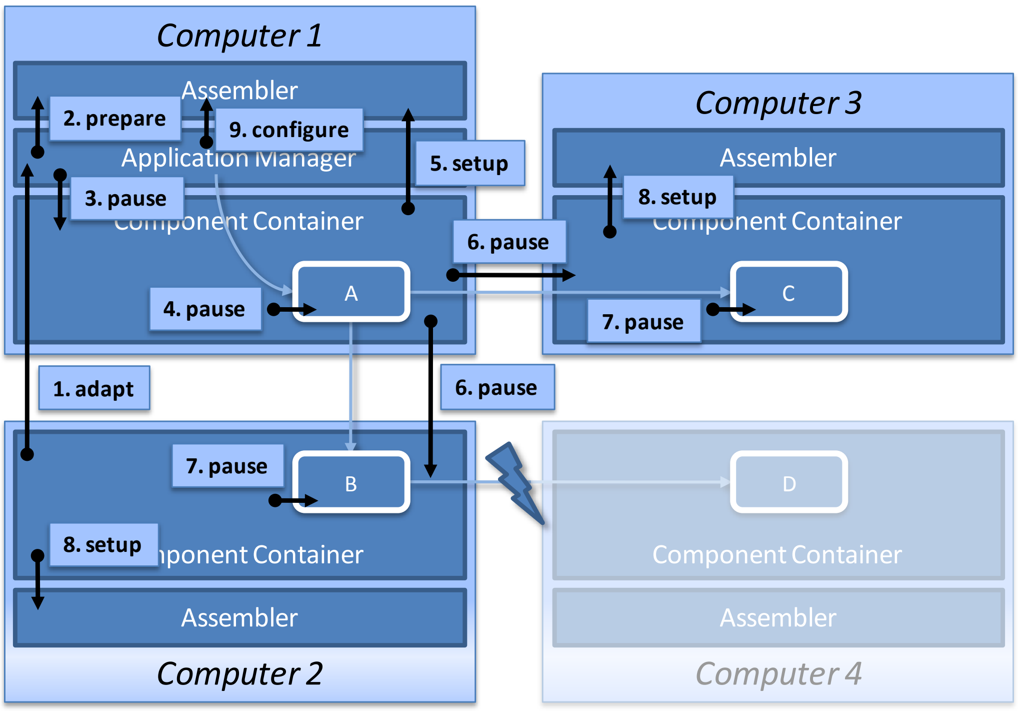


Figure 36 – Initializing an Adaptation

After the prepare call returns, the application manager signals the container to pause the anchor (3). This transition in the lifecycle is sent through the provider interface of the container. Thereby, the application manager passes the reference of the assembler to the container. The container will then send a pause signal to the anchor (4) since component instances and resource assignments are paused and stopped in a top down fashion. In addition, the container will prepare a setup object that describes the anchor, its contractually specified dependencies, and the remote systems that host child components for these dependencies. This setup object essentially represents the model that is used to provide the assembler with sufficient information about the existing configuration to compute a new configuration.

The setup object is sent to the assembler represented by the reference and the assembler returns a reference to an assembler for each component declared in the setup object (5). Since the references are issued by the assemblers, neither the component container nor the application manager must be aware of the distribution degree of the assembler. Thus, the initialization interface of the assembler is also used to hide the internal details of the assembler implementation that are related to distribution.

Thereafter, the component container sends the pause request to all containers that host children of the anchor. As part of the call, the component container passes the reference of the corresponding assembler to the container (6). To do this the component container uses the provider interfaces of the component containers. The contacted component containers will continue with this process recursively until the complete configuration has been put into the assembler and all component instances have been paused (7, 8).

As soon as the initial pause call returns, the assemblers have a complete view of the configuration. Thus, they can start to compute a new configuration. However, since the configuration is pushed into the assemblers, they need to be informed about the fact that the pause call is complete. To do this, the application manager sends a configure call to the initial assembler (9). Again, this is done through the control interface of the assembler.

In response, the assembler will prepare a valid configuration. Since the assembler has received a setup object for each component instance, it can determine which dependencies need to be resolved in order to transform the current invalid configuration into a valid one. To compute a valid configuration, the assembler needs to be able to determine the set of resources that is available on each computer and it needs to be able to find the components that can be used to resolve a dependency. To this end, the assembler uses the query interface of the component containers that are part of the smart peer group.

As soon as the assembler has computed a valid configuration, it returns the configuration to the application manager as return value of the configure call. In order to provide efficient support for different distribution degrees, the result must not necessarily contain the complete configuration. Instead, the data structure used to describe the tree of components can either contain the full configuration data or a reference to the assembler that can provide the configuration data for a certain sub-tree upon request. Using this data structure, the configuration data can be retrieved lazily.

Figure 37 shows the sequence of actions for the previous example required to transform the running invalid configuration into a valid configuration. To do this, the component *B* running on computer *3* is replaced with component *B'* on computer *4*. Note that the example assumes that the configuration is stored in each assembler and is returned lazy upon request as explained previously.

After receiving the configuration, the application manager sends a start request to the anchor (1). Thereby, it passes the configuration data received by the assembler. Since this is a transition of the lifecycle of a component instance, this call is done through the provider interface of the component container.

The container retrieves the configuration for each child component required by the anchor (2). Towards this end, the container places calls to retriever interface of the corresponding assemblers. Note that the component container does not immediately start the anchor component as a result of the bottom-up startup of applications.

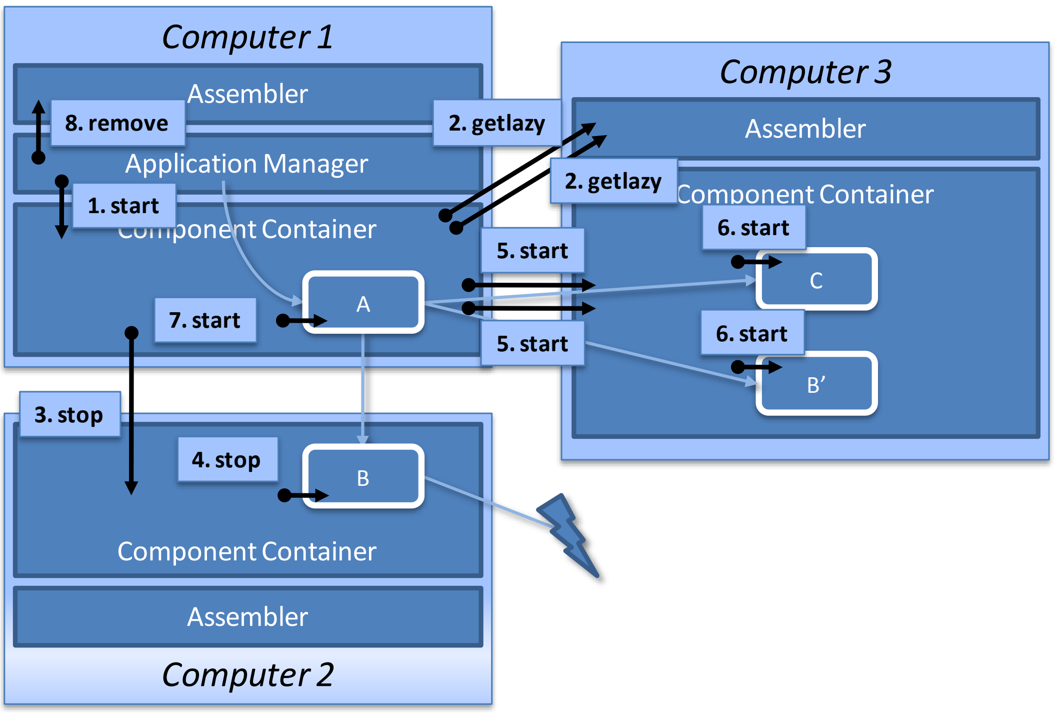


Figure 37 – Performing an Adaptation

Using the retrieved configuration, the component container decides whether the child must be reused or replaced. If the child must be replaced, the container first stores the internal state of the component instance (3) and releases it by sending a stop call (4). The internal state can later on be used to restore the state of the component instance in another component instance on another component container.

Thereafter, the container will send a start call to the container that hosts the new component (5). The start call then contains the state of the stopped component. If the component is reused, the container simply sends a start call to the container of the reused component. As part of the start call, the container sends the configuration for the child and the recursion continues through the provider interface of the component containers.

The return value of a start call signals whether a child has been started successfully. If all start calls for all children of a certain component have returned successfully, the component itself is started (6) and all state is restored (7). After the component and its children have been started recursively, the start call returns the status. If this procedure fails at any point in time, e.g. because a computer is no longer available, the start calls will simply return that the startup is not successful. If that happens, the application manager can then restart the complete adaptation process which pauses the components that have been started.

When the start call returns successfully, the application manager sends a remove call to the control interface of the assembler (8). This allows the assembler to remove all data stored for the application. If the assembler used instances on other computers, it can forward the remove call to release their data, too.

5.2 Implementation

In order to evaluate the concepts introduced by the PCOM component system as well as the architecture presented in the previous section, we have implemented PCOM on top of BASE, a service-oriented communication middleware for smart peer groups. In addition to developing the core elements of the overall architecture described previously, we extended the functionality of BASE in order to enable the reliable monitoring of remote objects and in order to allow an efficient implementation of the algorithms. Furthermore, we developed a set of optional graphical user interfaces as well as a development tool for PCOM components. In the following, we briefly outline BASE as well as the extension that we made to monitor remote objects. Thereafter, we discuss some implementation details of the component container, the assembler and the application manager. Finally, we provide a brief overview of the graphical user interfaces for the component system and the development tools for PCOM components.

5.2.1 Communication Middleware

As presented in (Schiele, 2007), BASE is a service-oriented communication middleware for smart peer groups that is based on the idea of a minimal yet extensible core. The functionality provided by this minimal core is the remote and local mediation of requests between application objects. To support different communication technologies and protocols, the BASE micro-broker can be flexibly extended with various communication plugins.

The details of this are not relevant for the remainder and thus, we refer the interested reader to (Schiele, 2007) for an in-depth discussion of the associated concepts. However, from the perspective of the PCOM component system, it is important to mention that BASE utilizes four components to mediate requests between application objects. These components are called InvocationBroker, PluginManager, DeviceRegistry and ObjectRegistry. The interaction between these components is depicted in Figure 38 and detailed in the following.

The InvocationBroker is responsible for accepting requests from application objects that need to be mediated. In order to represent such requests, BASE introduces a unifying abstraction called Invocation. Invocations that need to be delivered to a local application object are directly handled by the InvocationBroker. Invocations that need to be transferred to some remote computer are passed to the PluginManager.

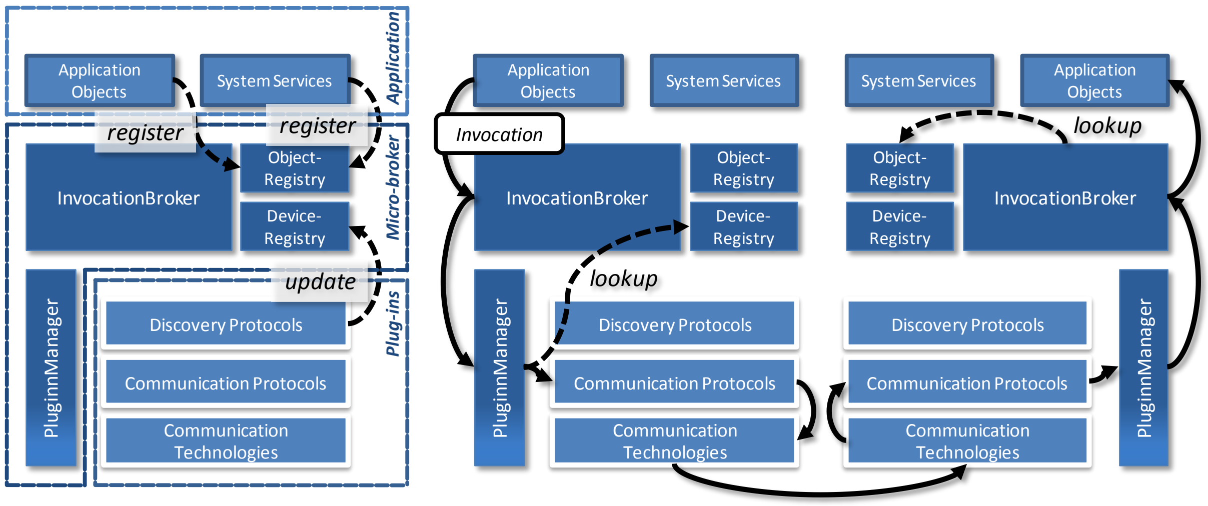


Figure 38 – BASE Architecture and Interaction

The PluginManager then takes care of transmitting the Invocation to the remote computer. Since the PluginManager can support different communication technologies and protocols, the PluginManager first needs to select a suitable communication technology and a suitable set of communication protocols depending on the capabilities of the local and the remote system. In order to make this selection, the PluginManager relies on the information held in the DeviceRegistry.

The DeviceRegistry is responsible for storing information about the computers that are currently part of the smart peer group. This information entails the set of plug-ins that are available on each computer as well as plug-in-related information that is needed to initiate the communication. In addition, the DeviceRegistry also stores information about the system services that are available on each computer. All information held by the DeviceRegistry is proactively distributed within the smart peer group by means of discovery protocols.

When an Invocation arrives at the PluginManager of the computer that contains the target object, the PluginManager passes the Invocation to the local InvocationBroker. The InvocationBroker then delivers the Invocation to the targeted application object. In order to do this, each potential target object is registered at the ObjectRegistry using a globally unique identifier. This globally unique identifier is contained in the Invocation and thus, the InvocationBroker can select the right target object and pass the Invocation to it.

It is important to mention that the ObjectRegistry distinguishes two possible types of identifiers, namely well-known identifiers and automatically generated identifiers. Well-known identifiers are used to identify system services. Automatically generated identifiers are used to identify other application objects. The main difference between the two types is that well-known identifiers are distributed proactively by means of discovery protocols and they are stored within the DeviceRegistry of each computer in the smart peer group. Thus, each computer can determine whether some remote system that is part of the smart peer group is equipped with a certain system service by locally inspecting the well-known identifiers contained in the DeviceRegistry.

BASE uses this proactive distribution of well-known identifiers to bootstrap further application support. The original system described in (Schiele, 2007) uses this for instance to implement a ServiceRegistry for user-defined services. For PCOM, we use this interface to develop the architectural building blocks as well as the BASE extension described next. This allows the seamless coexistence of computers that are equipped with the PCOM extensions with other computers that are solely executing the core functionality of BASE. Moreover, it can also support the utilization of existing application objects that have been developed with BASE by providing appropriate wrappers that interface with the PCOM container either as components or resources.

5.2.2 Lease Registry

The core functionality of BASE, as described in (Schiele, 2007) does not provide strong guarantees on the information stored in the DeviceRegistry. The reason for this is that the DeviceRegistry does not explicitly regulate how the discovery plugins need to update the stored information in order to allow the implementation of different device discovery protocols. This leads to the problem that BASE does not guarantee that temporary disconnections are recognized by all group members. The resulting asymmetry of information renders the DeviceRegistry useless in order to monitor the availability of a certain remote object.

In order to understand the implications of this problem, consider the following example in which a client and a server are solely relying on notifications of their local DeviceRegistries to manage client-specific session data stored at the server. If the client and the server are temporarily disconnected, there are four possible outcomes. If the period of time in which client and server are disconnected is very short, neither the client nor the server might recognize the disconnection. In this case, nothing needs to be done. In the second case, both DeviceRegistries detect the disconnection and signal this to the client and the server respectively. In this case, the client and the server both know that the client-specific session data has been removed and the client can perform corrective actions, e.g. it can reinitialize the client-specific session data. In the third and the fourth case, either the client or the server detects the disconnection. The former might lead to stale state on the server, while the latter can break the client implementation when it continues to interact with the server.

In order to mitigate this problem without changing the semantics of the DeviceRegistry and existing discovery plugins, we extend the functionality of BASE with a generic and configurable monitoring mechanism for remote objects. However, it should be noted that the overall system model of a smart peer group corresponds to that of an asynchronous system. Thus, the reliable monitoring of remote objects can only be approximated in general and the premature removal of state cannot be prevented in every case. Yet, for the requirements of the PCOM it is sufficient to reliably detect the cases where the state has been removed prematurely and to provide a solution that avoids the premature removal on a best effort basis.

We have implemented this mechanism as a BASE system service that we named the LeaseRegistry to underline the similarity with leases (Gray & Cheriton, 1989). The LeaseRegistry introduces periodic communication between the computer that hosts a remotely used object and the computer that relies on the presence of the object. The periodicity can be flexibly configured for each object. To hide the details of the underlying communication, the object is associated with a lease. The lease is created when the remote object is registered at the LeaseRegistry of its hosting computer. The lease can then be passed to one or more interested clients. The clients need to register the lease at their local LeaseRegistry. Using the information contained in the lease, the participating LeaseRegistries can cooperatively ensure that both, the clients and the computer that hosts the remote object, are notified when either one is no longer available. To do this, the client LeaseRegistries periodically extend the lease. If the hosting computer has already released the object, the lease extension will fail and the clients know that the object has been removed. If the hosting computer does not receive a lease extension in a certain amount of time, the object can be removed. If the removal was premature, all remaining clients will be notified upon their next lease extension attempt.

5.2.3 Streaming Semantic

The flexibility to switch between different communication technologies and protocols introduces additional overhead. In order to support temporary disconnections on different technologies, BASE composes a suitable communication stack for each remote interaction. Usually, the overhead for doing this can be neglected when compared to the delay resulting from remote communication. However, the configuration algorithm and the adaptation heuristics described in Chapter 3 and 4 are typically transmitting a high number of comparatively small messages (usually few bytes to represent variable identifiers and value assignments). Furthermore, the algorithm assumes that the messages that are exchanged by two computers are received and processed in first-in-first-out order. As a result, the utilization of a synchronous remote method call to transmit a large number of small messages sequentially would result in very high configuration delays. Of course, it is possible to use sequence numbers to enable the parallel message transmission while maintaining the first-in-first-out order. Yet, this approach still suffers from unnecessary overhead for composing protocol stacks over and over again.

To avoid this overhead, we have developed a high-level BASE communication plugin that provides remote objects with direct access to the connection-oriented transports provided by the communication plugins on lower levels. This way, we can use the first-in-first-out guarantees provided by connections and we can reuse the same communication stack for multiple transmissions. According to the terminology introduced by BASE, the plugin is situated on the semantic layer and thus, it defines the communication semantic in terms of synchronization, retransmissions, etc. We use the flexibility of this layer to introduce a special type of semantic that simply opens a connection to another service using the standard facilities of the PluginManager. After the connection has been established, it passes on the Connector that represents the connection to both, the source and the target object. In order to do that in a controlled manner, we introduce a streaming interface that needs to be implemented by objects that support connection-oriented communication and we modify the proxy and skeleton generator of BASE so that it can generate special proxies and skeletons for objects that implement this particular interface.

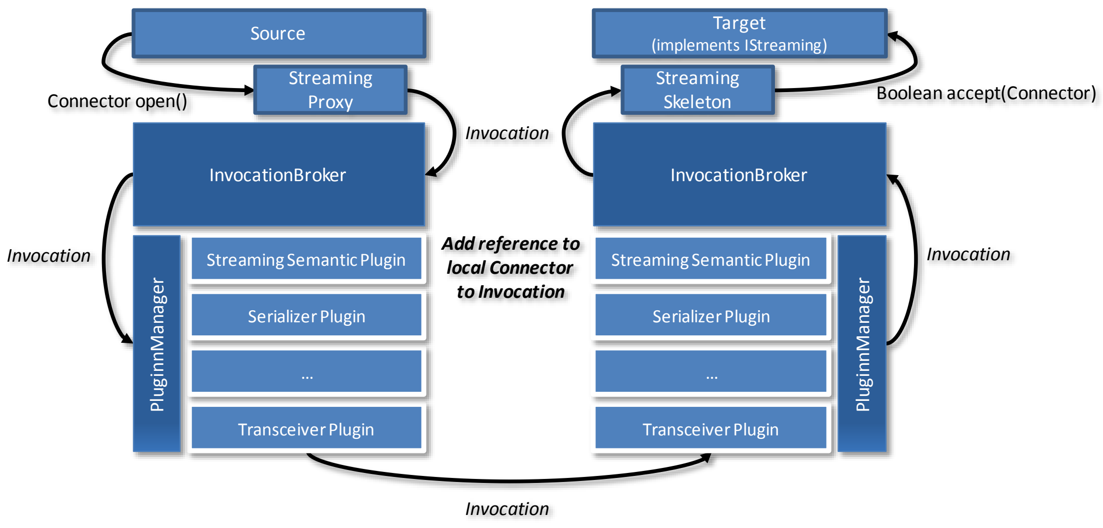


Figure 39 – Streaming Plugin

Figure 39 shows the resulting interaction between a source and a target object. The target object implements the streaming interface which defines an accept method. When the proxy and skeleton generator detects this interface, it generates a special proxy that implements an open method. By calling this method, the source object can get access to a Connector. To create this connector, the proxy sends an invocation to the InvocationBroker that is eventually delivered to the streaming semantic plugin. The plugin then creates a connection that with the remote object using the standard facilities of the PluginManager. When the connection has been established, the streaming plugin puts the corresponding Connector in an invocation and returns it to the proxy which passes it to the source. On the target system, the streaming semantic does the same. The skeleton then dispatches the connection to the target object which receives the connection through the accept method. After the connection has been established successfully, the source and the target object may use it to transmit an arbitrary sequence of bytes.

Since the streaming plugin allows direct access to the underlying connections and protocols, it can be used to avoid the repeated composition of communication stacks. However, the drawback of this approach is that many of the advantages of the BASE plugin architecture are bypassed as well. For instance, the semantic puts the burden of dealing with disconnections on the application developer. Furthermore, the application developer needs to ensure that the connections are released properly in order to avoid unnecessary resource consumption. As a consequence, the advantages and disadvantages implied by the streaming semantic should be considered carefully. However, since we use the streaming semantic to efficiently implement the communication between different assemblers, the advantages clearly outweigh the potential drawbacks in this case.

5.2.4 Component Container

As introduced in the architectural overview, the component container is the core building block of PCOM. Besides from exposing the interfaces required by the application manager and the assembler, the component container is responsible for implementing the component and the resource abstraction. In order to realize these abstractions, the component container provides a framework for developers. The framework consists of three main parts.

The first part is the contract object model which implements the component and resource contracts described in Section 2.3.2. In order to reduce the memory footprint of the contract object model, the implementation is condensed into a single class whose instances can be organized into a tree-structured object graph. Instead of using different classes in order to represent the individual sections of the contracts, the contract model uses a specialized type system that is build using a predefined set of constants and built-in mapping tables. These tables define valid compositions and ensure that the sections of the contracts appear in the right order at the right place. However, since this approach results in a rather unintuitive programming interface, we decided to abstract from this type system using a set of interfaces together with an adapter class. Besides a small memory footprint, this implementation also allows us to restrict the access to different parts of a contract depending on its usage. Towards this end, the implementation conceptually distinguishes between three different classes of contracts, which we called setup, template and status. The setup class is used during the creation of a contract through the component factory or the resource manager and thus, it allows full read and write access to all parts of the contract. The template class is used to represent the contract of a component instance or a resource assignment at runtime and thus, only the provision specified by the contract can be changed. Finally, the status class is used to represent the dynamic provision of other component instances and resource assignments that are used by some other component instance and thus, they cannot be changed.

The second part is a set of interfaces that define the callbacks of component factories, component instances and resource managers. Note that there is no interface for resource assignments since these are generic objects that are implemented by the component container. Application developers can create new components and resources by implementing the corresponding interfaces. In addition to providing its container-specific interface, a component instance will usually support further application-specific interfaces. These application-specific interfaces can be designed freely, but they must be compatible with the interfaces defined by the framework, i.e. the methods declared in the application-specific interfaces must not contain methods that have the same or a conflicting signature. The same holds true for the handle to a resource that may be contained in a resource assignment. There, the handle will usually expose an application-specific interface. However, since the handle is not accessed by the component container, it can expose an arbitrary interface.

The third and final part is a set of interfaces that enables component factories, component instances and resource managers to retrieve and to manipulate relevant information provided by their component container at runtime. Towards this end, the container provides component factories, component instances and resource managers with references to their so-called context objects. Each context object stores all information that is needed by a single component factory, component instance or resource manager. The exact information and functionality that is made accessible through the context object depends on the type of the application element. In addition, the context object also manages all information required to execute the corresponding application element.

Due to this design, the container itself is solely responsible for dispatching incoming calls from the architectural building blocks of PCOM to the corresponding context object. To perform this dispatch, the container assigns a globally unique identifier to all application elements whenever they are installed or created and it stores the mapping between identifiers and application elements in a component table and a resources table. An overview over the resulting internal structure of the component container and the context objects is shown in Figure 40 .

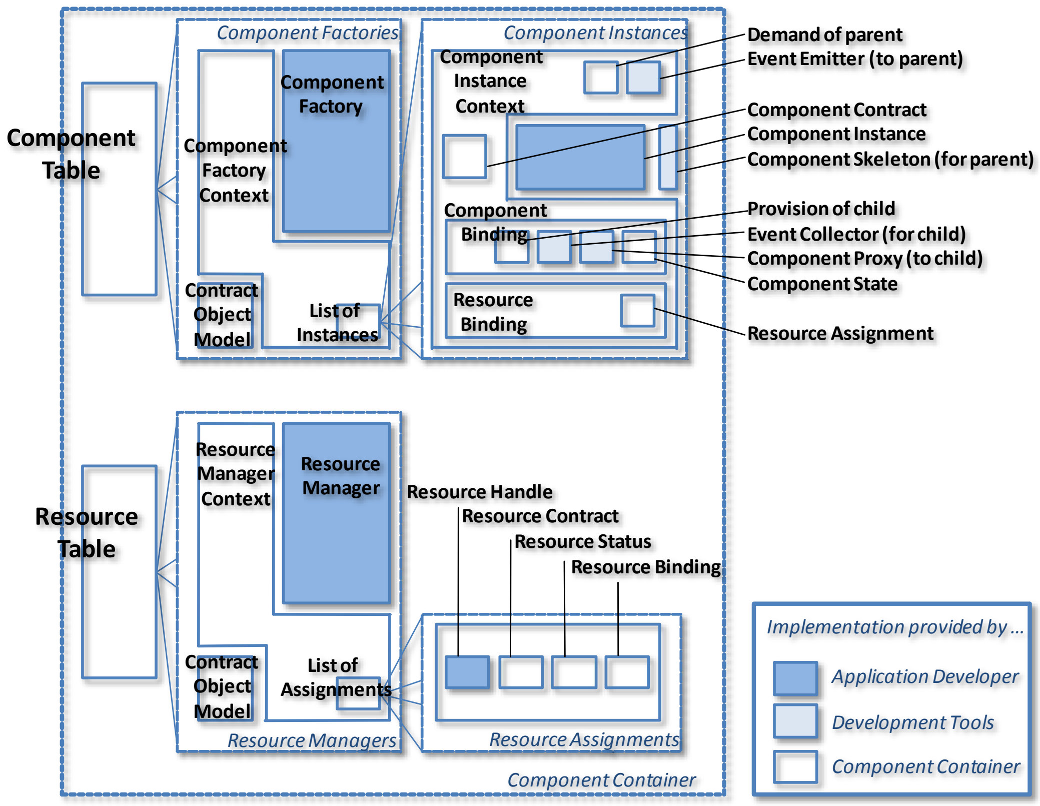


Figure 40 – Component Container Implementation

The component table contains the mapping between the context objects of the component factories and their globally unique identifier. The context object of a component factory contains references to the factory in order to query for potential component contracts during the configuration of an application and to instantiate component instances if they are needed. In addition, it also maintains a table of the context objects of component instances that have been created by the corresponding component factory. Apart from enabling the dispatch of incoming calls to the right context object of a component instance, this table is also required to support the clean removal of a component factory together with all of its component instances from the component container. The component factory can use its context object to create new candidate component contracts during configuration.

The context object of the component instances is more complicated than that. Analogous to the context object of component factories, it also needs to maintain a reference to the component instance. This reference is used to trigger changes to the lifecycle of the component instance. In order to enable the remote communication between different component instances, each context object of a component instance also needs to manage an appropriate set of proxies and skeletons. Since a component instance may expose two types of interfaces, i.e. one interface for transmitting events to its parent instance and one interface for receiving calls from its parents, each component is equipped with a so-called event emitter to send events and a skeleton to dispatch the remote calls of the parent.

In addition, each component instance may exhibit an arbitrary number of dependencies on other components and resources. To represent these dependencies, the context object relies on so-called binding objects. Since each required component instance may, in turn, expose an interface for events and an interface for calls, the instance binding object needs to maintain a reference to a proxy for calling the methods provided by a child component instance and a reference to an event collector for receiving events that are generated by the child. Furthermore, in order to support the automatic adaptation of stateful component instances, the binding object also needs to maintain information about the state of the child, i.e. the history of method calls and checkpoints. For resource bindings it is sufficient to maintain a reference to the corresponding resource assignment. Since resources in PCOM are always used locally, this reference can be a direct reference. Finally, in order to enable component instances to efficiently inspect the demand of their parents as well as the provision of their children, the context object and the bindings respectively are maintaining a cached version of the relevant parts of their contracts. These cached versions are stored initially during the configuration and they are automatically updated if the part of the underlying contract is changed. In order to receive notifications about changes, the component instance may register listeners at its context object.

Analogous to components, the resource table also contains a mapping between the globally unique identifiers and the corresponding context objects for resource managers. Since resource managers perform similar tasks to component factories, the context objects of resource managers are also maintaining a reference to the resource manager and table of the issued resource assignments. The reference to the resource manager is used to query for resource contracts and to request resource assignments. The table of issued resource assignments is used to compute the remaining amount of available resources and to remove the resource from the component container upon request.

In contrast to component instances, which are application elements that are designed by the application developers, resource assignments are generic objects that are a part of the component container implementation. Thus, there is no need for providing a special context object for resource assignments. Instead, the resource assignment can be seen as the context object. Similarly, the optional handle to the resource which can be included in a resource assignment can be seen as the application element. Since resource assignments do not depend on other application elements and cannot be used directly from a remote computer, the information held by them is rather simple. Apart from the optional handle to the represented resource, the resource assignment stores the resource contract and information about the status of the assignment, i.e. whether it is started or stopped. Finally, in order to avoid duplicate copies of the same contract on the same component container, it contains a reference to the resource binding that uses the assignment in order to gather the details of the demand directed towards the assignments instead of a cached version.

5.2.5 Application Manager

As discussed previously, the application manager is responsible for managing the preferences and initiating the configuration and adaptation. To do this, the application manager provides an interface for users that can be used to define application preferences and to start and stop an application. Furthermore, in order to interact with component containers, the application manager provides an interface that allows the component containers to signal changes and failures. Since there is no conceptual difference between preferences and requirements, we can reuse the demander interface of the component container. As a consequence, a component container does not have to differentiate another component container from an application manager which unifies its implementation. Furthermore, since the application manager now needs to mimic the behavior of a container its internal implementation becomes quite similar.

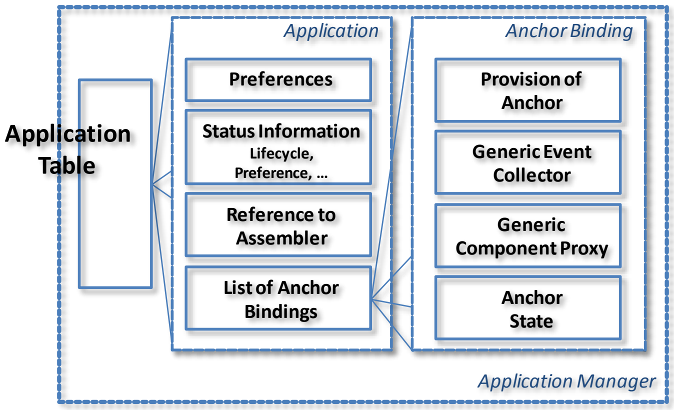


Figure 41 – Application Manager Implementation

Figure 41 shows the main data structures of the application manager. Similar to a component container, the application manager holds a table that maps globally unique identifiers to application objects. The application object consists of the preferences which are essentially an ordered list of contracts that describes the requirements on the application anchor. Furthermore, for each application it stores information such as the current lifecycle state and the configured preference, etc. In order to enable the support for different assemblers, the application manager stores a reference to the assembler that is responsible for configuring the application. Finally, in order to mimic the behavior of a component container, the application manager must hold a data structure that closely resembles the component binding of the component container. However, since the application manager will not interact with the application anchor, it is sufficient to maintain generic proxies and event collectors that implement the protocols required by the component container. In order to react to changes, e.g. to start the configuration and adaptation process, the application manager uses one thread per application that is started when the user requests an application startup and stopped when the application is stopped.

5.2.6 Assembler

The assembler is responsible for configuring and adapting a configuration. As indicated in Section 5.1.9, the application manager and the component container cooperate closely with the assembler during the configuration and adaptation. Specifically, they provide the model of the existing configuration and the associated adaptation costs. Furthermore, they enable the assembler to retrieve relevant information about the environment such as the available components, resources and contracts. Since the general interaction between these three entities is fixed, we have developed a basic framework to simplify the implementation of different assemblers.

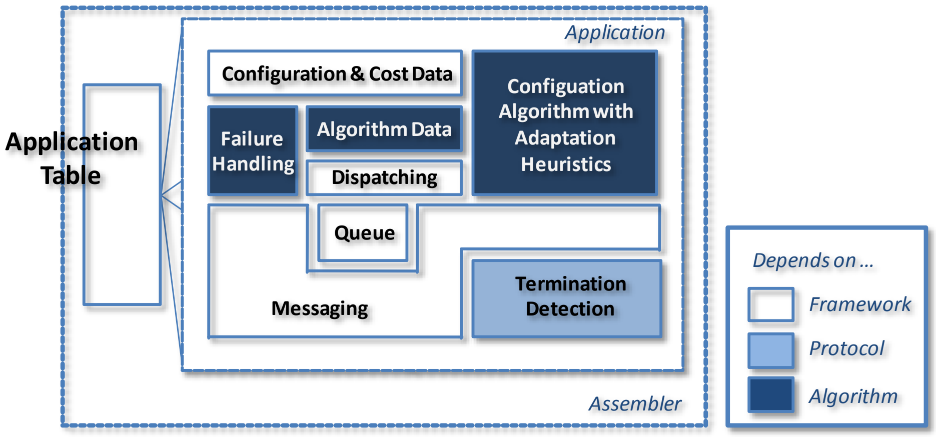


Figure 42 – Assembler Implementation

Figure 42 depicts this framework together with the protocol and algorithm-specific parts. Since an assembler must be able to support multiple simultaneous configuration and adaptation processes for different applications, it contains an application table that maps the unique identifier of an application to a data structure. This data structure represents the configuration or adaptation process for a single application on a single computer. It is created when the configuration or adaptation process is about to be started and it is deleted when the process has finished. In order to avoid stale state, the data structure is tied to leases provided by the registry described in 5.2.2. In our implementation of the configuration algorithm and adaptation heuristics described in Chapter 3 and Chapter 4, the leases are monitored by the assembler that is hosted on the computer with the application anchor.

The general parts of the data structure consist of a messaging subsystem that uses queues and a dispatching subsystem to decouple the message transmissions and receptions from processing. In order to efficiently support algorithms that require termination detection protocols, the messaging subsystem provides a generic interface for different termination detection protocols. In order to support failure handling during the execution of the algorithm, the messaging subsystem can signal failures and it can suppress outdated messages using an epoch value as described in 3.2.3.6. To maximize the performance of the messaging subsystem, it is implemented multi-threaded and it uses the streaming semantic described in Section 5.2.3. Since multi-threaded algorithm implementations are complicated, the messaging subsystem serializes the processing of messages using a dispatch module. This allows us to implement the algorithm described in Chapter 3 and Chapter 4 as reactive processes without additional synchronization – just like we have described them. Finally, in order to support adaptation, the framework stores the information about the existing configuration in a data repository. This data repository can be consulted, for example, to determine the adaptation costs of different adaptation options.

To implement the configuration algorithm and the adaptation heuristics, we implement a credit-based termination detection protocol that we hook up to the messaging subsystem. Furthermore, we add storage for algorithm-specific information, i.e. the configuration objects, the amounts of available resources on the component container, etc. Finally, we implement the basic algorithm as detailed previously and in order to support resilience, we implement a module that performs failure handling. The failure handling module is responsible for signaling failures to other computers and it creates the necessary additional constraints, in cases where a computer becomes unavailable during configuration. Thus, it essentially implements the algorithm modifications described in Section 3.2.3.6.

5.2.7 Graphical Interface

In addition to implementing the system services of PCOM, we also developed a set of graphical user interfaces that enable users and developers to interact with the core runtime system. Since the implementation of the PCOM system service allows the flexible configuration of a system with different subsets of the system services, we have also implemented the graphical user interfaces in a configurable manner. To do this, we divide the user interface into application browser, component browser and assembler visualizer library as depicted in Figure 43.

In order to support a broad range of computers, the user interface parts have been implemented using the multi-platform user interface library SWT. Thus, the user interfaces can be executed on personal computers that are running Windows, Linux or MacOS as well as on most systems based on Windows CE. To support a broad spectrum of mobile phones as well, we also implemented MIDP versions of the application browser and the container browser. However, due to the limited screen real-estate that is typically available on mobile phones, we did not implement the assembler visualizer library using MIDP. Thus, this library is solely available on systems that are supported by SWT.

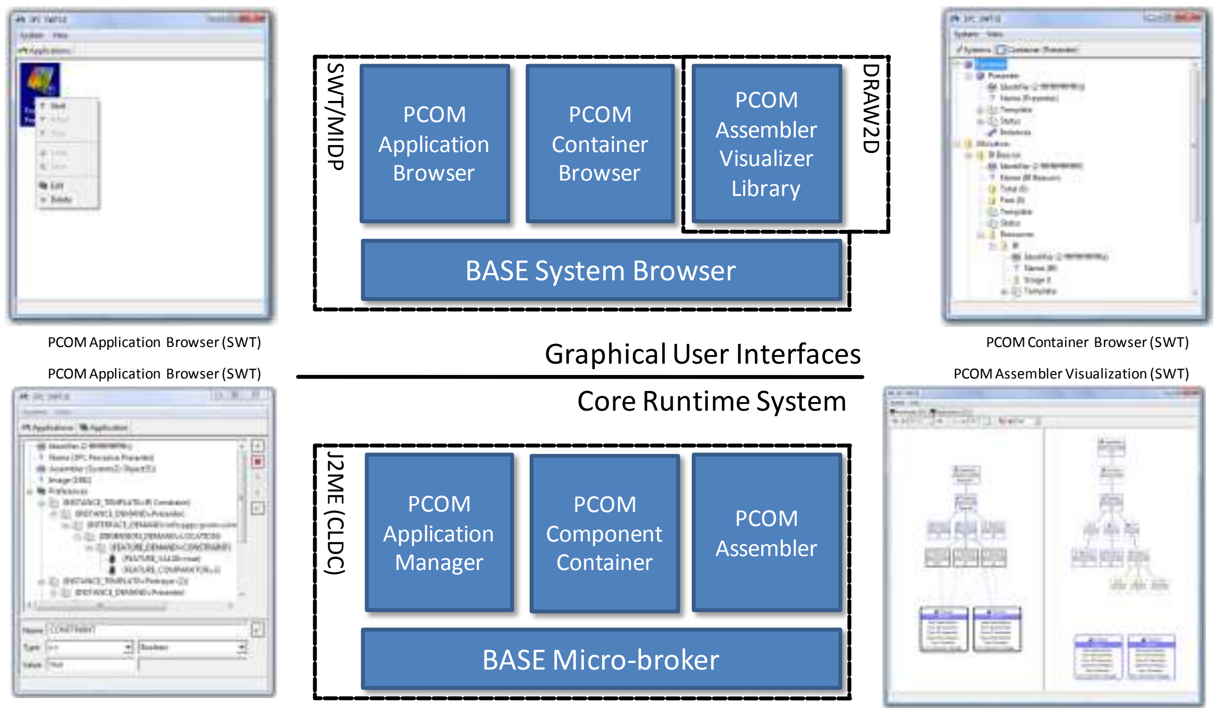


Figure 43 – Graphical User Interface

As indicated by the names, each user interface part supports one architectural components of the core runtime system of PCOM. The application browser enables the interaction with the application manager. The container browser enables the interaction with the component container and the assembler visualizer library simplifies the development of visualizations for assembler implementations. All user interface parts are developed on top of the BASE system browser and they solely depend on the presence of this user interface and the corresponding PCOM system service. In the following, we briefly outline the functionalities of the individual user interface parts.

The application browser enables users to start and stop PCOM applications. To do this, a user may define its preferences for a certain application anchor and the user may select a desired assembler implementation that is used to configure and adapt the application. The preferences can be used immediately to start an application and they can be stored in the application browser for later reuse. In addition to starting and stopping applications, the application browser also enables users to manually trigger the reconfiguration of a running application. This functionality can be used to manually search for better application configurations and it can be useful for developers in order to test Assembler implementations.

The container browser enables users to visually inspect the current internal state of a local or remote component container. The container browser is capable of displaying the installed components and resources. Furthermore, it can list the executed component instances and resource assignments together with their contracts. In addition, the container browser can also display the amount of available resources for each resource and the amount of resources used by the currently issued resource assignments. Finally, the container browser can also revoke resource assignments and it can stop the execution of a component instance. This simplifies runtime testing of resource and component implementations by manually injecting the typical failures resulting from mobility and unforeseeable resource fluctuations.

In contrast to the assembler browser and the container browser which can be used directly by users, the assembler visualizer provides a library for developers. This library simplifies the process of developing visualizations for Assembler implementations. To do this, the library provides a set of user interface controls to visualize the dependencies between component instances and resource assignments. In addition, the assembler visualizer Library also contains a control that animates the construction of a configuration and hides the details of computing a proper layout for the resulting tree of component instances and resource assignments. By integrating this control with the assembler implementation, a developer can create animated visualizations of the underlying configuration algorithm that can be executed at various speeds.

5.2.8 Development Tools

In addition to the graphical user interface, we also provide a set of development tools that simplify component development. In order to develop a component, a developer needs to implement a component factory that creates component contracts and a component instance that provides the component-specific functionality. Both, component factories and instances, are required to implement a certain set of interfaces provided by the container. In addition, the component instance also needs to implement the component-specific interfaces declared in its component contract. Furthermore, to connect component instances with arbitrary component interfaces, PCOM requires an appropriate skeleton for the component instance and a set of proxies for the dependencies of the instance. Due to the technical restrictions of the J2ME CLDC runtime environment, these proxies and skeletons cannot be generated at runtime by the container but they must be generated offline.

As a result, it may become quite complicated to ensure that all proxies and skeletons are generated properly and that the component instance, the component factory and the component contract exhibit the required structure. This is especially problematic for developers that do not fully understand the overall concepts of PCOM, e.g. when they just started to work with the component system. In order to simplify this, we have developed a development tool that is integrated into the Eclipse platform as shown in Figure 44.

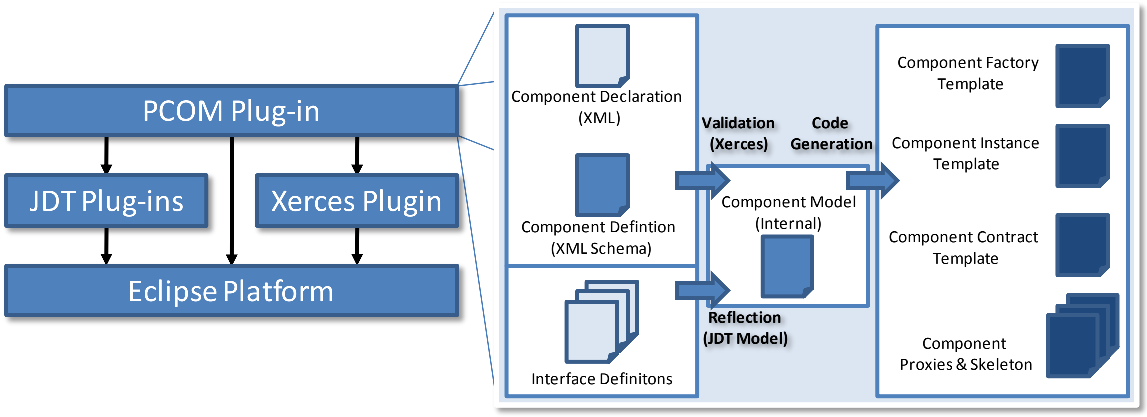

PCOM Plug-in
JDT Plug-ins
Xerces Plugin
Eclipse Platform
Component Declaration (XML)
Component Defintion (XML Schema)
Interface Definitons
Validation (Xerces)
Code Generation
Component Model (Internal)
Reflection (JDT Model)
Component Factory Template
Component Instance Template
Component Contract Template
Component Proxies & Skeleton

Figure 44 – Development Tools

To use this plug-in, a developer first defines the application interfaces of the component using Java. Thereafter, the developer needs to specify an XML document that describes the contents of the component contract. Using this specification, the plug-in can check whether all required application interfaces are available and whether they are suitable, e.g. whether their parameters can be serialized by BASE and whether the methods of the interfaces declare remote exceptions that might be raised by BASE, etc. If this check succeeds, the plug-in generates appropriate templates for the component instance, the component factory, the component contract as well as the required skeleton and all potentially required proxies. If the check fails, the plug-in can bring the problems to the attention of the developer by marking the lines of code that are causing the problem.

The basis of this overall process is an XML schema that defines all possible structures of a component in an abstract manner. Using this XML schema, a developer can declare the provision and the demand of the component instance as an XML document. By using existing Eclipse plugins such as the XML editors provided by the WTP project, this declaration can be done graphically. The graphical editor ensures that the XML document conforms to the XML schema at design time. When such an XML document is passed to the PCOM development tools, the plug-in can use existing validating XML parsers such as Xerces to ensure that the structure is valid. In order perform the checks on the component-specific interface, the plug-in relies on the reflection capabilities of the Eclipse JDT plugins. The JDT plugins are also used to generate the resulting templates for the component factory, instance and contract as well as the proxies and the skeleton.

5.2.9 Simulator

Finally, in order to ease the evaluation of the configuration algorithm and the adaptation heuristics, we have developed an event-discrete simulator for the component system. Just like the simulator used in (Yokoo, Durfee, Ishida, & Kuwabara, 1998), the simulator abstracts from all details of the underlying computer hardware and networking technology. To do this, it delivers and processes all messages generated in one time "tick" in the next time "tick". This allows us to measure and compare many relevant properties of different algorithms in a broad variety of scenarios in an abstract manner. By performing a small number of real-world experiments, we can then relate the abstract figures to real configuration and adaptation overheads.

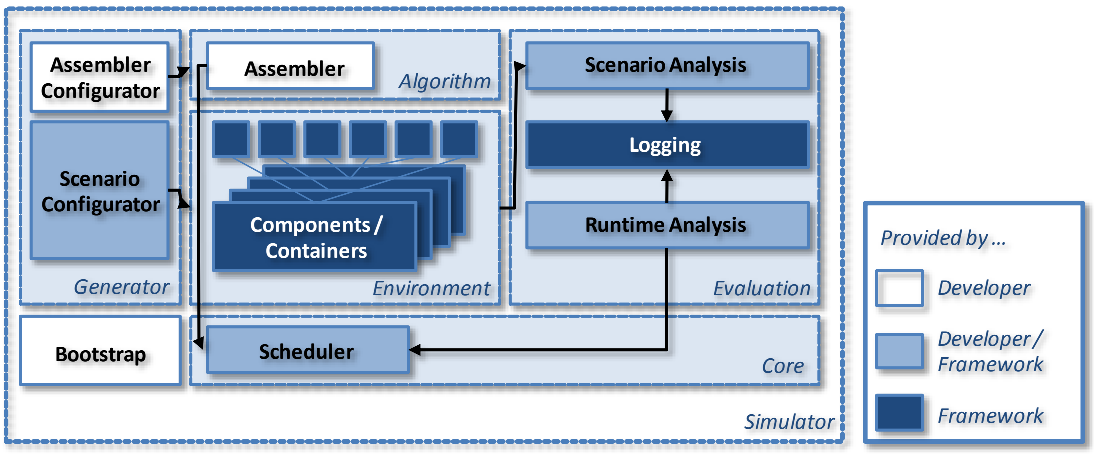


Figure 45 – Simulator

As shown in Figure 45, the simulator consists of five basic modules. The core of the simulator is formed by a scheduler that is responsible for message scheduling and processing. In order to process messages, the core relies on the environment module which provides a light-weight implementation of the fundamental concepts of the component system, e.g. containers, components, resources, etc. The environment module has interfaces that allow the implementation of different configuration and adaptation algorithms. Furthermore, both the environment and the core have interfaces to support the analysis of characteristics of the scenario modeled by the environment and the properties of the algorithm. Finally, in order to generate different applications and smart peer groups, the simulator provides a scenario generator.

Since the simulator is a tool for the evaluation of the properties of algorithms, practically all relevant aspects can be configured and extended. The only exceptions to its extensibility are the abstractions of the component system and the logging module. However, as indicated by the different colors in Figure 45, there are (multiple) default implementations for all aspects of the simulation that are independent from the algorithm. Thus, in order to simulate the execution of an algorithm, a developer solely has to implement the algorithm using the framework defined by the simulator and if necessary, a configuration module for the algorithm. Thereafter, the developer can perform different simulations by creating a bootstrap program that selects the proper modules for scenario generation, scheduling and evaluation. The output generated by the logging module can then be imported in other programs to aggregate or visualize the results.

5.3 Discussion

In this chapter, we have presented a prototypical architecture and implementation of the component system described in Chapter 2 and we have discussed how the configuration algorithm and the adaptation heuristics detailed in Chapter 3 and Chapter 4 can be integrated. The design goals for the architecture are completeness, resilience, minimalism, efficiency and support for pluggable algorithms. The conflict of support for pluggable algorithms regarding efficiency and minimalism is resolved in favor of pluggable algorithms to simplify the comparison of alternative approaches for automatic configuration and adaptation.

Completeness is achieved by integrating all concepts and mechanisms described in the previous chapters. Minimalism is achieved by applying the classical design principle of separation of concerns. This results in a lean component container implementation that does not have to deal with the specifics of preferences, automatic configuration or automatic adaptation. Resilience is achieved by allowing the individual parts of the architecture to fail independently. This is made possible by enabling the component containers to control all processes that are critical to ensure the desired behavior. Efficiency is supported by designing the interfaces between the architectural parts in such a way that they avoid unnecessary communication. Furthermore, in cases where the architecture must support multiple implementations, we incorporated alternative ways of interface usages that allow the efficient implementation of many alternatives. Examples for this are the assembler interface which supports the eager as well as the lazy retrieval of a configuration or the container interface which supports simple and batch queries.

To simplify the implementation of the component system, we have reused an existing implementation of the BASE middleware for smart peer groups. To enable the efficient implementation of the architecture on top of this middleware, we have extended it with an additional system service to detect the unavailability of arbitrary remote objects. Furthermore, we have added an additional communication plugin to support the efficient transmission of comparatively high amounts of small messages in FIFO order. Using these extensions, we can implement all architectural parts as system service on top of the micro-broker. In order to improve the usability of the resulting core system, we have additionally developed a set of graphical user interfaces and an Eclipse-based development tool. Finally, in order to simplify the systematic evaluation of configuration algorithms and adaptation heuristics in various settings, we have developed an event-discrete simulator that allows us to create arbitrary smart peer groups. Using this simulator, we can compare the performance and overhead of multiple algorithms in thousands of different scenarios.

In the next chapter, we use this implementation to evaluate the component system presented in Chapter 2 as well as the configuration algorithm and the adaptation heuristics presented in Chapter 3 and Chapter 4. In Chapter 7, we discuss how other system software compares to the proposed architecture and in Chapter 8, we outline possible extensions.

6 Evaluation

The chapter evaluates the component system proposed in Chapter 2 using the implementation presented in Chapter 5. To show the performance of the implementation, the Chapter presents a set of micro benchmarks for relevant parts of the system. To show the benefits and limitations of the abstractions, the chapter discusses an exemplary application. After the evaluation of the component system, the chapter presents an evaluation of the configuration algorithm described in Chapter 3. Thereby, the evaluation is centered on performance measurements in different settings. To perform a thorough evaluation, the chapter presents an extensive set of event-discrete simulations and to show that the simulations approximate reality, the chapter additionally presents a set of real-world experiments. After the evaluation of the configuration algorithm, the chapter provides an evaluation of the adaptation heuristics presented in Chapter 4. To do this, it discusses the results of a number of event-based simulations. Finally, the chapter closes with a discussion.

6.1 System Software

In the following, we evaluate the overheads as well as the benefits and limitations of the component system proposed in Chapter 2. First, we present a set of micro-benchmarks that have been gathered using the prototypical implementation presented in Chapter 5. Thereafter, we contrast the proposed approach for component-based application development with the service-oriented approach for application development taken by other systems. As basis for this discussion, we use the abstractions introduced by the BASE communication middleware. Given that we used BASE, both to derive the necessary abstractions and to implement them, this choice is quite natural.

6.1.1 Performance

To show the overheads associated with utilizing the component system, we first present a set of measurements and micro-benchmarks. To put the results into context, we compare them with similar measurements and micro-benchmarks that have been gathered using the BASE communication middleware. The reason for selecting BASE for comparison is twofold. First, PCOM has been implemented on top of BASE and thus, the performance of BASE defines the baseline for PCOM. Secondly, BASE is also targeted at simplifying the development of pervasive applications. However, in contrast to PCOM, BASE does not enable the automatic configuration and adaptation of applications. Instead, it shifts the responsibility for this to the application developer. As a consequence, the comparison of BASE and PCOM clearly illustrates the cost incurred by automation.

We start the discussion with a comparison of the overall memory footprint of BASE and PCOM. Thereafter, we compare individual mechanisms along the lifecycle of an application. Specifically, we compare BASE service selection with PCOM component matching, communication between BASE services and between PCOM components and finally, we describe the overheads of the monitoring and signaling mechanisms of PCOM. To get meaningful absolute numbers for resource-poor computers, all measurements have been gathered with the IBM J9 JVM on Windows Mobile PDAs (Xscale PAX270) that are connected via a wireless network (802.11b). To reduce uncontrollable side-effects, the just-in-time compiler of the virtual machine has been deactivated so that it is solely running in interpreter mode. After presenting the overheads, we discuss the benefits and limitations of the component system in the next section. The overall overheads of automatic configuration and adaptation are discussed in depth in Section 6.2 and Section 6.3.

6.1.1.1 Footprint

Due to the implementation of PCOM as extension to BASE, the component system increases the memory footprint. Since the component system does not require all architectural building blocks on all computers, the exact overhead depends on the configuration of the computer. The set of building blocks that is required on a computer, in turn, depends on the application scenario, for example, whether it shall provide a component or whether it shall solely start applications, etc.

	LOC	NOC	NOI	SIZE (KB)
BASE Micro-broker	4514	33	25	141
BASE Service Registry	621	9	1	28,5
BASE Plugins (Bluetooth, IP, IRDA)	5273	48	1	186
BASE GUI (LCD)	3234	26	3	105
BASE GUI (SWT)	2931	28	4	171
PCOM Lease Registry	743	9	1	30,1
PCOM Component Container	7607	56	68	299,6
PCOM Application Manager	1411	6	1	41
PCOM Assembler	1500-3000	15-30	10-30	50-100
PCOM GUI (LCD)	2113	22	2	32,6
PCOM GUI (SWT)	3329	42	4	217
PCOM GUI Assembler (SWT)	1152	7	0	50

Table 1 – Memory Footprint per Building Block

Table 1 shows the memory requirements for different architectural building blocks of BASE and PCOM. Besides from showing the uncompressed binary code size in kilobytes (SIZE), the table also shows the lines of code without comments and blank lines (LOC), the number of classes

(NOC) and the number of interfaces (NOI). The table indicates that the PCOM component container is significantly larger than the BASE service registry. However, this should not come as a surprise, given that the BASE service registry solely provides a minimal lookup service whereas the component container provides the component lifecycle guarantees and mechanisms for automatic configuration and adaptation. The memory footprint of the application manager is similar to the size of the service registry. The memory footprint of the assembler depends on the concrete implementation. A simple assembler can be implemented with approximately 1500 lines of code. A more complicated assembler that implements the configuration algorithm and adaptation heuristics described in Chapter 3 and 4 can easily double the size. The user interfaces for BASE and PCOM exhibit approximately the same size.

	LOC	NOC	NOI	SIZE (KB)	DIFF (%)
BASE (Usage)	10408	90	27	355,5	-
PCOM (Usage)	11941	96	28	398,1	12
BASE (Provision)	10408	90	27	355,5	-
PCOM (Provision)	19673-21173	161-176	105-125	706,7-756,7	99-113
BASE (PDA)	13339	118	31	526,5	-
PCOM (PDA)	27308-28808	237-252	114-134	1135,7-1185,7	116-120

Table 2 – Memory Footprint per Configuration

More interesting than the raw size of the architectural building blocks is the increase of the footprint for different usage scenarios. Table 2 depicts three configurations of BASE and PCOM with the necessary architectural building blocks for using a service or an application (Usage), providing a service or a component (Provision) as well as an exemplary configuration for a PDA. As indicated by the table, with exception of using an application (Usage), the memory footprint increases approximately by a factor of two. The fact that the application usage does not lead to a drastic increase can be attributed to the fact that PCOM solely requires the lease registry and the application manager in order to use an application. However, if a computer shall offer a component, it needs to be equipped with a lease registry, a component container and an assembler.

It is noteworthy that the absolute memory footprint in kilobytes (SIZE) of BASE and PCOM shown in Table 1 and Table 2 might seem prohibitively high for resource-poor systems at a first glance. However, the comparatively high absolute numbers result mainly from the Java programming language and can easily be reduced by a factor of two to five without modifying the implementation. The root cause of the problem is that a Java application can always load additional code. As a consequence a Java class file always contains full qualified signatures of implemented and referenced methods and types. At the same time, the Java coding conventions require that package collisions ought to be avoided by using full qualified domain names to uniquely identify packages. Together these two factors lead to very large constant pools in the Java class files which consist mostly of repetitive string constants.

The usual way to mitigate this problem is to compress the binary code before shipment. Applied to BASE and PCOM, this approach already reduces the footprint by a factor of two to three. A more effective way to reduce the size is to use an obfuscator to modify the signatures of methods and types. This approach is often taken by development tools for embedded systems. Usually, the programs on these systems must be uploaded into a static program memory with external development tools. As a result, the program code cannot be changed and extended at runtime. In effect, this nullifies the extensibility of Java programs and thus, it is possible to statically link the individual program parts.

By using these approaches to reduce the memory footprint, we have successfully deployed PCOM on all target platforms that are also supported by BASE. This includes embedded micro-processors such as the JStamp+, Linux- and Windows Mobile-based personal digital assistants, mobile phones, laptops, desktops, etc. As a consequence, we can say that the increase in memory footprint is not prohibitive, but it must be noted that the additional memory required for PCOM is not available for applications. On the other hand, if an application needs support for parameterization or automatic configuration, the application developer has to implement the functionality of the component system as part of the application and this increases the size of the application. If a computer provides only a single service, the increase might be slightly smaller but if the computer shall host multiple services the increase can easily be higher, especially, if the mechanisms are implemented individually for each service.

6.1.1.2 Matching

Another source of potential overhead is the description of components and resources with contracts. In order to find suitable components and resources, the component system has to match demands with provisions since components and resources are always bound dynamically. In contrast to that, services in BASE may interact with other services directly – given that they have the necessary remote reference. However, in order to acquire a reference in the first place, a service must retrieve it via some naming or trading services. BASE implements a federated naming and trading service as part of its service registry. In order to search for suitable implementations, a BASE service uses properties that support the comparison of different services. To get an indication for the overhead of the contract model introduced by PCOM, we compare it with the property-based service description of BASE service in the following.

Parameters	10	20	30	40	50
BASE Service Description	0,140	0,234	0,349	0,468	0,621
PCOM Component Contract	2,851	4,836	7,107	10,600	13,491
Overhead	20,364	20,667	20,364	22,650	21,725

Table 3 – Local Matching Latency and Overhead

Table 3 and Table 4 compare the overhead of one local and one remote lookup for service descriptions and component contracts with an increasing number of name value pairs or parameters, respectively. The parameters contained in the contract are strings with a comparison operator that validates case-sensitive equality to closely mimic the behavior of BASE service descriptions. To reduce uncontrollable side-effects of system immanent mechanisms such as automatic garbage collection, the values resemble the average of ten runs with 1000 matches.

Parameters	10	20	30	40	50
BASE Service Description	86,970	142,200	187,210	235,760	267,960
PCOM Component Contract	123,190	201,480	269,540	336,720	389,700
Overhead	1,416	1,417	1,440	1,428	1,454

Table 4 – Remote Matching Latency and Overhead

As indicated by the figures for local lookups, searching for a component contract with twenty attributed requires approximately 4.836 milliseconds whereas searching for a service description requires only 0.234 milliseconds on average. Throughout the measurements, matching a component contract is approximately twenty times slower than matching a service description. This can be attributed to three factors. First and foremost, component contracts provide a significant higher flexibility by supporting multiple data types and different comparators. This increases the overhead during the evaluation as it requires multiple type inspections and casts. Secondly, the lookup interface of the component container supports batch queries and thus, the query and the results of the query need to be wrapped in an additional data structure. Finally, the contracts are created dynamically upon each request, whereas service descriptions are created statically in advance.

Although this overhead seems to be quite high at a first glance, the comparison of the remote lookups shows a different picture. Here, the relative overhead shrinks down to approximately 40% percent. The reason for this is that remote method calls in BASE are much more expensive than the local lookup, e.g. factor 400 to 600. Thus in a real-world scenario, in which components need to searched on all computers of the smart peer group, the seemingly high overhead for component matching is clearly dominated by remote communication. At the same time, the absolute numbers indicate that the total cost of 2-14 milliseconds for local lookups is still reasonable. The remaining cost factors of remote calls can be attributed to differences during serialization and the more flexible structure of contracts.

6.1.1.3 Communication

Besides from lookups and matching, a potential source of overhead is communication between component instances. Since our implementation of PCOM uses similar proxies and skeletons as BASE and since the component container registers them directly at the BASE micro-broker, we can expect very little overhead. However, due to the fact that the component container inspects the return values in order to detect broken dependencies, there is some small overhead for each method call. Furthermore, if a component developer wants to utilize the logging to automate the management of application-specific component state, there is additional overhead. To determine exact numbers for these overheads, we compare communication of BASE services with communication of PCOM component instances.

Size	0K	10K	20K	30K	40K
BASE	101,050	120,650	123,900	132,800	158,550
PCOM (Plain)	103,900	120,600	124,500	135,000	159,100
PCOM (Logging)	106,000	127,050	130,600	140,600	170,600
Overhead (Plain)	1,028	1,000	1,005	1,017	1,003
Overhead (Logging)	1,020	1,053	1,049	1,041	1,072

Table 5 – Remote Communication Latency and Overhead

Table 5 shows the latency of a remote method call between services and components for varying parameter sizes. For BASE, we simply measure the latency of a synchronous method call between two services. For PCOM, we perform two measurements, one without message logging (Plain) and one with message logging (Logging). If message logging is disabled, only the return value needs to be checked. If logging is enabled, the parameters of the call need to be serialized and stored. To vary the size of the parameters of the remote call, we transmit a byte array with different sizes from 0 to 40000 bytes. As indicated by the measurements, PCOM does not introduce significant overhead in cases where logging is disabled. If logging is enabled, the overhead depends on the size of the parameters but it remains low, even for larger parameter sizes, e.g. seven percent for 40000 bytes.

6.1.1.4 Monitoring

Another source of overheads is the monitoring mechanism introduced by PCOM. In order to detect the unavailability of a required component instance, PCOM monitors the communication between component instances. As shown previously, the overhead for this can be neglected. In addition, the component system utilizes a lease mechanism to detect the unavailability of component instances that are not contacted frequently. The lease mechanism is also used to prevent stale component instances on computers that have left the smart peer group. By its very nature, the mechanism introduces periodic communication to renew existing leases. If a communication attempt or a lease renewal fails, the application is notified by means of a transition in its lifecycle. The overhead for signaling these transitions is discussed in the next section. Since there is no counterpart for these mechanisms in BASE, we cannot perform a comparative evaluation to determine the overhead. Instead we discuss an analytical model for the cost of the monitoring mechanism in terms of additional communication and we perform measurements to determine the cost of the signaling mechanisms.

Since the soft-state lease mechanism introduces period communication, the overhead for monitoring depends mostly on the periodicity and the total monitoring time. If there are no failures during the execution of an application, the total monitoring time boils down to the total time that the application is executed. At the beginning of the monitoring, the component container that provides a component instance for another component container registers a lease at the lease registry and transfers it to the requester. Thereafter, the requesting component container registers the lease to start the monitoring. During the monitoring period, the lease registry on the computer that hosts the requesting component container will periodically send lease renewals. At the end of the monitoring period, the lease registry sends a lease removal to indicate that the associated component instance can be released. As a consequence, we can model the overall requirements with respect to communication as $M_C + n*(M_R + M_A) + M_D$ where M_C represents the transfer of the newly created lease, M_R represents a renewal message, M_A represents the acknowledgement and M_D represents the deletion request. The number of lease renewals n can be computed from the total monitoring time T and the periodicity P as $n = T / P$. In order to compute the actual cost, we measure the size of the messages that need to be transferred.

Message	M_C	M_R	M_A	M_D
Size	47	69	56	60

Table 6 – Message Sizes for Monitoring

Table 6 lists the message size for all necessary messages in bytes. The figures solely represent the application level payload and do not include any headers from lower layers of the protocol stack. The primary reason for this is that BASE can support different protocol stacks and communication technologies which results in different header sizes. Furthermore, the lease registry automatically bundles lease renewals that need to be sent to the same computer at around the same time. Thus, the relative cost introduced at lower layers decreases with an increase of leases. As one can see, the absolute size for the messages is small, i.e. below 100 bytes, and given that the periodicity is usually specified in the order of seconds, the total cost for monitoring is also small. However, it should be clear that there is a tradeoff between the monitoring quality and the periodicity. By decreasing the frequency of renewals, failures can be detected earlier. However, this also causes higher costs.

6.1.1.5 Signaling

The last source of overhead is the component lifecycle introduced by PCOM. If a new application is started, all component instances need to be instantiated and started. If an invalid application configuration is detected, all available component instances need to be paused. If an application is stopped, all component instances need to be stopped and released. To perform these notifications, PCOM traverses the configuration starting from the application anchor. Since BASE services are not equipped with a lifecycle, we cannot compare it with an existing mechanism in BASE. Thus, in analogy to the monitoring mechanisms, we mainly focus on measuring the cost for performing these notifications.

However, it must be noted that the traversals of the configuration are not only used to signal a transition in the lifecycle. As indicated in Section 5.1.9, the traversals are also used to instantiate components upon initial startup, to compute the cost model and to load the application model into the configuration algorithm upon pause, to retrieve the new configuration from the algorithm and to modify the existing configuration after a successful reconfiguration, and to release the component instances while stopping an application. As a result, the total cost indicated by the following measurements would have to be divided among these mechanisms as all of them require traversals.

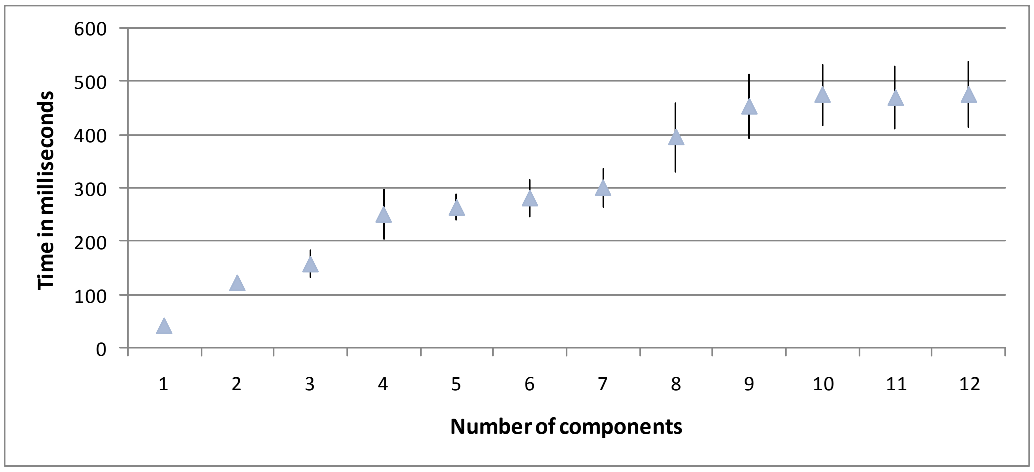


Figure 46 – Latency for Pausing a Configuration

To determine the overall cost for a traversal, we measure the total time to pause a configuration before an adaptation takes place (Figure 46) as well as the total time to start an application after the application has been reconfigured successfully (Figure 47) for applications with a varying number of components. The application structure resembles a binary tree, i.e. each component instances requires at most two further components, for which we ensure that all adjacent components reside on different computers. For these measurements, we use 4 Windows Mobile PDAs (Xscale PAX270) that are connected via a wireless network (802.11b). To reduce the effects of uncontrollable system properties, Figure 46 and Figure 47 show the average result of 500 runs per value as well as the standard deviation.

As indicated by the steeper increases at application sizes of 2, 4, and 8 in Figure 46, the overhead for performing a traversal depends to a large extend on the height of the tree as well as on the latency for performing a remote call. The correlation between the delay and the height of the application can be explained easily by the parallel execution of the traversals as discussed in Section 2.3.5.4. Thus, when the application height increases by one, the overall delay is increased by the delay for performing one more remote method call sequentially. In addition, another interesting observation can be made by comparing the total delays of Figure 46 and Figure 47.

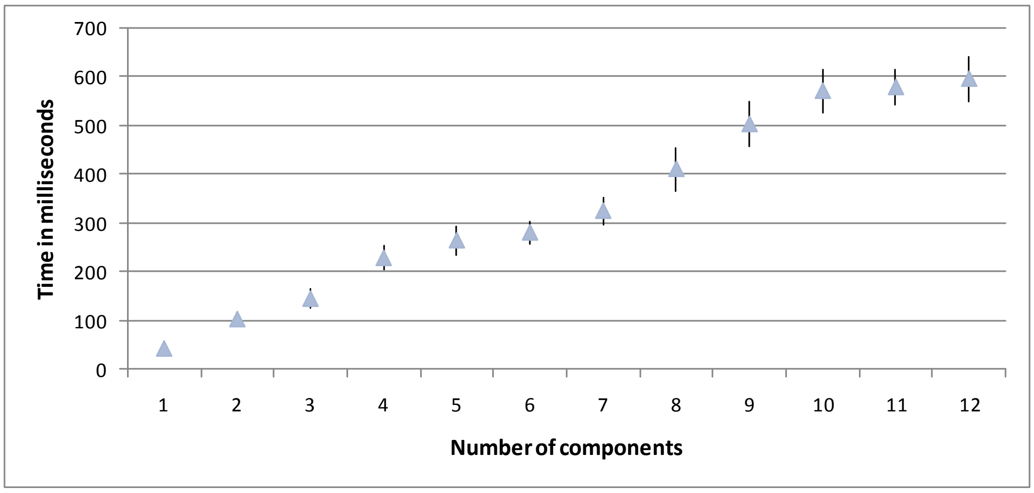


Figure 47 – Latency for Starting a Configuration

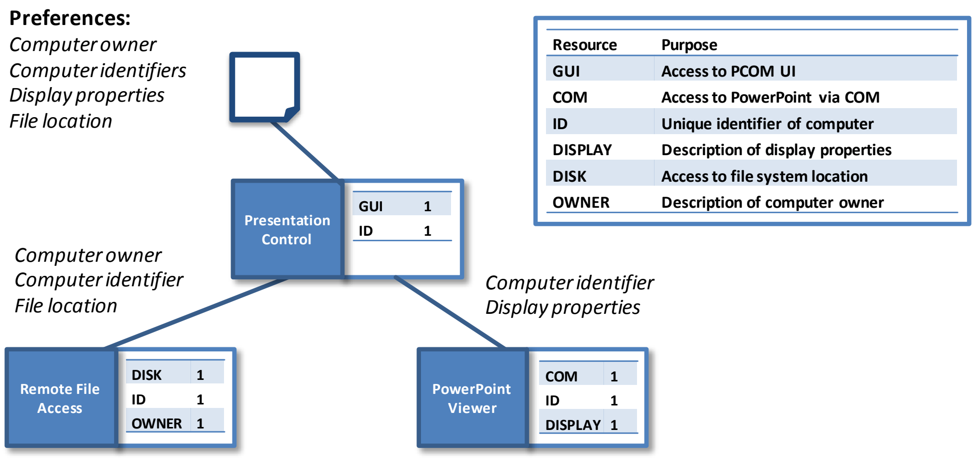
The figures indicate that the overhead for starting an application is higher than the overhead for pausing it. Thus, computing the values of the cost model is obviously significantly less expensive than determining the necessary changes to the configuration which allows us to conclude that the cost model is indeed very light-weight. Finally, with a total overhead of less than 600 milliseconds for notifying an application with 12 components running on 4 Windows Mobile PDAs, we can also conclude that the cost for signaling is tolerable. But probably more important, we can also conclude that in many scenarios, there will only be short periods of time in which the application is started with an invalid configuration.

6.1.2 Benefits and Limitations

In the following, we discuss the benefits and limitations of the component abstraction introduced by PCOM. To do this, we contrast application development in PCOM with application development in BASE. We are aware of the fact that a thorough analysis would require the development of several identical applications using both systems with different groups of developers including professional application developers. Due to the associated monetary costs of performing such an evaluation, we must consider this analysis beyond the scope of this dissertation. Thus, we cannot claim that PCOM simplifies application development in general. However, by describing the development of a realistic exemplary application with PCOM and by discussing how a developer can benefit from the capabilities of PCOM, we can provide anecdotal evidence for its usefulness and we can derive potential limitations.

6.1.2.1 Application

In order to experiment with PCOM, we have developed a set of simple applications that allow us to test, demonstrate and evaluate individual aspects of PCOM. In addition, we also have developed a small set of realistic applications. From this set, we describe one application called *Pervasive Presenter* in the following. The *Pervasive Presenter* allows us to discuss all relevant aspects of PCOM since it uses all features. Furthermore, the implementation of the *Pervasive Presenter* is quite mature and we have used it for multiple public demonstrations of PCOM, e.g. (Handte, Urbanski, Becker, Reinhardt, Engel, & Smith, 2006).



Resource	Purpose
GUI	Access to PCOM UI
COM	Access to PowerPoint via COM
ID	Unique identifier of computer
DISPLAY	Description of display properties
DISK	Access to file system location
OWNER	Description of computer owner

Figure 48 – Pervasive Presenter Application

The use-case for the *Pervasive Presenter* is similar to the use-case of the exemplary applications that we have used to motivate automatic configuration and adaptation in Section 3.1.1 and Section 4.1.1. It allows a user to control a presentation from a hand-held computer in an arbitrary smart peer group. To make use of the capabilities of the smart peer group, the application is split in three main functionalities as indicated in : a control functionality that provides the user interface, an input functionality that acts as data source and an output functionality to display individual slides during the presentation.

For the simplest possible implementation, we have realized these three functionalities as three separate components. The *Presentation Control* running on a personal digital assistant allows a user to load a PowerPoint presentation from some remote file system and to skip from one slide to another. The access to the remote file system is provided through the *Remote File Access* component. The individual slides are shown using a *PowerPoint Viewer* component on a desktop or a laptop. The *PowerPoint Viewer* component simply provides a PCOM wrapper for PowerPoint through COM. In addition to displaying slides, the *PowerPoint Viewer* also converts the currently displayed slide to a small image which is shown by graphical user interface of the *Presentation Control*.

In order to differentiate between components on different computers, we introduce a set of non-functional properties in the component contracts. To avoid the configuration of each individual property on a per-component basis, we create resources (*Owner*, *ID*, *Display*) that provide the details for each computer. As a result, the component implementation becomes generic and can be deployed on different computers without any configuration. Consequently, in order to function properly each component requires some local resource to determine the concrete properties of its hosting computer. As an example consider for instance the *Display* resource used by the *PowerPoint Viewer*. Instead of specifying the concrete resolution or the physical size of the display for each *PowerPoint Viewer* upon deployment, we introduce a resource that specifies these properties. If a *PowerPoint Viewer* with specific display-related properties is requested, the component can simply forward the requirements to the resource. If the resource supports them, the *PowerPoint Viewer* can create a suitable component. As indicated in , by implementing a similar forwarding of properties in the *Presentation Control*, the application can be configured flexibly through user preferences.

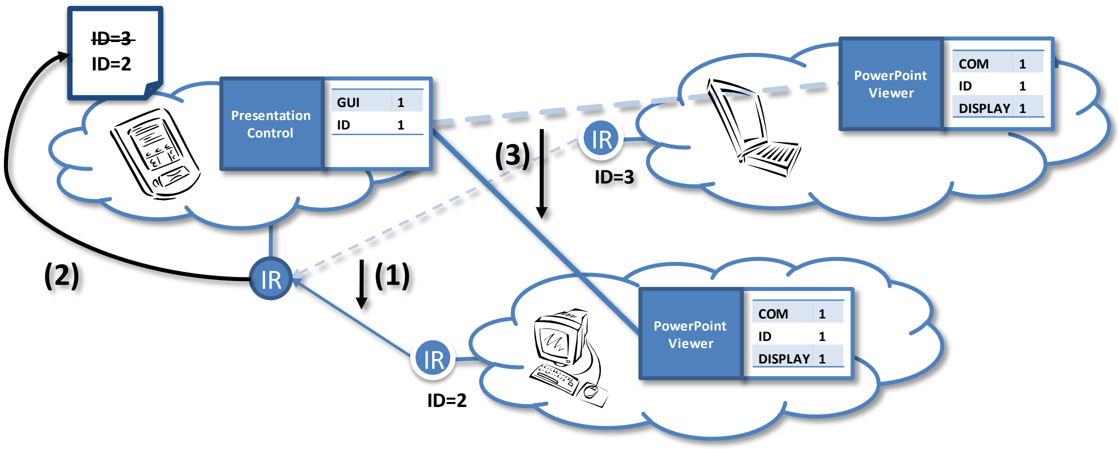
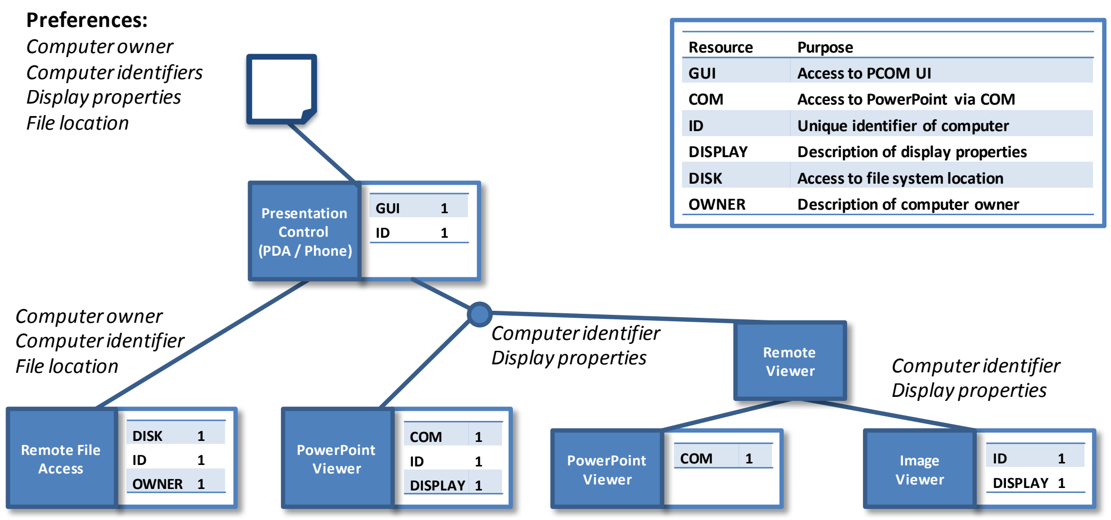


Figure 49 – Pervasive Presenter Application with Dynamic Preferences

Since it might be cumbersome to specify the desired properties of the *PowerPoint Viewer* for different smart peer groups as part of the preferences, we have extended the application manager and the resource that provides the computer identifier. Instead of specifying the concrete identifier in the preferences, a user may specify an abstract value to indicate that the application manager should provide the concrete identifier at runtime. To determine the proper identifier, the application manager listens to incoming identifiers via infrared. The resource manager installed on the computer that hosts the *PowerPoint Viewer* uses infrared to broadcast its identifier. This allows a user to "point" to the computer that shall be used to display the presentation. As shown in Figure 49, a user can also switch from one computer to another by pointing to it. When the application manager receives the new identifier via infrared (1), it simply adjusts the preferences (2). This, in turn, initiates an adaptation which selects the desired component (3). This simple extension demonstrates nicely, how additional context information can be used to simplify the specification of dynamic preferences. Yet, to ensure a broader applicability, the simple infrared-based sensor would have to be replaced with a more powerful context management infrastructure such as Nexus (Hohl, Kubach, Leonhardi, Rothermel, & Schwehm, 1999) or Cobra (Chen, Finin, & Joshi, 2004), for example.



Resource	Purpose
GUI	Access to PCOM UI
COM	Access to PowerPoint via COM
ID	Unique identifier of computer
DISPLAY	Description of display properties
DISK	Access to file system location
OWNER	Description of computer owner

Figure 50 – Pervasive Presenter Application with Multiple Configurations

To demonstrate that the PCOM component model allows the flexible extension of an application in different dimensions, we have developed three additional components. First, we have developed a replacement for the *Presentation Control* component that can be executed on a mobile phone. In terms of functionality, this component is similar to the original version except that it uses a different API to interface with the windowing toolkit of a J2ME-based mobile phone. The implementation of this component shows that it is straight-forward to reuse the functionality of existing components. Secondly, to demonstrate that it is also possible to use different implementations of the same component, we have developed a replacement for the *PowerPoint Viewer*. This replacement removes the requirement of having PowerPoint installed on the computer that shows the slides. To do this, the replacement reuses the image conversion functionality of the PowerPoint component to convert slides. The converted slides can then be displayed on any component that is capable of displaying an image. The resulting set of configurations is depicted in Figure 50.

6.1.2.2 Benefits

In the following, we discuss the benefits of the PCOM component model in contrast to the service model introduced by BASE. To clarify the individual points, we use the exemplary application described in the previous section and we show how this application benefits from the abstractions and mechanisms used by PCOM. The benefits can be summarized as simplified configuration, simplified adaptation and simplified monitoring of applications. In the next section, we describe the potential limitations of the abstractions and the mechanisms and we discuss how they can be avoided or mitigated.

The primary goal of PCOM is to automate configuration and adaptation at the system level in such a way that the application developer can benefit from a high degree of transparency. To do this, the developer must specify both, the requirements and the provision of all atomic functionalities, as component or resource contracts. In terms of development effort, the specification of component or a resource contract is similar to the specification of service descriptors that are used to export or to find services in BASE. Consequently, if the developer of a BASE service needs to use the trading capabilities of the service registry, the development effort for BASE services and PCOM components is similar.

In contrast to a service developer, the component developer does not have to provide logic that evaluates and selects a component or resource as this is done by the configuration and adaptation algorithm. Instead, the component developer must implement the callbacks that signal transitions in the component lifecycle. At a first glance, this might not seem to be a great relief, since performing a query in BASE can be done with a single local method call and comparing different services boils down to iterating through them. However, the configuration and adaptation algorithm proposed in Chapter 3 and Chapter 4 performs much more than just selecting the next matching component. In fact, the algorithm ensures that all selections are conflict-free and low-cost on a global, i.e. application-spanning, basis. This is definitely the single most important simplification introduced by the PCOM component system.

To clarify this, consider that the suitability of the *Remote Viewer* component in the *Pervasive Presenter* depends on the availability of a *PowerPoint Viewer* and the suitability of the *Image Viewer*. Thus, in order to implement adequate decision logic, a service developer would not only have to look at a single service but at all possible combinations of recursively required services. Clearly, implementing this functionality in an interoperable way across multiple services is not desirable and in fact, it is also not sufficient. As counter example consider that even two services in independent parts of the application configuration can conflict due to resource constrains. If this happens, the service developer can either ignore the problem which unavoidably results in cases where applications cannot be executed despite a sufficient amount of available resources. Alternatively, the service developer can also try to coordinate the selection across the whole application which essentially requires the implementation of a configuration algorithm at the application level. Given that an implementation of a simple complete algorithm can easily exceed one thousand lines of code, this is clearly not a desirable alternative. The same argument can be made with respect to adaptation since finding a cost-effective adaptation also requires a systematic cooperative search for a conflict-free set of services.

When looking at the components and resources of the *Pervasive Presenter*, one might argue that the resources utilized by the application are not limited per se. For instance, the virtual resource that provides the computer identifier can be issued an infinite number of times. Similarly, one may argue that memory or processing capabilities do not have to be modeled due to virtual memory and fair scheduling. Thus, if many components are instantiated on the same computer the worst thing that can happen is graceful performance degradation. After all, the virtualization of resources is a frequently used approach to avoid resource conflicts. While this might be true for many resources, the *Pervasive Presenter* also provides two adequate counter examples, namely the user-interface related resources. It is conceivable that in a typical presentation scenario the input capabilities of the mobile phone should be dedicated solely to the control of the presentation. Similarly, if the slides ought to be shown in full-screen mode, the display of the computer running the *PowerPoint Viewer* cannot be shared with other applications. As a consequence, the associated resources must be assigned exclusively which bears a potential for conflicts.

Besides from simplifying the configuration of an application, PCOM also eases the development of robust components that can deal with multiple applications simultaneously. The key to this is component instantiation since it provides a natural way of defining boundaries between applications. If these boundaries are not removed deliberately by the developer, they create locality and thus, they ensure that the scope of failures is restricted. For example, due to the fact that bindings are always established between instances, instances from one application cannot contact the instances of another application accidentally. As a side-effect, the developer does not have to maintain an error-prone mapping between parent components and child components of different applications. Thus, if the proxies are not passed across instance boundaries, two simultaneously running instances of the same component are isolated. Of course, if the cooperation of multiple instances is required, it can be implemented by means of shared memory. However, this is likely to be an exception and the components that we have implemented so far, including the components of the *Pervasive Presenter*, do not require it.

Along the same line, instances can also be used to simplify the management of application-specific state. In our Java-based prototypical implementation, various aspects are even fully automated due to the garbage collection performed by the Java Virtual Machine. For example, by referencing application-specific state from instance variables, the component container can ensure that the memory used by an instance can be freed completely as soon as it is no longer needed. Similarly, it is possible to automatically generate the necessary code to store and to initialize the application-specific state of a component instance which can be used to support the migration of stateful component instances during adaptation. In combination with the automatic message logging facility of PCOM, this can greatly simplify the development of stateful components.

In fact, all output components of the *Pervasive Presenter* are using a combination of checkpointing and automatic message logging to support the seamless continuation of a presentation despite potential reconfigurations. In this application, the code for dealing with failures usually boils down to the same pattern. If a communication attempt fails, the calling thread is suspended until an adaptation has taken place. A successful adaptation is indicated by means of a lifecycle transition. When the transition is signaled, the component system has already reconfigured the application and it has restored the state in components that have been replaced. Thus, the suspended call can be repeated either until it finishes successful or until the application is stopped. The code to achieve this behavior is well below one hundred lines in all components of the *Pervasive Presenter*.

Finally, PCOM also simplifies the monitoring of components in scenarios where the interaction frequency is low. In order to monitor the availability of BASE service, the service developer has no other chance than to initiate periodic communication attempts. In PCOM, this periodic communication is performed automatically by the component containers. If a computer leaves the smart peer group, the component containers automatically trigger an adaptation and they free the component instances and resource assignments that are no longer needed. For interactive applications, this functionality is particularly relevant since communication between components is often an implication of some manual input and the frequency of such inputs may vary heavily in an unpredictable way. In the *Pervasive Presenter*, for example, communication between the *Presentation Control* and the other components is only needed when a presentation is loaded or when a slide is switched. Thus, the periods of time in which no communication takes place can often exceed one minute. However, the monitoring mechanisms of PCOM go beyond the pure monitoring of the availability as PCOM also supports the monitoring of contracts. In the *Pervasive Presenter* application, this monitoring facility is, for example, used to detect changes to resolution of the display that shows the slides. If such a change occurs, the component container first validates that the new resolution is still within acceptable limits. If this is not the case, it initiates an adaptation. If the resolution is still sufficient, the change is signaled to the parent, which may adapt its contract as well.

6.1.2.3 Limitations

In the following, we outline the limitations of application development with PCOM when compared to BASE. Since most of the limitations do not apply to the *Pervasive Presenter* application introduced previously, we discuss additional examples to clarify the points where necessary. The limitations can be summarized as limitations due to overheads, mechanisms and abstractions.

Perhaps the most obvious limitation of PCOM is its overhead with respect to resource consumption. Although, we did not experience this problem on our test platforms, the increased memory footprint might be too high for some computers. Similarly, the additional requirements on communication and computation might rule out some platforms. One way to mitigate this is to develop a specialized component container that solely provides the necessary abstractions. However, when considering the exemplary components and applications that we have developed so far, we are convinced that the proposed abstractions are needed in a broad spectrum of applications and it is highly questionable whether an implementation at the application-level would be much more space efficient. Yet, since our prototype implementation trades off support for pluggable algorithms against minimalism and efficiency, even a complete but less flexible implementation might reduce the associated overhead to some extent.

Another potential limitation is that PCOM only simplifies the development but does not automate it. In order to develop a component that is robust and that provides a reasonable performance, the developer has to understand the potential changes in the component lifecycle and he needs to provide suitable code that reacts appropriately. A similar argument can be made about the mechanisms that automate the migration of component-specific state as well as the monitoring and signaling mechanisms. Although, it is often straight-forward to develop a component instance whose state can be migrated, developing an instance whose migration can be done quickly requires thoughtful use of the mechanisms. As an example consider the dependency between the *Remote Viewer* and the *Image Viewer*. Although one may restore the state of the *Image Viewer* by replaying the sequence of images that have been displayed, it is actually sufficient to restore the state using the last image. Thus, in order to minimize the overhead a developer can manually clear the message log when the next image is transmitted.

Finally, another potential limitation of PCOM is its static application model. As discussed in Chapter 2, PCOM has been developed specifically to support applications that require a certain set of functionalities during the execution of an application. As a consequence, unresolved dependencies are immediately triggering an adaptation and if a dependency cannot be resolved at all, the execution of the application is stopped. This makes the application model ill-suited to support cases in which the application makes use of optional dependencies. Of course, it is possible to integrate such optional dependencies by means of alternative contracts or user preferences. From a broader perspective, however, some of the mechanisms and algorithms in PCOM would have to be extended or at least fine-tuned to integrate optional dependencies well. To clarify this, consider that an adaptation algorithm should probably not replace a required component of the configuration to bind an optional component. Similarly, it might be often undesirable to search though all possible configurations systematically just to resolve an optional dependency that is not absolutely necessary. Thus, we would argue that for a suitable integration, we would have to reconsider the tradeoffs made with respect to configuration and adaptation. However, it should be clear that this does not affect the required dependencies.

6.1.3 Discussion

The previous description of the overheads, benefits and limitations of PCOM clearly indicates that introducing an additional layer of system software not a free. Even though PCOM does not introduce complicated abstractions, the prototypical implementation approximately doubles the size of BASE. However, our exemplary components and applications indicate that the abstractions are indeed needed and useful. As a consequence, the overhead introduced by PCOM cannot be avoided by implementing applications directly on top of BASE. This would solely shift the overhead but not reduce it. In fact, if multiple components require the abstractions, an application-specific implementation is likely to increase the overheads. In addition, some parts of PCOM such as the configuration and adaptation algorithm are hard to implement at an application level since they require the coordinated interaction of all parts of the application. The exemplary applications leave little doubt that PCOM can simplify the development of applications but developing a robust and efficient component still requires thoughtful implementations. As indicated in the previous section, a detailed and more meaningful analysis would require a structured comparison with multiple groups of developers, which is beyond the scope of this dissertation. Thus, we continue with the evaluation of the configuration algorithm and the optimization heuristics in the following two sections.

6.2 Automatic Configuration

As discussed in Chapter 3, our approach to automatic configuration is based on asynchronous backtracking. Together with the mapping introduced in the chapter, the algorithm fulfills the requirements regarding completeness, optimism and distribution by design. Furthermore, it can be extended to fulfill the requirements with respect to resilience. In this section, we evaluate how well the resulting algorithm can fulfill the last remaining requirement, namely efficiency. To do this, we first discuss factors that have an impact on the efficiency and we describe our assumptions with respect to them. Thereafter, we describe the relevant metrics to measure the efficiency. Finally, we present a number of simulations and experiments and we close the section with a discussion.

6.2.1 Influencing Factors

Asynchronous backtracking resolves unrelated conflicts simultaneously and it reconsiders only those instances that have the potential to resolve a conflict. Thus, the configuration complexity depends on the induced width, i.e. the size of sub problems that can be solved independently, and not the total width of the search space (Baker, 1995). Besides from the size of the application, the induced width of automatic configuration depends mostly on the conflict potential and the locality of conflicts, i.e. the number of instances that have conflicting requirements towards the same resources.

The potential for conflicts can be increased by creating more structurally valid configurations without increasing the number of valid configurations, e.g. by increasing the number of components that are structurally valid options but cannot be part of a valid configuration due to resource constraints. This increases the chance that the configuration algorithm randomly selects an option that leads to a conflict. The conflict locality can be decreased by creating the resource constraints in such a way that more and more components compete for the same scarce resource. Thus, when a resource conflict is detected, there will be an increasing number of alternative combinations for resolving it.

Of course, an interesting question is how these factors behave in future pervasive systems and it should be clear that there is no definite answer. However, in many pervasive systems, resource conflicts can be assumed to be relatively local. To justify this, consider that the worst-case occurs, when many instances are executed on one computer and a widely used resource (e.g. memory or processing power) is not available. However, the integration of computers into everyday objects leads to smart peer groups in which the majority of computers are specialized embedded systems. Just like everyday objects, they will be tailored towards a small number of specific purposes, which will increase the locality. The potential for conflicts itself depends on the number of available components that can be used successfully as part of the application and thus, it heavily depends on the capabilities of the smart peer group.

6.2.2 Metrics

Undoubtedly, from a user's perspective, the most relevant metric is the *total configuration delay*. That is the total time required to configure an application. Measuring this delay is absolutely unproblematic for a concrete scenario. However, relying on this metric as performance indicator exhibits several drawbacks. First and foremost, the absolute configuration delay depends heavily on the performance of the underlying hard- and software.

Secondly, it also depends on external factors such as the usage of shared resources like processing power or network bandwidth by other applications.

In order to reduce the dependencies on hard- and software or external factors, we can evaluate the performance in terms of abstract metrics. Due to the fact that the configuration algorithm is a reactive process that performs computations solely in response of incoming messages, we can use the *number of messages* as a primary indicator for the overall overhead. Yet, if the smart peer group consists of multiple computers, messages are usually processed in parallel and thus, more messages do not necessarily result in a higher configuration delay.

As an abstract indicator for the achievable parallelism in a certain setting, we can divide the configuration process into rounds consisting of message reception, message processing and message generation, i.e. within one round all messages from all computers generated in the previous round are processed by all computers which generates the messages for the next round. As a result, the *total number of rounds* provides an indicator for the best achievable configuration delay since it reflects the case that all messages can be handled in parallel.

Clearly, no pervasive system will be able to handle an arbitrarily high number of messages in parallel. Thus, solely relying on the number of rounds is too optimistic in cases where the total number of messages is high. To identify such cases, we can combine the number of rounds with the total number of messages and compute metrics such as the *average number of messages* per round. From this metric, we can then create an estimate for the latency of a round for a given system in a given scenario as a higher average number of messages will result in higher latencies in cases where the resulting amount of transmitted data exceeds the available network bandwidth.

Due to the complexity of automatic configuration and due to the requirement on completeness, the total configuration delay in a demanding scenario may be high. Obviously, it is not realistic to assume that a user is satisfied with very high configuration delays. Consequentially, we can look at the *achievable completeness with bounded overhead*. To measure the overhead, we can rely for instance on the total number of messages or in a concrete system, we can also use the total configuration delay directly.

6.2.3 Simulations

To analyze the effects of different degrees of conflict locality and different conflict probabilities for varying application sizes, we have performed an extensive set of simulations using the event-discrete simulator introduced briefly in Section 5.2.9. Since the total amount of collected raw data boils down to several hundred megabytes of information, we only show a condensed excerpt of relevant results in the following.

In order to put the simulation results into a broader context, we have implemented a greedy configuration algorithm in addition to the algorithm proposed in Chapter 3. This allows us to compare the metrics for both algorithms. The greedy configuration algorithm solely selects an assignment for a variable and if this assignment is not possible, it selects the next one until there are no further options. If a chosen assignment can be used to configure a complete sub-tree successfully, the algorithm will not modify the assignment anymore. Due to the fact that it never reconsiders successful value assignments, it never performs backtracking. Consequently, the approach is incomplete but it is also very light-weight.

To generate scenarios with varying degrees of conflict locality, conflict probabilities and application sizes, we use the following construction procedure. First, we create an application that consists of n instances with $n \geq 2$ by adding n components to a binary tree from left to right, top to bottom. Thereafter, we create one container and place the application anchor on it. For the remaining $n-1$ components, we create m containers with $1 \leq m \leq n-1$ and we distribute the components on them round-robin. During this process, we set the resource requirements of each component to one unit of one resource that is used by all components on the same container. Furthermore, we set the available amount of the resource to the number of components that are hosted on this container.

After this procedure has been completed, we have a created a configuration that resembles a binary tree of n components distributed on $m+1$containers. There is only one structurally valid way of configuring the application and this configuration is also valid with respect to resources since there are sufficient resources on each container. To increase the conflict possibility, we randomly pick k components and replicate them. Before we place them on randomly selected containers, we increase their resource requirements from one unit of the resource on the container to two units. The k replicated components increase the structurally valid configurations. However, due to the fact that the available amount of resources on all containers is not sufficient to start one component that requires two units of a resource, they can never be part of a valid configuration.

As a result, increasing the number of initial components n increases the size of the application. Furthermore, increasing the number of replicated components k will lead to a higher potential for conflicting selections during configuration, since it increases the structurally valid options without increasing the number of valid configurations. Similarly, decreasing the number of containers m will decrease the locality of the resulting conflicts. For instance, if the number of components excluding the anchor $n-1$ is equal to the number of generated containers m, value selections that are not part of the valid configuration can be identified immediately. The reason for this is that every container is only equipped with a single resource and each replicated component requires at least two resources. Thus, if the resource reservation fails, it is clear that this particular component can never be used, i.e. the conflict set during backtracking will only contain one value assignment. If the number of containers is decreased to one, all components compete for the same resource and thus, whenever a conflict occurs, all possible reconfigurations need to be considered, i.e. the conflict set generated during backtracking will contain all value assignments.

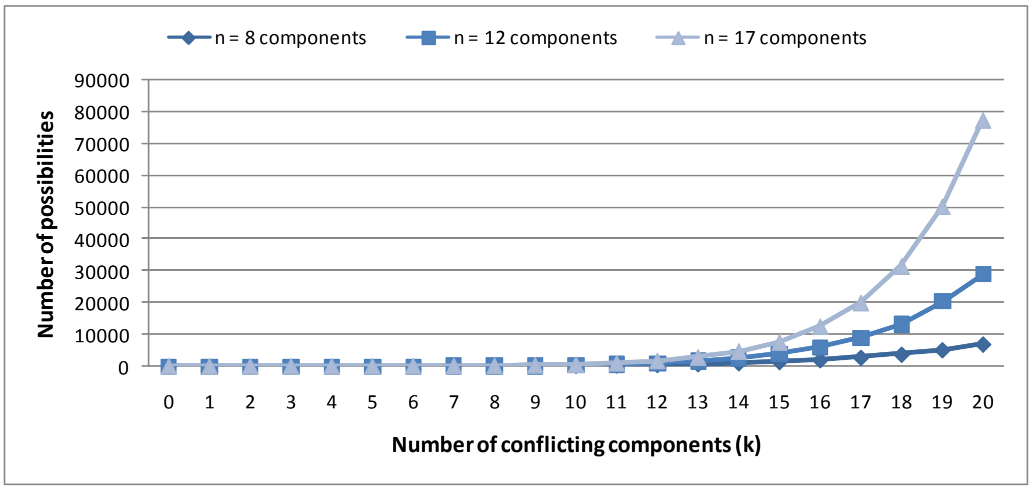


Figure 51 – Average number of structural possibilities

Independent from the settings for all parameters, there will always be exactly one valid configuration. This configuration consists solely of instances provided by the initially placed components. However, the number of structurally valid configurations increases with an increase of the number of replicated components k. Figure 51 shows the average number of structural possibilities that result from 100 randomly generated scenarios for applications with $n=8$, $n=12$, and $n=17$ components and $k=1..20$ replicated components. For example, with $n=17$ and $k=20$ there are almost 80000 structurally valid configurations on average. The exponential growth in the number of structural possibilities can be attributed to the tree structure of applications, i.e. an increase of possibilities in one sub-tree has multiplicative effects on other sub-trees. The high factor of the exponential growth can be attributed to the way conflicting components are introduced, i.e. by replicating existing components. If a component is duplicated and added, it can reuse all existing combinations of child components.

To analyze the effects of an increasing conflict probability, we measured the performance of the configuration algorithm in two different scenarios. In the first, conflicts are comparatively local. In the second, the assumption of locality does not hold. For the first scenario, we dynamically adapt the number of containers to the size of the application by setting $m=(n-1)/2$. As a consequence, conflict sets usually consist of 2-3 value assignments. For the second one, we statically set the number of containers to $m=4$ which results in larger conflict sets as the size of the application increases. The following figures refer to these scenarios as *Locality* and *No Locality*. Aggregate values are a result of 100 simulations. Although, we have performed the measurements for a broad spectrum of application sizes, we restrict the figures to $n = 8$, $n = 12$ and $n = 17$. This allows us to show the tendency and at the same time it avoids cluttering.

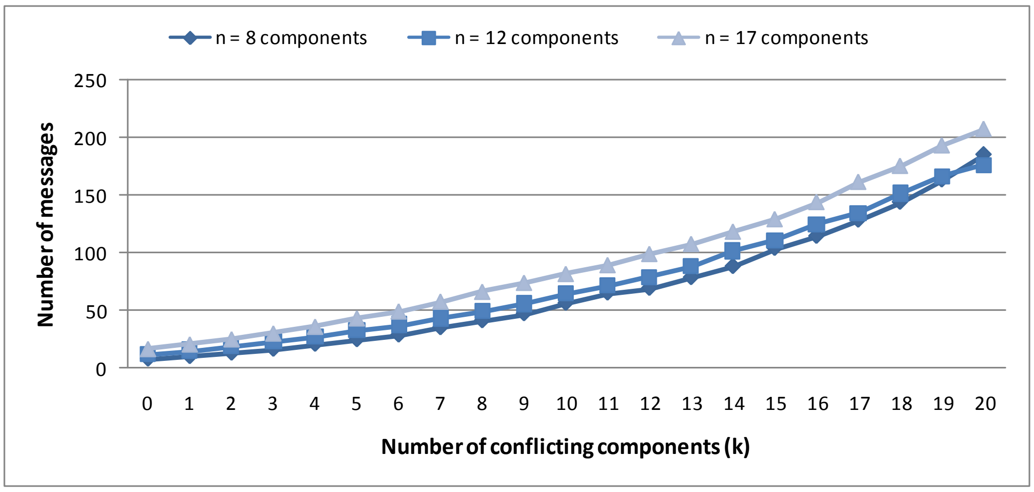


Figure 52 – Average Number of Messages (Locality)

Figure 52 shows the average number of messages for the scenario in which conflicts exhibit locality. For a medium-sized application consisting of 8-12 components, the average number of messages required to find the configuration stays well below 200, even if we introduce 20 replicated components. Additionally, the figure also already indicates the exponential increase as the number of replicated components increases.

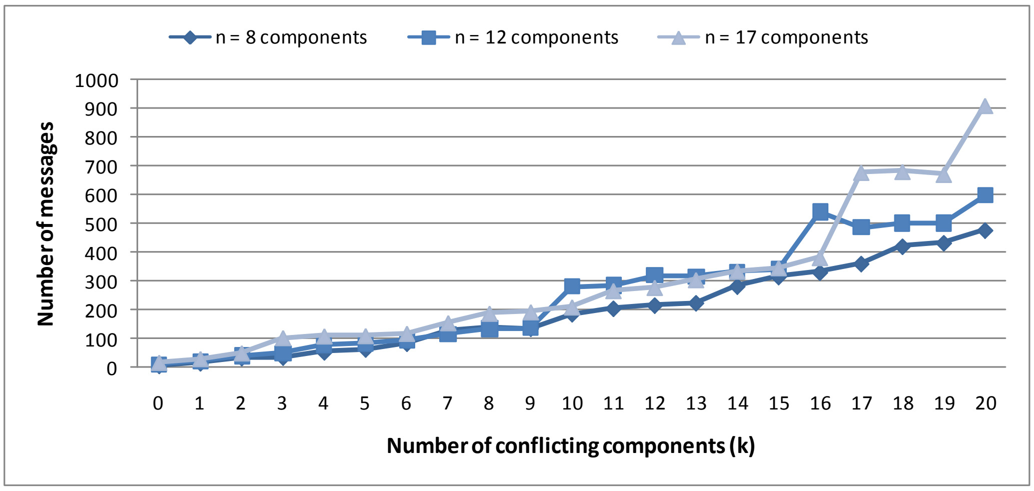


Figure 53 – Maximum Number of Messages (Locality)

However, the exponential increase becomes more apparent, when looking at the maximum number of messages as shown in Figure 53. Especially, when looking at the series of simulations with the application consisting of 17 components. There, the maximum number of messages reaches 900 when the number of replicated components is increased to 20. Depending on the underlying hard- and software, such a high number of messages may lead to significant configuration delays. Yet, when comparing the average and the maximum numbers, it is clear that such high numbers are an exception in this scenario.

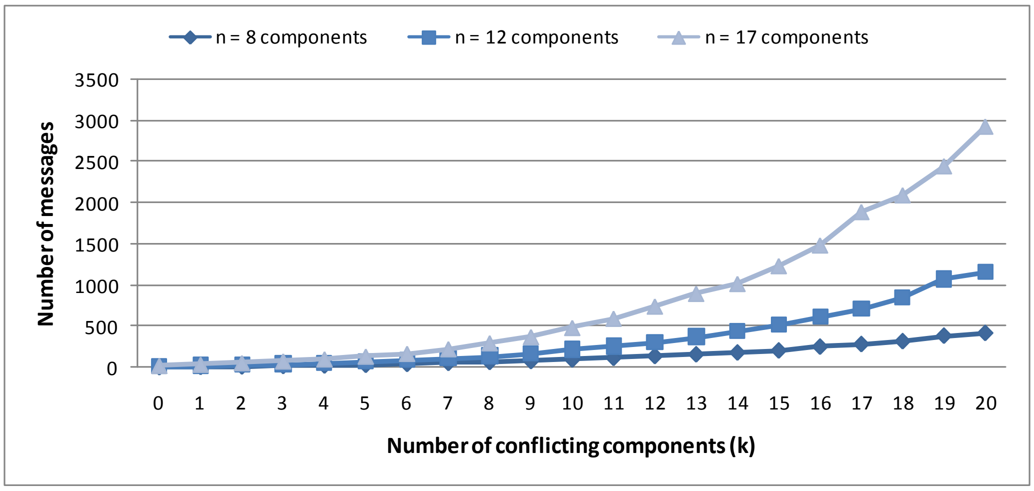


Figure 54 – Average Number of Messages (No Locality)

When we compare the average and the maximum number of messages in the scenario that exhibits locality (Figure 52 and Figure 53) with the same metrics for the scenario that exhibits low locality (Figure 54 and Figure 55), we can see a drastic increase. This increase can easily be explained. Without locality, the configuration algorithm cannot create suitable restrictions on the conflict set. As a consequence, it can only start enumerating all possible configurations. Due to the fact that the scenario generation results in a high number of structurally valid configurations, this enumeration is extremely costly.

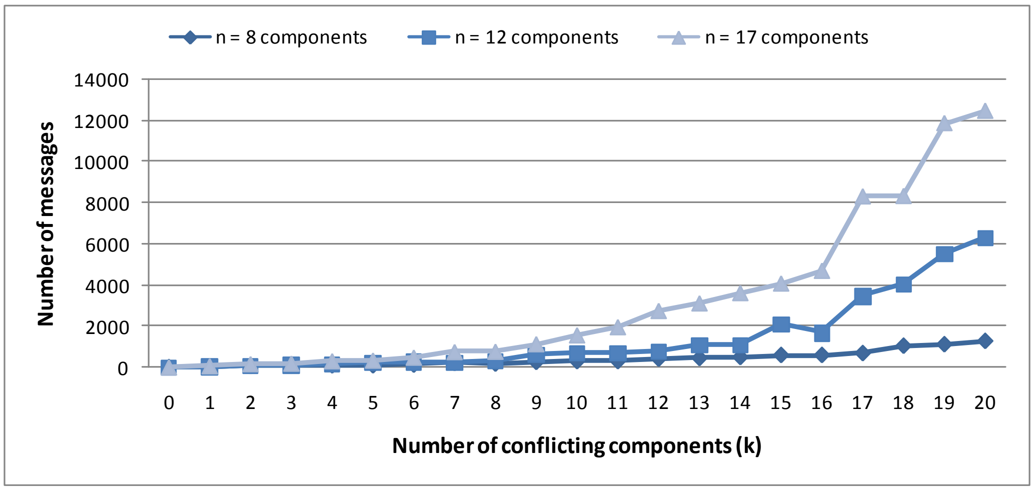


Figure 55 – Maximum Number of Messages (No Locality)

Yet, in many smart peer groups it is possible to utilize parallelism to reduce the resulting configuration delay. This can be seen by looking the average number of rounds which is depicted in Figure 56. The comparison of the average number of rounds with the average number of messages shown in Figure 54 indicates that is possible to achieve very high speed ups. Clearly, the exact speedup depends on the concrete structure of the search space and the underlying hard- and software.

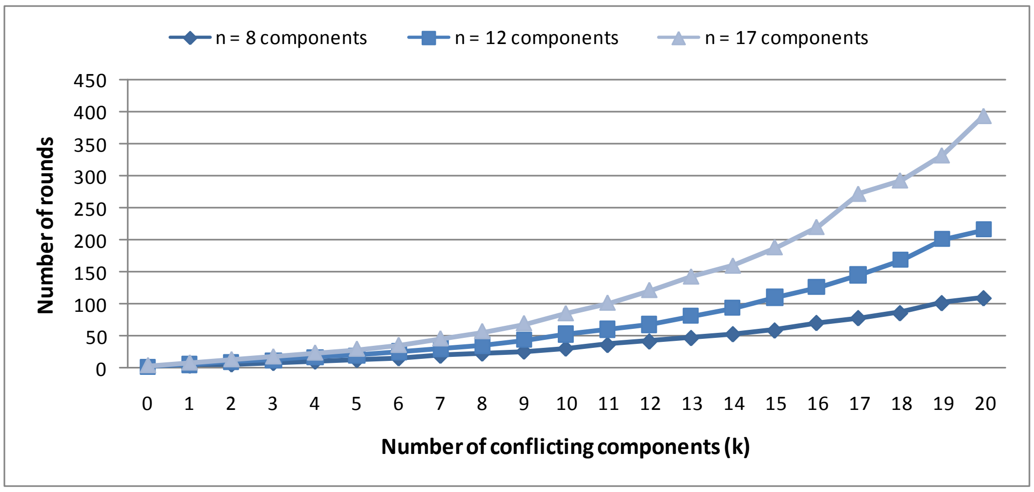


Figure 56 – Average Number of Rounds (No Locality)

Given these overheads, one might argue that it would be better to use a heuristic approach to automatic configuration. In order to show that this is not the case in general, we have compared the achievable completeness with a bounded amount of messages with the completeness that can be achieved by the greedy heuristic. As explained earlier, this heuristic does not perform backtracking and thus, it will fail in cases where it does not select appropriate components that can be used as part of a valid configuration.

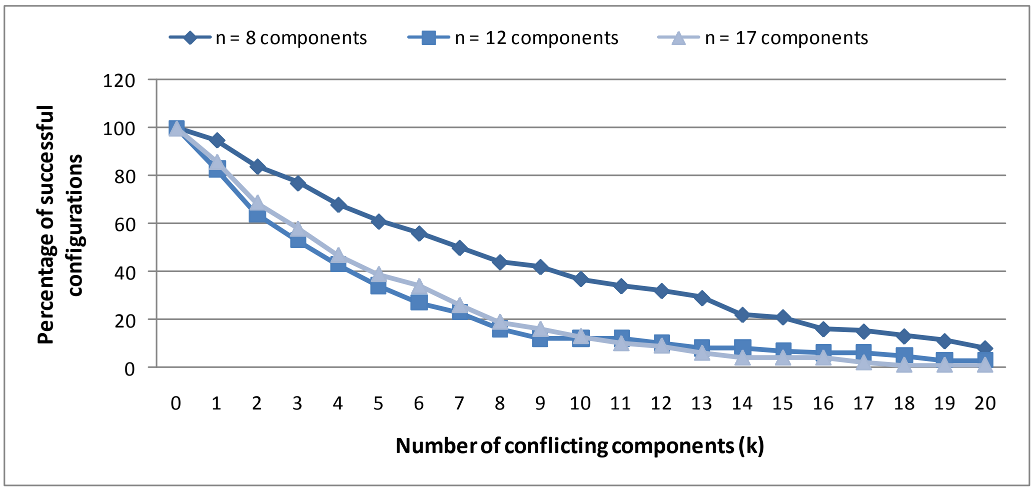


Figure 57 – Achievable Greedy Completeness (Locality)

Figure 57 shows the achieved completeness of the greedy heuristic for different application sizes in a scenario with locality. When computing a configuration, the heuristic approach typically requires less than 100 messages. Yet, in contrast to the complete algorithm, it also fails frequently and after introducing more than 8 replicated components, the achievable completeness drops below 50%, even for small applications with few components.

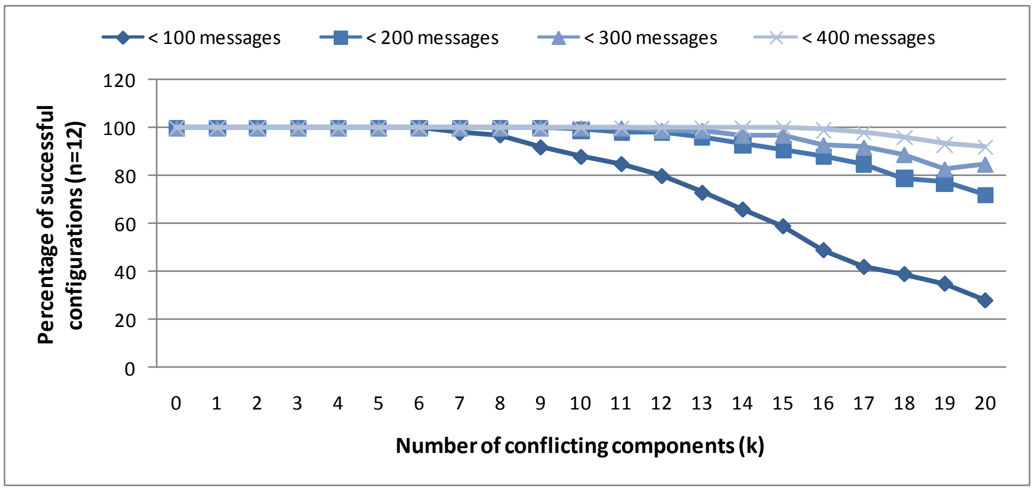


Figure 58 – Achievable Backtracking Completeness (Locality)

If we perform the same experiment with the proposed configuration algorithm, we can see a different picture. Figure 58 shows the achievable completeness for an application that consists of twelve components when the number of messages is limited to 100, 200, 300, and 400 messages, respectively. In contrast to the heuristic algorithm, the complete algorithm can still find the configuration in more than 90% of the runs for 8 replicated components, even if number of messages is limited to 100.

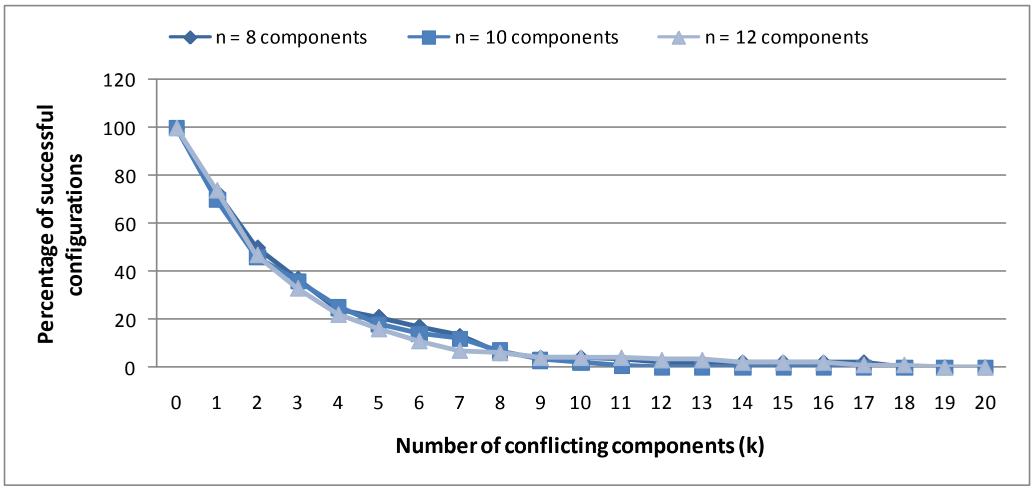


Figure 59 – Achievable Greedy Completeness (No Locality)

If we measure the greedy completeness in a scenario where the locality assumption does not hold, the success rates drop even further. As shown in Figure 59, with 8 replicated components, the greedy heuristic can only find the valid configuration in ten percent of the scenarios. The reason for this is that without locality, the greedy heuristic must perform multiple flawless decisions. This becomes increasingly improbable with an increasing conflict probability. Clearly, the same argument holds true for the complete algorithm. However, the complete algorithm can apply backtracking to correct minor flaws.

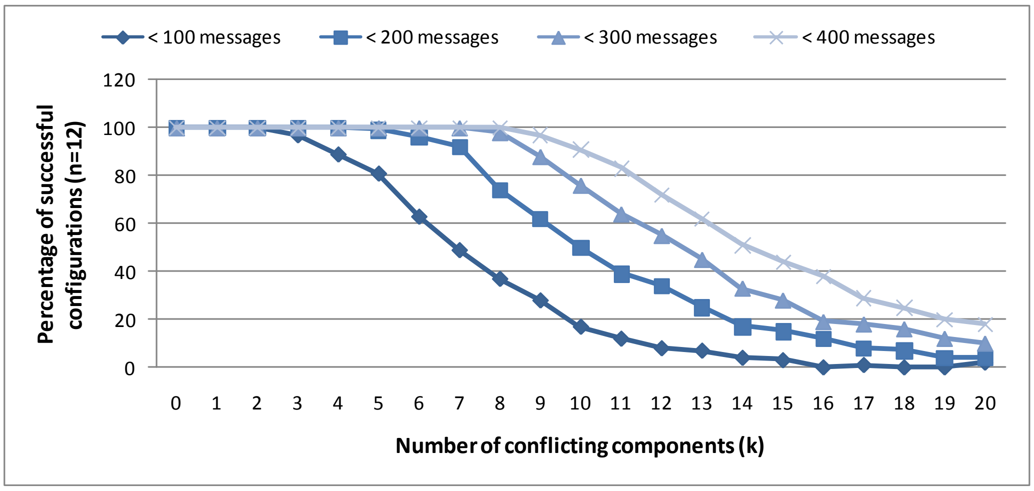


Figure 60 – Achievable Backtracking Completeness (No Locality)

Figure 60 demonstrates this behavior by depicting the achievable completeness for an application that consists of 12 components. In comparison to Figure 58, the achievable completeness with a limited number of messages drops faster. However, when limited to 100 messages, the algorithm is still capable of succeeding in almost forty percent of the cases with 8 replicated components.

6.2.4 Experiments

To validate the results of the simulations, we have performed an additional set of experiments with the prototypical implementation described in the previous chapter. To provide meaningful values for resource-poor computers, we placed an application with 7 components on 2 personal digital assistants (XScale PAX270) connected via a wireless network (IEEE 802.11b) using the procedure described in the previous section. Since all components were using the same resource on each of the component containers on the personal digital assistants, this experiment reflects a situation where the locality of conflicts is low.

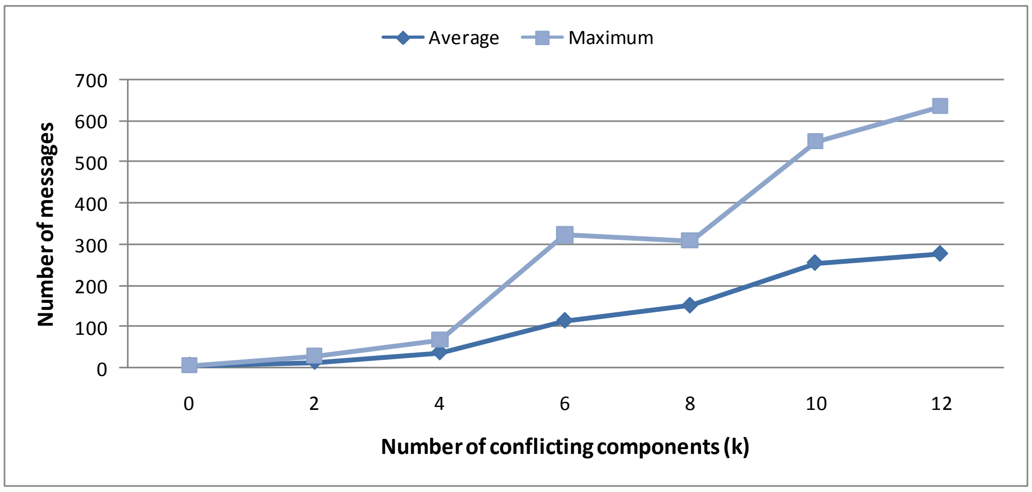


Figure 61 – Number of Messages (n=7, m=2)

We ran 7 experiments with 0, 2, 4, 6, 8, 10 and 12 randomly created conflicting components. Due to the fact that setting up the experiments is more complicated, we have reduced the number of runs from 100 per value to 10. Figure 61 and Figure 62 show the results of these experiments with respect average and maximum number of messages as well as the achievable completeness within bounded delays. The delay as well as the number of messages also includes the overhead introduced by the distributed termination detection protocol.

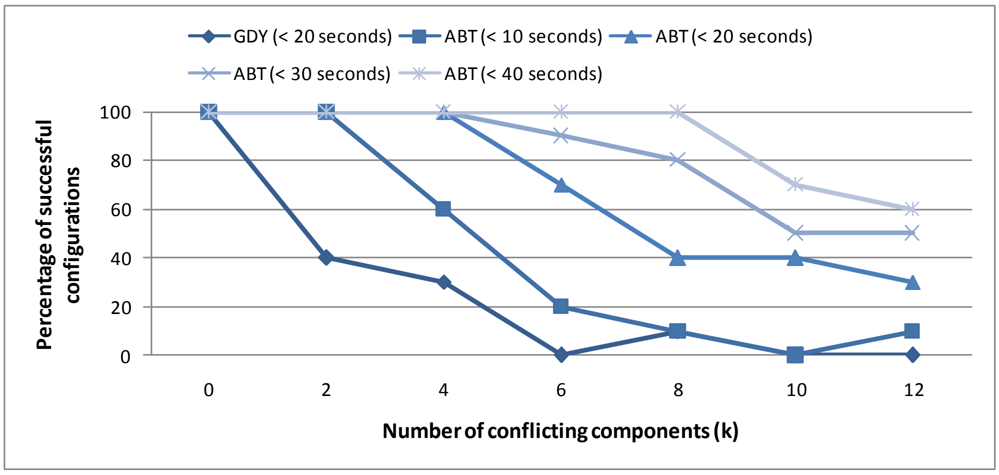


Figure 62 – Achievable Completeness (n=7, m=2)

Our experiments show that the completeness of the backtracking algorithm that can be achieved with bounded delay is always higher, even if the delay is limited to 10 seconds. Note that this is only slightly higher than the average runtime of the greedy algorithm which lies between 8 and 9 seconds.

6.2.5 Discussion

Our simulations and experiments show that automatic configuration is a complex problem and solving this problem in a satisfying way is not an easy task. Due to the fact that our configuration algorithm is complete, it exhibits exponential overhead. This is especially problematic in scenarios that exhibit a high conflict probability and a low locality of conflicts. In these cases the optimizations of the configuration algorithm become less effective and the algorithm has no other chance then to enumerate the possible solutions. If the conflict locality is high or if the conflict probability is low, the optimizations make our approach viable. We can assume that in many future pervasive systems, the conflict locality will be high but there may be a remaining set of cases in which the high locality assumption does not hold.

Given that an average user can tolerate unresponsive systems for approximately 10 - 15 seconds (Testa & Dearie, 1974) without becoming totally distracted from a problem-solving task, our comparison with a greedy heuristic shows that our proposed approach is also preferable in such scenarios. The primary reason for this is that although being fast, the greedy heuristic will often fail to find a valid configuration. In a relevant set of scenarios, a small amount of backtracking can significantly improve the achievable completeness. Besides that, our proposed approach also enables a user to tradeoff configuration delay and completeness. Thus, if a user is willing to wait, the complete approach can find a solution eventually, if it exists.

6.3 Automatic Adaptation

As discussed in Chapter 4, our approach for automatic adaptation is based on a value-ordering and a variable-ordering heuristic. Since the overhead for computing these heuristics can be neglected and due to the fact that the approach does not introduce additional messages when compared with configuration, the overall approach for automatic adaptation exhibits the same properties than the approach for configuration with respect to efficiency. As a consequence, we evaluate the approach with respect to optimality. To do this, we first discuss the influencing factors and we introduce the relevant metrics to measure their impact. Thereafter, we present the results of a set of event-discrete simulations and we close the section with a discussion.

6.3.1 Influencing Factors

Due to the fact that our adaptation heuristics are greedy, they are susceptible to the starting point of adaptation. This problem can be neglected in scenarios that support only few possible adaptations and in scenarios where all possible adaptations exhibit similar costs. In other scenarios, the achievable optimality depends primarily on the effect of the local greedy decisions on the global adaptation cost. If the local costs provide a good estimate for the resulting global costs, the heuristics are effective. If the local costs do not approximate the resulting global cost, the heuristics compute sub-optimal adaptations.

Besides from rather obscure cases in which an expensive replacement is less expensive than a series of cheaper replacements due to resource constraints, the local and global costs can differ significantly if different parameterizations of one component require completely different sub-trees. Thus, in order to create scenarios with varying effects on the achievable optimality, we can vary the number of parameterizations that do not support the reuse of existing components. By increasing the number of such parameterizations it becomes more and more likely that the heuristics select one of them leading to less optimal adaptation costs.

6.3.2 Metrics

A simplistic metric to measure the suitability of our adaptation heuristics is the total adaptation cost. However, without additional information on the scenario, this metric is not meaningful and does not allow us to draw conclusions across different scenarios. To avoid this, we must put the total adaptation cost into the context of the scenario. This can be done easily by normalizing the cost using the optimal and the worst costs of a given scenario. On top of normalization, we can then compute the *normalized distance to the optimum solution* to get a meaningful metric that is independent from the concrete scenario. If the normalized distance is close to zero, the computed solution is close to the optimum of the scenario. If the normalized distance is close to one, the computed solution is close to the worst-case.

6.3.3 Simulations

As basis for our simulations, we create a binary tree consisting of 15 contracts that originate from different components. This tree constitutes the running application. The cost for replacing each component is randomly initialized using a standard distribution with an average of 10 and a deviation of 10. We use this abstract metric for the state size since we are only interested in the relative differences. The assumption here is that most components will carry a similarly low amount of state, yet, there are some components that carry high amounts of state.

In order to create alternative configurations, we randomly pick d sub-trees of the application and for each, we create a additional sub-tree that can be used as its replacement. Thus, by increasing d, we create more alternative configurations. For each contract of the new sub-tree, we decide with a probability p, whether the contract is a parameterization of the corresponding existing component (i.e. it can be reused). If it is not a parameterization, we create a new component for the contract. Thus, by increasing p it becomes more likely that the d sub-trees are parameterizations as opposed to alternative components.

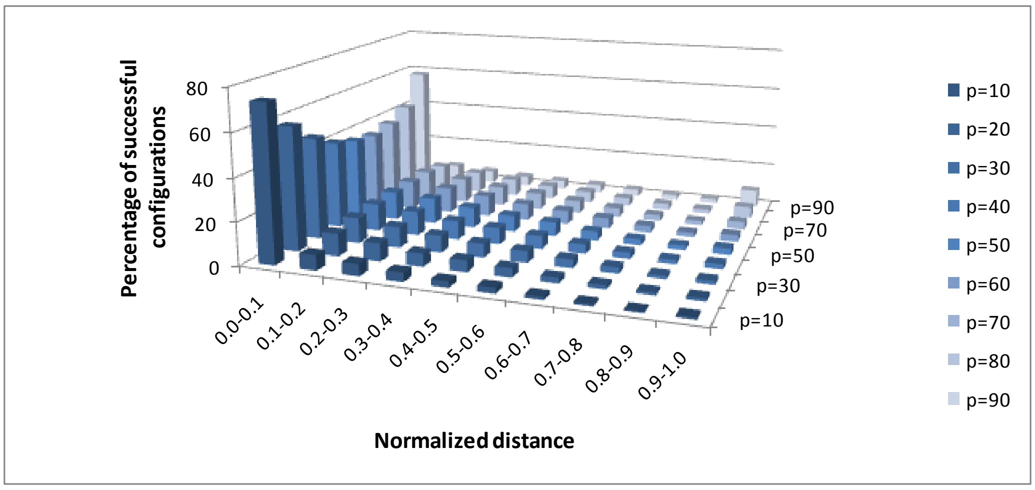


Figure 63 – 1st Run Solution Quality (d=20)

The resource requirements are initialized randomly such that each component requires 1 and 10 instances of a single abstract resource type. Then we create 4 containers on which we place the components randomly. Each container provides 30 resources. Finally, we increase the resource requirements of one of the contracts of the original application to 31 to simulate that the chosen configuration can no longer be executed. In order to get representative results despite the randomization, each measurement is based on 10000 simulations.

Figure 63 shows nine histograms of solutions grouped into 10 categories according to their solution quality for 20 duplicated sub-trees $d = 20$ and varying parameterization probabilities $p = 20..80$. Note that these histograms denote the solutions found after the first run of the algorithm. For this parameterization probability, the heuristics are able to find a solution that has a maximum normalized distance of 0.1 from the optimum in 58% of the simulations. Intuitively, if the probability is increased, the solution quality becomes worse up to a certain point where it increases again. The reason for this lies in the fact that if the probability is relatively low, there are only few parameterizations and, thus, the value-ordering heuristic works effectively. If the probability is high, most duplicated sub-trees will not inflict any costs and, thus, each parameterization leads to a similar (high) quality. If the parameterization probability lies around 50%, the scenario is likely to exhibit a number of parameterizations whose selection will indirectly introduce costs since they lead to the replacement of some child instance. Thus, the value-ordering becomes less effective since the local costs are not longer a good indicator for the resulting global cost.

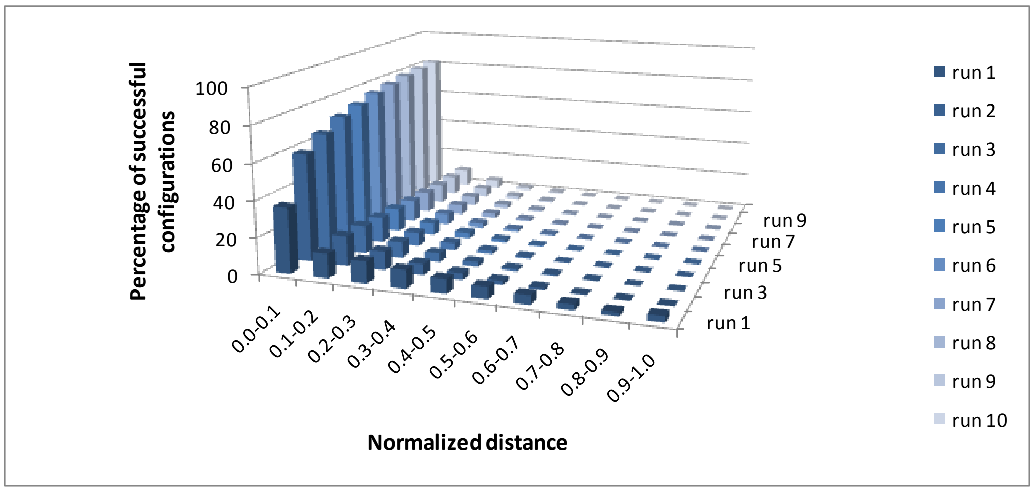


Figure 64 – Nth Run Solution Quality (d=20, p=60)

To improve the quality of the solutions, we may execute the algorithm iteratively starting with a random variable assignment on each run. After each run the best configuration is used. The resulting solution quality after a varying number of runs for $d = 20$ and $p = 60$ is shown in Figure 64. The figure indicates that the quality can be increased significantly by running the algorithm a second time. Since the overhead in these simulated scenarios is moderate (approximately 100 messages per run), a second execution is likely to be worthwhile in cases where the first solution exhibits high costs.

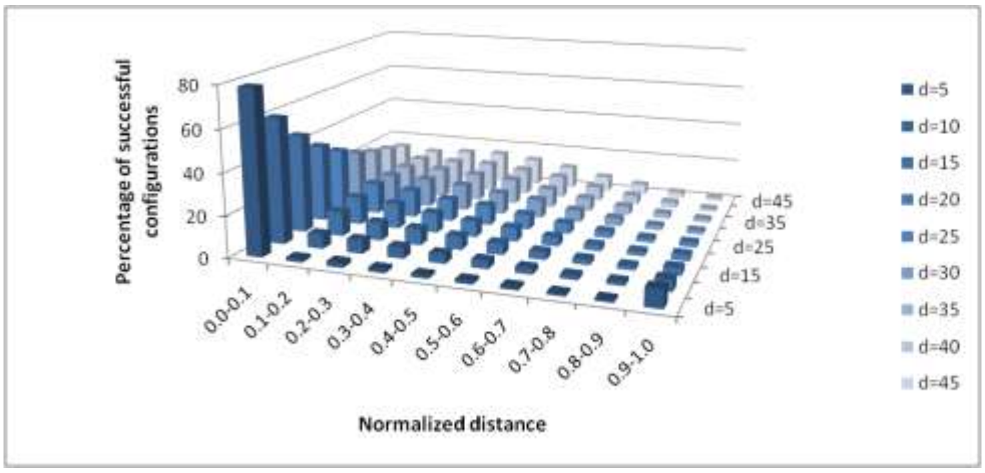


Figure 65 – 1st Run Solution Quality (p=50)

Finally, in order to measure the effects of a changing number of possible solutions, we have varied the number of duplicated sub-trees for a fixed parameterization probability $p = 50$. Figure 65 shows that the solution quality is reduced if the number of duplicated sub-trees is raised. This is most likely a result of the fact that the relative number of solutions with average cost increases disproportionately high. Again, this can be mitigated by executing multiple runs.

However, in real world scenarios, the number of parameterizations supported by one component will be limited since each parameterization needs to be programmed and tested.

6.3.4 Discussion

As indicated by the simulation results presented previously, the value- and variable-ordering adaptation heuristics are an effective and light-weight approach that can achieve good results in a broad spectrum of scenarios. In scenarios, where the resulting adaptation cost after the first run is not satisfactory, it is possible to utilize randomization to improve the result. However, it must be noted that the second run is likely to double the configuration delay. As a result, this is only a viable option in cases where the adaptation costs are high. Consequently, a simple strategy to determine whether another randomized run is worthwhile would only perform a second run if the current cost is significantly higher than the cost resulting from doubling the search latency. Given the overall results, we argue that this case will occur only rarely.

7 Related Work

This chapter presents and discusses related work. Thereby, the chapter groups related approaches according to characteristics of the targeted system. As basis, the chapter provides a brief review of system-support for conventional applications. Thereafter, the chapter discusses various approaches to support pervasive applications. Finally, the chapter closes with a discussion that highlights the differences between the existing systems and the approach described in this dissertation.

7.1 System-support for Conventional Applications

Traditionally, distributed applications have been developed for static systems. Although, most systems are not completely static, e.g. they might change sometimes due to failures. The overall approach of ignoring changes works well, given that the frequency is sufficiently low and the consequences of communication and application failures are tolerable. If this is not the case, some of the abstractions proposed by system software for conventional applications can also be extended. Yet, existing system software that has not been targeted specifically at pervasive systems covers the resulting challenges only partially and fails to address them in an integrated way.

7.1.1 Communication Middleware

Communication middleware commonly refers to a layer of software that is stacked on top of a traditional network operating system to ease the development of distributed applications. Thereby, one of the main tasks of most middleware is to hide the heterogeneity of the underlying hardware, operating systems or programming languages. Existing middleware offers various communication abstractions to develop distributed applications. These abstractions range from network-oriented message passing primitives and publish-subscribe abstractions (Eugster, Felber, Guerraoui, & Kermarrec, 2003) over associative memory paradigms such as tuple spaces (Carriero & Gelernter, 1986) to remote procedure calls (Birrel & Nelson, 1984) or their object-oriented counterpart of a remote method invocation. Since the latter extends the concept of a local procedure call – or a method invocation respectively – to distributed application development, it is commonly supported by mainstream middleware.

Prominent examples of middleware that supports the remote method invocations are systems that are based on the Common Object Request Broker Architecture (CORBA) (Object Management Group, 2004). CORBA defines an Interface Definition Language (IDL) as basis for interoperability. The IDL enables application developers to explicitly describe the interfaces of their (remotely) accessible objects in a platform-independent manner. To do this, application developers must explicitly model all methods including return and parameter types. Using tools, the description can then be compiled into platform-specific code that hides most of the details of (remote) communication[2]. The generated code can then be integrated into the application in the form of so-called stubs that act as local representatives for remote objects (proxies) and abstract templates that need to be implemented by accessible objects (skeletons). In addition to IDL, interoperability protocols and language bindings, CORBA also defines a number of middleware services such as naming and trading services that are used to bootstrap communication. However, CORBA leaves the management of the dependencies between the objects that constitute a distributed application to the application developer.

Nevertheless, the general approach of CORBA towards enabling interoperability has proven to be successful. The basic concepts found their way into popular programming frameworks such as the Java 2 Standard Edition in the form of Java RMI (Sun Microsystems, 2004) and the .NET Framework as .NET Remoting (Chappell, 2002). The concrete implementations exhibit notable differences when compared with CORBA since both frameworks are based on a standardized virtual machine that provides a homogeneous platform. Due to this fact, Java RMI and .NET Remoting can simplify the task of defining a remotely accessible interface by reusing the language specific interfaces available in the Java programming language and the Microsoft Intermediate Language respectively. In addition to that, they can also provide advanced features such as automatic stub generation on application load time and distributed garbage collection. To enable interoperability with legacy computers and applications, both frameworks can be configured to support various interoperability protocols as well. Java RMI, for instance, can be configured to generate stubs that communicate with IIOP, i.e. the interoperability protocol used by CORBA. Yet, using this configuration sacrifices advanced RMI-specific features such as distributed garbage collection.

Apart from programming frameworks, many fundamental concepts of CORBA can also be found in current web technologies such as web services (World Wide Web Consortium, 2002). Similar to the Object Management Group that defined CORBA, the World Wide Web consortium introduces standards that describe interfaces and data types (World Wide Web Consortium, 2007) as well as protocols (World Wide Web Consortium, 2007). In contrast to CORBA, many standards are based on the Extensible Markup Language (XML) which has become a widely used language for data exchange on the Internet. From a conceptual point of view, however, there are only minor differences between remote objects in CORBA and web services.

In addition to these mainstream developments, there exists a body of research on various specialized areas of system-support with communication middleware. These include for instance

[2] One aspect that is usually not hidden from the application developer is handling of network and node failures. In many implementations, such failures are signaled by a failure-specific type of exception.

minimum middleware for resource-poor computers such as minimum CORBA (Object Management Group, 2002) or the PalmORB (Roman, Singhai, Carvalho, Hess, & Campbell, 1999) that restrict the functionality provided by the middleware. Further examples are configurable middleware systems that can be adapted to the capabilities of the computer, e.g. (Roman, Kon, & Campbell, 2001), and reflective middleware systems (Blair, Coulson, Robin, & Papathomas, 1998) that can adapt their internal mechanisms and protocols dynamically.

While the previous examples for communication middleware are primarily targeted at static systems, Jini (Sun Microsystems, 2001) is a Java-based network technology that enables the utilization of distributed services in dynamic computer networks. As such, it provides a number of basic communication-related services. These include service discovery to detect the services that are available in an environment and leases to avoid orphaned client state. With respect to its goals, Jini is similar to Universal Plug and Play (Microsoft Cooperation, 2000) which also supports the dynamic detection of the services that are available in an environment. In contrast to Universal Plug and Play which uses XML-based protocols to facilitate interaction, Jini relies on mobile code to distribute service-specific proxies to clients. Both, Universal Plug and Play and Jini, provide basic lookup and signaling functionality that enables the development of distributed applications in dynamic settings. Yet, communication failures resulting from disconnections must be handled programmatically by the application developer.

To avoid application-level failure handling despite temporary disconnections, communication middleware can utilize persistent message queues. On top of these queues, it is possible to implement higher abstractions like queued remote procedure calls as done by the Rover Toolkit (Joseph, Tauber, & Kaashoek, 1997), for example. By storing requests and responses in persistent queues, a client and a server can be disconnected temporarily at any point in time and both can continue their interaction once they are connected again. Yet, the application of queued remote procedure calls is only suitable in cases where very high latencies can be tolerated and they require that the communication partners meet eventually to finish their interaction. These assumptions can decrease the applicability of the approach, in general.

7.1.2 Component Systems

A potential shortcoming of communication middleware is a lack of defined application structure which may limit the reusability of individual application parts. Component systems try to mitigate this by introducing the notion of a software component as a structuring element. The basic idea of software components is to facilitate the reusability of application logic by dividing it into smaller parts that can be composed in various ways. Thus, software components are essentially an analogy to hardware components which can be assembled differently to suit various purposes. Yet, in contrast to hardware, software is an immaterial artifact that does not have to follow the laws of physics. As a result, there exists a higher flexibility in defining abstractions and this flexibility has led to various competing definitions of the term of software component (Szyperski, 1997). Some of these definitions are so weak that the some of the previously discussed middleware systems may be classified as component systems as well.

Besides from spawning multiple definitions, the flexibility of software has also led to a plethora of component abstractions that vary significantly depending on the goals of their designers. The JavaBeans component model (Sun Microsystems, 1997), for instance, is primarily targeted at the visual construction of applications. Thus, it is mostly concerned about defining components in such a way that they can be easily used by visual builder tools. In contrast to that, the COM[3] component model (Microsoft Corporation, 1995) defines software components in such a way that they can be replaced easily, that they can evolve independently, and that different implementations can coexist peacefully on the same computer. As a result, it is mostly concerned about defining identification schemes for interfaces and implementations. The service platform defined by the Open Service Gateway Initiative (OSGi Alliance, 2007) has a similar focus but it uses a slightly different approach. To support the evolution of code on a single computer, the service platform relies on a component abstraction called bundle that defines a lifecycle and supports the specification of dependencies between different bundles. Bundles may import and export code directly and they may provide and use each other's services. A framework provides the required supportive functionality such as lifecycle and dependency management or service registration and lookup. In addition to specifying the component abstraction and the framework interface, the Open Service Gateway Initiative also standardizes the interfaces of frequently used services (OSGi Alliance, 2007) and it defines interfaces with communication middleware such as Universal Plug and Play.

Interestingly from a high-level point of view, such extreme differences are not present in mainstream distributed component systems, e.g. DCOM/COM+ (Microsoft Corporation, 1996), (Chung, et al., 1998), (Chappell, 2002), the CORBA Component Model (Object Management Group, 2006) and Enterprise JavaBeans (Roman, Siganesh, & Brose, 2005). A potential reason for that is that they are all targeted at enterprise applications. Thus, even though the concrete abstractions vary to some degree they feature similar services to support the component and application development. Typically, these services include:

- Object pooling and just in time activation: A component may require a considerable amount of resources. Thus, for some components it might make sense to active them only in cases where they are actually accessed. Similarly, it might be beneficial to pool component

[3] Note that COM is also used as an umbrella term for other technologies such as OLE, ActiveX, DCOM and COM+. Here, we refer to the original version of the Component Object Model developed in 1993.

instances to reduce the resource consumption. As a result, many component systems are readily equipped with such mechanisms and they foresee them in their component model.

- Load balancing and clustering: Since business applications are frequently used simultaneously by large numbers of users, a single computer may not be able to handle all requests. To increase the scalability of applications, many distributed component systems provide mechanisms for load balancing or clustering on a set of computers.
- Authentication and access control: Many business applications manipulate business data that may only be accessed by privileged personnel. As a result, most component systems provide mechanisms to authenticate requests. The authenticated requests are then accepted or denied on the basis of access control specifications provided by a developer or an administrator.
- Distributed transactions: Since data integrity and consistency is essential for business applications, most component systems are able to support distributed transactions. In many cases, performing a transactional rollback is the only way of dealing with failures that appear during the execution of an application.
- Message queuing: Besides support for distributed transactions, some component systems are relying on asynchronous message delivery using message queues to mask the temporary unavailability of other computers. However, obviously such mechanisms can only be applied in cases where the result of a request is not required in a timely manner.

As one might guess from the set of services that can be used to deal with failures, mainstream distributed component systems are mainly targeted at execution environments that exhibit transient failures. If the response to a message is not needed immediately, such failures can be easily masked by decoupling the sender and the receiver using persistent message queues. If a response is needed, distributed transactions can be used to ensure that a certain set of operations is either executed successfully or not at all. As explained earlier, the failures and disconnections that arise in a smart peer group may be permanent and thus, such mechanisms should not or cannot be applied there to deal with failures in a generic manner.

Besides from mainstream component systems that focus on enterprise applications, there are also a number of distributed component systems that specifically focus on the automatic configuration of distributed applications. Many of these systems are targeted specifically towards multimedia applications (Rothermel, Barth, & Helbig, 1994), (Dermler, 1999), although some of them have been used for other applications as well (Kon, 2000). Besides from introducing application-specific component models, these systems typically rely on a centralized computer for configuration. Given that multimedia applications often require resource-intensive transformations such as on-the-fly stream transcoding, this is a clearly viable approach. Yet, in smart peer groups the availability of such a powerful computer cannot be guaranteed and thus, the approaches taken by these component systems cannot be applied. In addition, most systems focus on the initial configuration of the application, e.g. to support different sets of clients that display the multimedia content or to support mixing of different media sources at runtime, and thus, they do not support the automatic adaptation of the configuration at runtime.

7.2 System-support for Pervasive Systems

Existing system-support for pervasive systems can be broadly classified into systems that aim at the integrated usage of computers in a physically restricted space, i.e. smart environments, and systems that aim at the integration of computers that are in physical proximity, i.e. smart peer groups. The first class of systems usually relies on a centralized server that takes care of the fundamental management tasks. Examples of such tasks are discovery, mediated communication and access control. The second class of systems does not rely on a centralized computer. Instead, they organize their management tasks in a distributed fashion. For both classes of systems, the support for application developers and the degree of automation depend heavily on the underlying application model. In the following, we provide an overview over both classes and we discuss a selection of relevant representatives of each class in greater detail.

7.2.1 Smart Environments

A smart environment is a spatially limited area, e.g. a meeting room or an apartment, equipped with various sensors, actuators and computers. In order to simplify application development, system software for smart environments typically provides a set of basic services for the applications executed in their spatial area. Common services include authentication of computers, access control to resources, unified access to persistent storage, and management of data captured by sensors of the environment. To provide further support for application development and administration, some systems additionally introduce abstractions used to structure an application. These range from basic service abstractions that enable the transparent usage of different implementations of the same functionality within an application up to comparatively complex component abstractions that support automated application adaptation. However, usually these systems rely on the permanent availability of a comparatively powerful computer which makes them ill-suited for smart peer groups. In the following, we discuss five major approaches towards enabling smart environments.

7.2.1.1 IROS

IROS (Ponnekanti, Johanson, Kiciman, & Fox, 2003) is a meta operating system for intelligent rooms that has been developed as part of the Interactive Workspaces project (Johanson, Fox, & Winograd, 2002) at the University of Stanford. It is mainly targeted at distributed applications that support collaborative work. The overall application model of IROS is based around the idea of enabling the simultaneous and coordinated use of interactive legacy applications on multiple computers in a single room.

Applications in IROS consist of independent entities that are only loosely coupled. These entities can be programs running on general purpose computers, e.g. an instance of a web browser running on a PDA, as well as appliances, i.e. specialized embedded systems such as a video projector that is capable of displaying images. In order to coordinate their behavior, entities may exchange messages and they may cooperatively use a globally shared storage. To support this application model, IROS introduces three different abstractions, namely ICrafter to control the individual entities, the EventHeap to support the message exchange between them and the DataHeap to support global storage.

ICrafter (Ponnekanti, Lee, Fox, Hanrahan, & Winograd, 2001) is a framework that provides support for user interface selection, generation and adaptation. Its main purpose is to enable users to interact with the individual entities that are available in the smart environment. To do this, each entity describes its capabilities and it continuously announces its presence using beacons. Using the announcements, ICrafter can detect the entities that are usable in an environment and using the description of the capabilities, it can synthesize a user interface. Optionally, each entity may also provide its own user interface which can be integrated by ICrafter. By combining these mechanisms, ICrafter can be used to remote control programs and appliances.

Besides from controlling individual entities using ICrafter, IROS also supports the coordinated utilization of a number of entities. To do this, it provides a communication mechanism called EventHeap (Johanson & Fox, 2002), (Ballagas, Szybalski, & Fox, 2004). The EventHeap enables entities to post and to receive messages. The messages essentially resemble strongly typed tuples. Upon reception of a message, the event heap stores this message for later retrieval by interested entities. In addition, the EventHeap can also redistribute the message to interested entities that are already known. In order to avoid memory overflows in the EventHeap, messages are pruned after a certain timeout. By using the EventHeap to mediate messages, IROS essentially decouples the individual entities and it can mimic different communication paradigms, e.g. unicast, broadcast, publish-subscribe, etc.

The EventHeap is mainly targeted at small messages that signal a state transition in some entity. In order to share larger amounts of data among entities, IROS provides a storage mechanism called DataHeap. The DataHeap is in many ways similar to a network file system, i.e. it stores meta information such as file type, creation time, owner, etc., and the actual content. However, beyond the capabilities of a traditional file system, the DataHeap also provides an extensible transcoding scheme using Paths (Kiciman & Fox, 2000). This allows one entity to store data in some data format and another one to retrieve the same data in some other format.

Together these mechanisms can be used to easily integrate legacy applications. To do this, the legacy application needs to be wrapped using the ICrafter framework and it can be extended with scripts to post and receive messages to announce and trigger internal state changes. Furthermore, if the application requires some data to operate on, it can use the DataHeap to store, retrieve and share it. Similarly, IROS also makes it easy to leverage varying compositions of entities since all communication is mediated through the EventHeap and through the DataHeap. Moreover, the mediated communication also allows entities to fail independently and since the messages are stored by the EventHeap, appropriately written entities may recover transparently from transient failures.

With respect to system-support for smart peer groups, the structure of IROS also exhibits three severe limitations. First of all, it requires a central computer to execute the EventHeap and the DataHeap. This alone limits the applicability of IROS to environments where the permanent presence of a single computer can be guaranteed. Secondly, the EventHeap and the DataHeap constitute single points of failures. This can be somewhat mitigated by the fact that the EventHeap of IROS can be restarted comparatively fast. However, in order to improve the efficiency, IROS does not store the messages contained in the EventHeap on persistent storage. Thus, a restart of the EventHeap might lead to inconsistencies due to lost messages. Thirdly, the coordination mechanisms in IROS are weak by design. As a result, the entities in IROS cannot rely on the presence of any other entity during their execution. Thus, one might argue that IROS makes the implicit assumption that the communication between different entities is mostly optional and solely improves the coordination.

7.2.1.2 AURA

AURA (Garlan, Siewiorek, Smailagic, & Steenkiste, 2002) is a software infrastructure for pervasive applications that has been developed at the Carnegie Mellon University. The architecture of AURA incorporates several building blocks developed originally for mobile computing applications. These building blocks include the Coda distributed file system (Satyanarayanan, 2002) to manage persistent data and the Odyssey system (Noble, Satyanarayanan, Tilton, Jason, & Walker, 1997), (Noble & Satyanarayanan, 1999) which enables adaptation to fluctuating resource availability. In addition, AURA relies on a remote execution framework called SPECTRA (Flinn, Narayanan, & Satyanaray, 2001) that enables the utilization of remote resources to optimize the performance or the resource consumption of an application. On top of these rather traditional building blocks, AURA introduces PRISM (Sousa & Garlan, 2002), a so-called task manager, to implement the concept of task-driven computing.

All building blocks of the AURA architecture are geared towards supporting adaptation at different levels. CODA provides adaptive access to persistent data that can even support disconnected operation by keeping logs and synchronizing them at a later point in time. In addition to this, Odyssey can adapt the data itself depending on the resource availability. To do this, Odyssey introduces the concept of data fidelity which can be thought of as a data type dependent tradeoff between data quality and size. For example, by using a more aggressive compression the size of an image can be reduced in order to safe bandwidth during its transmission. SPECTRA tries to optimize the local resource utilization of a mobile computer by distributing individual functions of an application based on the capabilities of the environment and the characteristics of the functions. Finally, PRISM adapts the composition of the applications to support a certain set of user tasks in different environments.

As indicated by the functionality provided by the building blocks of the AURA architecture, applications in AURA are usually not distributed, although individual functions can be distributed on demand by SPECTRA. Instead of distributing the application to use the specific functionality of an environment, AURA adapts the set of applications to support a certain set of user tasks. To do this, user tasks such as “edit a document” or “display a piece of information” are modeled in an application-independent manner. If a certain task shall be executed, PRISM determines an appropriate application, launches it and supplies it with the necessary data, e.g. the desired document or the requested piece of information. To perform the mapping from tasks to applications, PRISM relies on a centralized environment manager which represents an environment-specific application registry. If a user wants to continue a task that has been performed in another environment, PRISM picks a suitable application and CODA ensures that the necessary data is made available.

7.2.1.3 GAIA

GAIA (Roman & Campbell, 2000), (Roman, Hess, Cerqueira, Ranganathan, Campbell, & Nahrstedt, 2002) is a meta-operating system for Active Spaces that has been developed by the University of Illinois at Urbana-Champaign. In order to support communication GAIA relies on a communication middleware called Unified Object Bus (Roman & Campbell, 2001). On top of this communication middleware, GAIA provides a number of generic services to simplify the application development. These services include discovery, shared persistent storage, context management as well as component and application lifecycle management.

Applications in GAIA are usually distributed and consist of a set of components. Components are glued together using a scripting language. In order to deploy applications in different active spaces and in order to customize them for different usage scenarios, GAIA introduces two types of application descriptions, the application generic description and the application customized description. The application generic description lists the set of required components, their maximum and minimum allowed numbers as well as their requirements on the executing computer. The application customized description entails a concrete mapping of components onto the computers of an active space.

To structure the applications, GAIA proposes a model-view-controller-coordinator pattern that extends the well-known model-view-controller pattern of desktop applications (Gamma, Helm, Johnson, & Vlissides, 1995). Thereby, the model, view and controller perform the same task as in a traditional application and the coordinator takes care of managing different views and controllers. Thus, the coordinator takes care of adapting an application to changes programmatically. Thereby, it essentially modifies the initial configuration of the application that is described in the application customized description.

Besides from these basic concepts and mechanisms there are a number of extensions that improve the flexibility and fault tolerance of applications. Instead of implementing applications in a scripting language that directly refers to individual components, it is possible to program applications on a higher level using the Olympus framework (Ranganathan, Chetan, Al-Muhtadi, Campbell, & Mickunas, 2005). The framework then takes care of finding matching components and enforcing constraints that have been specified by the programmer or an administrator. Towards this end, the framework relies on an ontology for match-making and on Prolog to define and enforce rules.

To detect failures and to deal with some failures in a generic manner, GAIA relies on periodic heart-beat messages. In order to mask transient application failures, the core system of GAIA has been extended to periodically capture checkpoints of the state of an application. Using these checkpoints, the system can then automatically restore an application in cases where it does no longer transmit heart-beat messages (Chetan, Ranganathan, & Campbell, 2005).

7.2.1.4 MetaGlue

MetaGlue (Coen, Phillips, Warshawsky, Weisman, Peters, & Finin, 1999) is agent platform that has been developed for the Intelligent Room project (Brooks, 1997) at MIT. The platform provides a runtime environment for mobile agents on top of the Java virtual machine. MetaGlue extends the Java language with a number of additional constructs to simplify the development of agents. However, these constructs are translated into Java code before execution.

Apart from constructs to load and store persistent data in a platform independent manner, the most notable extensions are statements that enable agents to specify their dependencies on the execution environment. The dependencies may consist of resource requirements on the computer that executes the agent and requirements on other agents. An agent can only be executed on computers that are able to fulfill the specified resource requirements. Furthermore, the runtime system ensures that an agent is only executed if all recursively required agents are present. If a required agent is not available or if it becomes unavailable at runtime, the system defers the execution of the dependent agents until the required agent becomes available again.

Since the resources in MetaGlue represent physical resources whose availability may be strictly limited, the configuration problem in MetaGlue is similar to the one discussed in this dissertation. The main difference is that an agent in MetaGlue may be reused to satisfy multiple requests. To solve the resulting problem, MetaGlue relies on the Rascal resource manager (Gajos, 2001). The tasks of Rascal consist of resource mapping and arbitration (Gajos, Weisman, & Shrobe, 2001). That is the selection of suitable agents and the selection of computers to execute them in such a way that their resources are not overloaded. To perform mapping and arbitration in an optimal manner, Rascal supports the definition of a utility and a cost function that describe the quality of a certain match and the penalty for changing an executed set of agents. To avoid changes with negative impacts, Rascal solely performs changes that lead to an increase in the overall system utility.

Although, MetaGlue is inherently distributed due to the fact that agents may reside on different computers, Rascal has been built as a centralized resource manager. This design enables the utilization of JSolver (Chun, 1999) – a standard non-distributed constraint satisfaction engine – to compute and compare different solutions. However, at the same time, this centralized design makes Rascal also a single point of failure which is ill-suited for the application in a smart peer group.

7.2.1.5 O2S

O2S (Paluska, Pham, Saif, Chau, & Ward, 2008) is the system software that has been developed as a part of the Oxygen project at MIT. It proposes a programming paradigm called goal-oriented programming (Saif, Pham, Paluska, Waterman, Terman, & Ward, 2003). From a non-technical perspective, a Goal represents a user requirement that must be fulfilled by the system. Often times such a user requirement can be decomposed hierarchically into a set of simpler goals whose simultaneous fulfillment will satisfy the initial high-level goal.

Technically speaking, a Goal can be seen as a procedure call that takes a number of typed arguments. In contrast to a procedure call, a Goal does not provide an implementation. Instead, a Goal may have multiple alternative implementations that can be selected dynamically at runtime. These implementations are called Techniques. A Technique consists of a declarative description and some arbitrary piece of code. In its declarative description a Technique may specify dependencies on further Goals. Thus, by resolving a Goal with a Technique, O2S decomposes a Goal hierarchically until it solely consists of Techniques. Together the transitive closure of techniques that has been selected in order to satisfy a certain Goal forms the executable application.

In O2S, computing a set of suitable Techniques is done by a centralized planning engine. In order to select the best Technique to resolve a Goal, each Technique provides a utility function. However, since the utility functions are evaluated independently, the planning engine does not perform a complete optimization. The O2S planning engine makes use of a number of heuristics in order to improve the efficiency. Some of these heuristics reduce the search space in such a way that the resulting search is no longer complete. To avoid the complete reconstruction of a plan during adaptation, the planning engine caches the currently executed plan and it solely evaluates some parts of the plan.

7.2.2 Smart Peer Groups

Smart peer groups are self-organizing groups of networked computers that are dynamically established when the computers are in physical proximity. Current system software for smart peer groups focuses mainly communication support. Towards this end, the solutions offer fundamental services such as device and service discovery as well as basic service abstractions used to unify access to the functionality available in a group. In order to communicate with other computers in a smart peer group, these systems support various communication paradigms, including message passing, remote method calls, and publish-subscribe-based event and data dissemination. Adaptation is typically supported at the communication layer but not at the application layer. Although, there is currently only one system that is specifically targeted at smart peer groups, there is a set of system software solutions that could accommodate this system model as well. In the following, we discuss relevant representatives in detail and we contrast them to the system software proposed in this dissertation.

7.2.2.1 BASE

BASE (Becker, Schiele, Gubbels, & Rothermel, 2003), (Handte, Becker, & Schiele, 2003) is a communication middleware for smart peer groups that has been developed in the Peer-to-peer Pervasive Computing project at the Universität Stuttgart. The core of this middleware is a minimal yet extensible micro-broker. The micro-broker takes care of managing local and remote communication. To integrate various communication technologies and protocols, the micro-broker can be flexibly extended using plug-ins. The basic functionality provided by BASE is energy-efficient discovery and a unified access to local and remote services available in a smart peer group.

As a result, applications usually consist of an atomic application core that leverages the BASE micro-broker to interact with the services that are hosted on the computers in their surrounding smart peer group. Towards this end, each computer that runs BASE hosts a service registry.

Installed services register themselves at their local service registry with their name, their type and optional properties that describe the non-functional characteristics of their implementation. Other services as well as application cores, in turn, may query their local registry for services that are available either on the local or on a remote computer.

With this approach, BASE closely follows the traditional model of object- and service-oriented communication middleware. The main difference to traditional middleware is threefold. First of all, the middleware consists of a minimal core that can be tailored to the capabilities of individual computers. This limits the amount of resources required to execute BASE and thus, it makes BASE suitable for resource-poor computers. Second, BASE decouples the communication model of the application from the communication model of the interoperability protocol. This enables BASE to dynamically switch between different communication technologies or protocols, even during on-going remote interaction. As a result, BASE can shield the application developer from many communication failures. Last but not least, all internal mechanisms in BASE can be federated dynamically without configuration or centralized services to support the concept of dynamically formed smart peer groups.

In cases where BASE cannot mask failures or in cases where a required service becomes unavailable, BASE solely provides signaling mechanisms. Thus, BASE forces application developers to deal with such failures explicitly. In contrast to BASE, PCOM aims to mask these failures as well. To do this, it adapts the configuration of an application, i.e. the set of services that constitutes an application, at runtime. Performing such an adaptation with BASE would require the development of adaptation logic within each service. However, the simple reselection of a service would not be sufficient as global resource conflicts might require the reconsideration of the complete application configuration. To enable the necessary global reconfiguration, each service that is used within an application would need to cooperate with the other services in order to enable a complete search through the space of possible application configurations.

7.2.2.2 Speakeasy

Speakeasy (Edwards K. W., et al., 2002) is a component system for pervasive computing applications that has been developed at the Palo Alto Research Center. Speakeasy is targeted at support for recombinant computing (Edwards K. W., Newman, Sedivy, Smith, & Izadi, 2002). The key idea behind recombinant computing is to enable the interaction of different components beyond the ways that have been foreseen by their developers. To realize this idea, the component model of Speakeasy defines four interfaces that need to be implemented by each component. These interfaces provide a description of the context of the component, e.g. its type, location, etc. and they define how to interact with the component, e.g. the supported data types, protocols and its user interface. Similar to Jini, accessing a component requires the download and execution of mobile code. However in contrast to Jini, the mobile code of each component is self-contained since it does not define specialized interfaces beyond the ones defined by the component model.

Once the code for a component is downloaded and running, a user may use the component's user interface to interact with it. Since the description of components by means of their built-in context interface is not standardized, the composition of an application from components cannot be automated. Instead, Speakeasy relies on manual configuration (Newman, et al., 2002) through the user. Due to the fact that the components are self-contained, there are no limits to the compositions that can be performed by the user. This makes Speakeasy extremely flexible and it can be used to support unstructured tasks such as the collaboration of users. Yet, the high level of compositional freedom is not always beneficial, since the component system cannot validate the compositions.

7.2.2.3 P2PComp

The goal of the P2PComp component system (Ferscha, Hechinger, Mayrhofer, & Oberhauser, 2004), (Ferscha, Hechinger, Mayrhofer, & Oberhauser, 2004) is to support the development of distributed component-based applications in a way that is independent from the underlying communication mechanism and protocol. To achieve this goal, a lightweight component container running on each computer mediates the communication between components on different computers. The component model of P2PComp is build on top of the model defined by the OSGi framework and it extends this model with a ports concept to enable the remote interaction between components.

The ports concept can be seen as the remote extension of the local service concept introduced in the OSGi framework. In the OSGi framework, a component can export one or more services that can then be used by other components. Similarly in P2PComp, a component can export one or more provides-ports. Other components can then make use of these ports through so-called use-ports. A port manager that is running on each component container is responsible for performing the necessary tasks to connect the provides-ports and the use-ports either locally or remotely. Both, provides-ports and use-ports, can be declared dynamically at runtime and components may also register continuous queries for provides-ports at the port manager. Since a network of computers may contain multiple provides-ports that might be suitable for a given use-port, a component may define a suitability value when registering a provides-port. This suitability value can then be used to break ties during the selection. If a provides-port is connected to a use-port, the port manager will dynamically create a proxy for the connection that is passed to the component that declares the use-port. This proxy can then be used for asynchronous and synchronous RPC-style interaction. If a connected component becomes unavailable, P2PComp is capable of switching transparently to another component that declares a suitable provides-port. However, P2PComp does not provide any services to support client-specific component state and thus, it cannot transparently handle the unavailability of stateful components.

As a result of this overall design, the components are shielded from the underlying communication mechanism and the port manager is the only part of the component system that needs to interface with it. As proof of concept, the P2PComp prototype system uses a JXTA (Sun Microsystems, 2007) implementation as basic communication mechanism. The JXTA protocols enable computers on a network to form a peer-to-peer overlay network that can then be used to share resources. Thus, JXTA takes care of registering and finding resources available on the network and it allows all computers to communicate with each other. Thus, the implementation of the port manager is greatly simplified, since it solely delegates the declaration of ports and queries to the corresponding parts of the JXTA implementation.

7.2.2.4 one.world

one.world (Grimm, 2004), (Grimm, et al., 2004) is an architecture for pervasive applications that has been developed at the University of Washington. The main goal of one.world is to empower application developers to program adaptive applications. Thus, in contrast to most other system software for pervasive applications, one.world does not strive for highly transparent adaptation. Instead, its main concern is to expose applications to all relevant changes that may be experienced during the execution of an application.

In order to avoid the complexities of dealing with the heterogeneity of computers, one.world relies on the Java virtual machine as single supported platform. On top of the Java virtual machine, one.world defines a framework for applications. The framework defines abstractions to structure an application and using these abstractions, it provides basic monitoring support as well as some mechanisms that can be used to adapt an application. However, the framework does not provide mechanisms to abstract from resources and it leaves the task of resource management to the application developer.

Applications in one.world are structured in environments, tasks and tuples. Environments are a controlling element in applications and they isolate different parts of an application from each other. An environment may contain tasks, tuples and other environments. Thus, environments may form a tree structure. Tasks are individual computations that are controlled by their surrounding environment – which, in turn, is controlled by its parent environments. Tasks can store their data in strongly typed and persistent tuples that are shared among the tasks of the same environment. Individual tasks can use asynchronous messages to communicate with each other.

The runtime system of one.world informs applications about changes using asynchronous events. As reaction they may create new environments and tasks, for example by copying them, or they may migrate or stop existing environments. Towards this end, the runtime system provides the necessary migration mechanisms. However, in contrast to traditional process migration, one.world does not deal with references to external resources. Instead, the runtime system will just free all resources of a migrating environment and it places the burden of allocating the new resources after the migration onto the application developer.

7.3 Discussion

The detailed analysis of existing system software for pervasive computing clearly shows that there is a great diversity in approaches and abstractions. At the communication level, systems like BASE or IROS are pursuing different approaches to hide the dynamic nature of the short-range wireless communication networks found in pervasive systems. Beyond the communication level, the systems introduce specialized abstractions to structure applications. Just like in system software for traditional applications, the abstractions vary heavily depending on the primary goals of their designers. Besides from classical service- and component-based approaches taken by BASE, Gaia, O2S, Speakeasy and P2PComp, there are systems that propose the utilization of traditional non-distributed applications as done in IROS and AURA, systems that propose the utilization of mobile agents as done in MetaGlue and even systems that propose novel abstractions as one.world.

To contrast these diverse approaches with the approach taken in this dissertation we can classify them. For the classification, we can look at their assumptions regarding the structure of the underlying pervasive system. On the basis of this, we can group the existing systems into system software for smart environments and system software for smart peer groups as done in the previous section. As explained earlier, system software for smart environments is usually built around a set of services that are coordinated by a single powerful computer. Due to this limitation, system software for smart environments cannot be applied directly to smart peer groups.

In addition to the system model, we can also classify the system software depending on how it handles the configuration and adaptation of an application. Here, we can identify approaches that propose the manual configuration by the user, approaches that target at support for programmatic configuration and adaptation by the application developer and systems that support the automatic configuration and adaptation at the system level. Clearly, approaches that strive for the automation of configuration and adaptation usually require some input from the application developer and they provide hooks to perform programmatic configuration or adaptation. Thus, many of the systems that are capable of automating configuration and adaptation can also support it programmatically, but not vice versa.

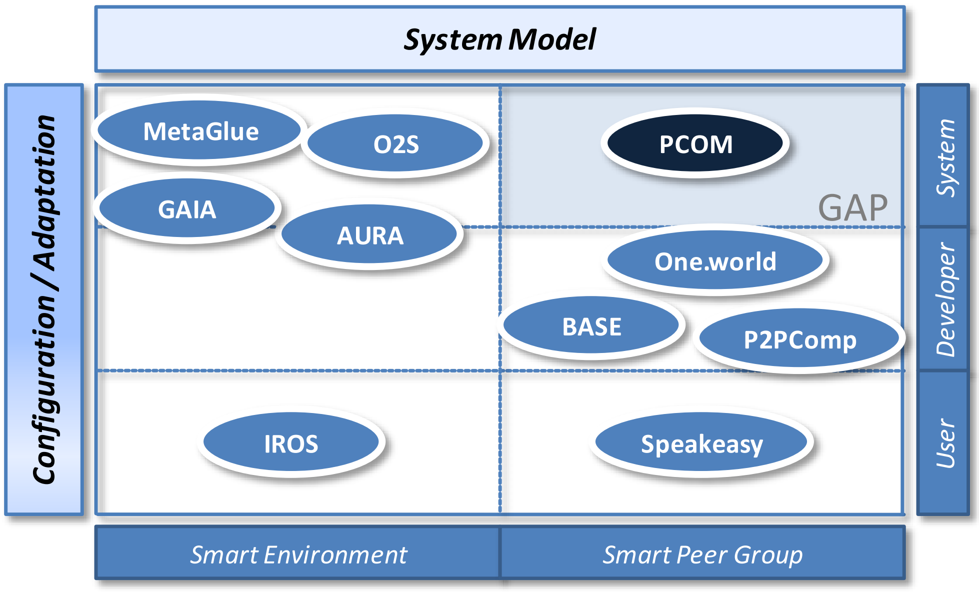


Figure 66 – Classification of System Support for Pervasive Applications

Figure 66 shows the result of this classification for the systems discussed previously. With the exception of IROS that proposes manual composition by means of ICrafter, all systems for smart environments are automating the configuration and adaptation. MetaGlue and O2S use centralized solvers for this task. Similarly, the Olympus service in GAIA uses a centralized approach to map components to the available computers. Finally, AURA uses the PRISM task manager which relies on a centralized environment manager to map tasks to applications. In contrast to this, the systems that are geared towards support for smart peer groups are focusing primarily on programmatic configuration and adaptation that is done by the application developer. As indicated in Figure 66, this results in a research gap with respect to system support for automatic configuration and adaptation in smart peer groups. The PCOM component system with its configuration algorithm and the adaptation heuristics fills this gap. This shows that automatic configuration and adaptation can also be done in a distributed fashion that is suitable for smart peer groups.

8 Conclusion

This chapter closes the dissertation with a short summary. The component system as well as the mechanisms and algorithms detailed in the previous chapters provide a solid basis for the development of adaptive pervasive applications that are executed by smart peer groups. Yet, the work presented in this dissertation indicates that a number of possible extensions might be worthwhile investigating. After summarizing the major findings of the dissertation, we present some of these extensions as an outlook on future work.

8.1 Summary

The proliferation of wireless communication technology and the ongoing miniaturization and integration of embedded systems are the technical foundation for future pervasive systems. On the one hand, pervasive applications that are executed in these systems have a potential to improve our daily life significantly. On the other, they complicate the task of application developers since they have to deal with the dynamics and the heterogeneity of the underlying networked computers as well as the continuous evolution of hard- and software.

Automatic configuration and adaptation at the system-level can greatly simplify application development as it enables applications to provide a seamless and distraction-free user experience despite their ever-changing execution environment. Yet, automatic configuration and adaptation at the system-level requires adequate abstractions that can be used to identify valid configurations and adaptations. In addition, it requires appropriate mechanisms to detect relevant changes as well as algorithms that can compute desirable configurations and adaptations.

In this dissertation, we have presented an integrated approach to enable automatic configuration and adaptation of pervasive applications. The approach is built around a component system that defines an application configuration as a set of components and resources that are hierarchically composed along contractually specified dependencies. The utilized contract model allows the specification of functional and non-functional properties. By supporting dynamic contracts with range operators and by allowing the specification of multiple optional contracts per component, the configurations can be adapted by means of reconfiguration and parameterization.

To automatically determine a valid configuration at runtime, we have proposed a distributed and parallel configuration algorithm. The algorithm is based upon asynchronous backtracking and thus, it inherits its properties. To apply this algorithm, we have presented a mapping that does not require the complete unfolding of the search space and that can be computed online without synchronization. Furthermore, we have added a termination detection protocol based on credit distribution and recovery to detect the successful computation of a configuration. Finally, we have proposed a number of problem specific optimizations and we have developed an approach to achieve resilience.

In order to support reactive automatic adaptation, we have introduced a number of supportive mechanisms for monitoring, signaling and state maintenance and we have designed a cost model to capture the cost of modifying parts of the configuration. On the basis of this model, we have developed a set of light weight optimization heuristics and we have discussed how these heuristics can be integrated into the configuration algorithm.

To integrate the component system, the configuration algorithm and the adaptation heuristics, we have designed a prototypical architecture which we have implemented. On the basis of this implementation, we have evaluated the resulting overheads, benefits and limitations of the component system. Furthermore, we have evaluated the overhead of automatic configuration and the suitability of the adaptation heuristics. The evaluation clearly indicates that introduced abstractions and algorithms may introduce considerable overheads. However, the evaluation also suggests that the proposed approach can greatly simplify application development and that the overheads are reasonable for a broad spectrum of application scenarios.

Although, the proposed component system exhibits some similarities with other system software for pervasive applications, the in-depth comparison clearly shows that there are important differences. Most noteworthy, our approach is specifically targeted at smart peer groups. As a consequence, it must be lightweight and it cannot rely on the presence of a powerful and reliable computer that can be used for centralized coordination. Thus, all mechanisms and algorithms must be designed in such a way that they cause little overhead and that they can tolerate unpredictable failures of other computers without permanent undesirable effects. The integrated approach described in this dissertation shows that such a design is possible and that it can be applied in practice.

8.2 Outlook

Together, the abstractions, mechanisms and algorithms presented in this dissertation provide a solid basis for the development of adaptive pervasive applications. Yet, it is possible to identify a number of extensions that might be worthwhile to investigate more closely as part of future research. With respect to configuration and adaptation algorithms possible research topics include metrics to enable proactive adaptation, optimized support for user preferences, and dynamically distributed configuration algorithms as well as system-wide conflict resolution. Besides the topics that are related to algorithms, there are also further research areas that might be worthwhile to explore such as end-user development and customization tools.

8.2.1 Metrics for Proactive Adaptation

The mechanisms and algorithms presented in this dissertation are geared towards reactive adaptation, i.e. adaptation that takes place after a disruptive change occurred. Given an adequate set of prediction methods, it might be possible to prevent reactive adaptation in some cases by adapting the application before a disruptive change occurs. As a simple example, consider that it might be possible to predict future disconnections by measuring the changes in connection quality of wireless links over time. This can significantly reduce the impact of changes, e.g. by minimizing the perceived adaptation delay. Yet, performing such proactive adaptations requires adequate prediction metrics and adequate adaptation metrics that decide when and how to compute an adaptation. Technically, such metrics could be integrated into the optimization heuristics during adaptation. However, the design of such metrics requires a conscious tradeoff between the adaptation costs and the achievable optimization. As a consequence, designing such metrics requires more thorough research.

8.2.2 Optimizations for User Preferences

The proposed component system enables users to specify user preferences as an ordered list of independent requirements towards the application anchor. As a consequence, the computation of different requests is also performed independently, i.e. one configuration attempt is performed for each individual preference. Yet, in some cases user preferences are merely an ordered set of relaxations that strictly extend the number of possible configurations. As an example consider a performance related set of preferences that first tries to find a high performance configuration before it additionally allows low performance configurations. For user preferences that are strict relaxations of requirements, it is possible to speed up the configuration process by reusing the results of the previous configuration attempt. For instance, one might try to compute a low performance configuration and if that is possible, one may introduce additional constraints to restrict the result to high performance configurations. If the first configuration attempt fails already, it is clear that a high performance configuration cannot be found as well. In order to perform such an optimization, it is necessary to specifically consider inclusion relationships during the configuration of an application.

8.2.3 Dynamically Distributed Configuration

In order to support smart peer groups that consist solely of resource-poor computers, our configuration algorithm is fully distributed. However, not all future smart peer groups will solely consist of resource-poor computers. Instead a number of them may contain resource-rich computers such as desktops or servers as well. In such smart peer groups it may be possible to speed up the configuration by dynamically adapting the distribution of the configuration algorithm according to the capabilities of the computers. As an extreme example, one could dynamically pick the most powerful computer to compute the configuration in a centralized manner (Handte, Herrmann, Schiele, & Becker, 2007). Clearly, there is a broad spectrum of intermediate possibilities between the full distribution proposed in this dissertation and centralized configuration that is frequently utilized by other systems. Technically, our proposed architecture can support the dynamic distribution of the configuration process as it does not prescribe the mapping between assembler and container. However, in order to optimally leverage the resources of a smart peer group it is necessary to design suitable distribution metrics and heuristics. This requires a thorough analysis of the cost factors and the achievable speedups.

8.2.4 System-wide Conflict Resolution

The configuration and adaptation algorithms described in this dissertation only resolve resource conflicts of a single application. Clearly, it is easy to extend the approach to multiple applications by combining multiple user preferences. In fact, our implementation can already support such cases as it does not pose restrictions on the number of components referenced within a user preference. Thus, it is technically possible to resolve resource conflicts across multiple applications that are executed on behalf of the same user. However, it is conceivable that future pervasive systems will be used by multiple users at the same time. This creates a potential for resource conflicts between applications that are executed on behalf of different users and a resource assignment on a first-come-first-served basis is not desirable in all cases (Schiele, Handte, & Becker, 2007). As a result, it is necessary to detect such cases and to provide adequate conflict resolution strategies that also consider potential interference and social aspects.

8.2.5 End-user Development Tools

The proposed component system enables the development of applications that can adapt to the capabilities of the execution environment and to the preferences of their users. Yet, the basis for this adaptation is the availability of components that combine different components and resources to provide the desired functionality. Clearly, future hardware providers will rely on professional application developers to provide a number of components that allow the usage in different applications. Yet, the available components may not be sufficient to support the heterogeneity of hardware and the diversity of application scenarios resulting from different user needs. This can be mitigated by allowing technically versed users to customize their applications with application-specific components. Yet, the abstractions introduced by the component system are clearly geared towards professional application developers and they might be too complicated for technically versed users. This problem can be mitigated by providing high-level component development tools that are suitable for end-users. In an initial attempt for building such a tool, we have developed Nexel (Weis, Handte, Knoll, & Becker, 2006). Yet, there is still a need for a thorough evaluation of the suitability of such approaches.

9 Bibliography

Affenzeller, M., & Mayrhofer, R. (2002, October). Generic Heuristic. *9th International Conference on Operational Research (KOI)* , pp. 83-92.

Aitenbichler, E., Kangasharju, J., & Mühlhäuser, M. (2005). Experiences with MundoCore. *3rd IEEE International Conference on Pervasive Computing and Communications Workshops*, (pp. 168-172). Hawaii, USA.

Arshad, N., Heimbigner, D., & Wolf, A. (2003). Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)*, (pp. 39-46). Sacramento, USA.

Baker, A. (1995, March). Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results. *PhD Thesis, University of Oregon* , pp. 1-172.

Ballagas, R., Szybalski, A., & Fox, A. (2004). Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments. *2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, (pp. 241-52). Orlando, USA.

Bayardo, R., & Miranker, D. (1994). Backtrack-Bounded Search in Polynomial Space. *Technical Report (94-12), University of Texas* , pp. 1-20.

Becker, C., Handte, M., Schiele, G., & Rothermel, K. (2004). PCOM – A Component System for Pervasive Computing. *2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, (pp. 67-76). Orlando, USA.

Becker, C., Schiele, G., Gubbels, H., & Rothermel, K. (2003). BASE - A Micro-broker-based Middleware for Pervasive Computing. *1st IEEE International Conference on Pervasive Computing and Communications*, (pp. 443-451). Fort Worth, USA.

Bessiere, C., Maestre, A., & Meseguer, P. (2001, August). Distributed Dynamic Backtracking. *Workshop on Distributed Constraints (IJCAI-01)* , pp. 1-8.

Beugnard, A., Jezequel, J.-M., Plouzeau, N., & Watkins, D. (1999, July). Making Components Contract Aware. *IEEE Computer , 32* (7), pp. 38-45.

Birrel, A., & Nelson, B. (1984, February). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS) , 2* (1), pp. 39-59.

Blair, G., Coulson, G., Robin, P., & Papathomas, M. (1998). An Architecture for Next Generation Middleware. *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* , pp. 1-16.

Bluetooth Special Interest Group. (2004, November). Bluetooth Core Specification, Version 2.0. *http://www.bluetooth.org* .

Brooks, R. (1997). The Intelligent Room Project. *2nd International Cognitive Technology Conference (CT'97)*, (pp. 271-278). Aizu, Japan.

Carriero, N., & Gelernter, D. (1986, May). The S/Net's Linda Kernel. *ACM Transactions on Computer Systems (TOCS) , 4* (2), pp. 110-129.

Chandy, M., & Lamport, L. (1985, February). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems , 3* (1), pp. 63-75.

Chappell, D. (2002). *.Net verstehen.* Addison-Wesley.

Chechetka, A., & Sycara, K. (2006, May). No-commitment branch and bound search for distributed constraint optimization. *Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems* , pp. 1427 - 1429.

Chen, H., Finin, T., & Joshi, A. (2004). Semantic Web in the Context Broker Architecture. *Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, (pp. 277-286). Orlando, USA.

Chetan, S., Ranganathan, A., & Campbell, R. (2005, Spring). Towards Fault Tolerant Pervasive Computing. *IEEE Technology and Society , 24* (1), pp. 28-44.

Chun, A. (1999, April). Constraint Programming with JSolver. *1st International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming* , pp. 1-12.

Chung, E., Huang, Y., Yajnik, S., Liang, D., Shih, J., Wang, C.-Y., et al. (1998, January). DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. *C++ Report , 10* (1), pp. 18-29.

Coen, M., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., & Finin, P. (1999). Meeting the Computational Needs of Intelligent Environments: The Metaglue System. *1st International Workshop on Managing Interactions in Smart Environments (MANSE'99)* , pp. 201-212.

Cook, S. (1971). The Complexity of Theorem-Proving Procedures. *STOC '71: 3rd Annual ACM Symposium on Theory of Computing* , pp. 151-158.

Dakin, R. J. (1965). A tree-search algorithm for mixed integer programming problems. *Computer Journal , 3*, pp. 250-255.

Dechter, R., & Rossi, F. (2000). Constraint Satisfaction. *Encyclopedia of Cognitive Sciences* , pp. 1-15.

Dermler, G. (1999). *Ressourcenreservierung und Task-Platzierung in verteilten Multimedia-Systemen.* Dissertation, Universität Stuttgart.

Dijkstra, E., & Scholten, C. (1980, August). Termination Detection for Diffusing Computations. *Information Processing Letters , 1*, pp. 1-7.

Edwards, K. W., Newman, M. W., Sedivy, J. Z., Smith, T. F., Balfanz, D., Smetters, D. K., et al. (2002). Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration. *2002 ACM Conference on Computer Supported Cooperative Work* , pp. 256-265.

Edwards, K. W., Newman, M. W., Sedivy, J., Smith, T., & Izadi, S. (2002). Challenge: Recombinant Computing and the Speakeasy Approach. *8th Annual International Conference on Mobile Computing and Networking* , pp. 279-286.

Environment Australia. (2001). *Major Appliances Materials Project.* Australia: Enproc Pty Ltd.

Eugster, P., Felber, P., Guerraoui, R., & Kermarrec, A.-M. (2003, June). The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR) , 35* (2), pp. 114-131.

European Telecommunication Standards Institute. (2000, September). General Packet Radio Service(GPRS), Version 7.4.1, EN 301 344, (2000-9). *http://www.etsi.org* .

Ferscha, A., Hechinger, M., Mayrhofer, R., & Oberhauser, R. (2004). A Light-Weight Component Model for Peer-to-Peer Applications. *2nd International Workshop on Mobile Distributed Computing*, (pp. 520-527). Tokyo, Japan.

Ferscha, A., Hechinger, M., Mayrhofer, R., & Oberhauser, R. (2004). A Peer-to-Peer Light-Weight Component Model for Context-Aware Smart Space Applications. *International Journal of Wireless and Mobile Computing , 2004* (4), pp. 1-10.

Flinn, J., Narayanan, D., & Satyanaray, M. (2001, May). Self-tuned Remote Execution for Pervasive Computing. *8th Workshop on Hot Topics in Operating Systems* , pp. 61-66.

Francez, N. (1980, January). Distributed Termination. *ACM Transactions on Programming Languages and Systems , 2* (1), pp. 42-55.

Gajos, K. (2001). Rascal - A Resource Manager for Multi Agent Systems in Smart Spaces. *2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems*, (pp. 111-120). Cracow, Poland.

Gajos, K., Weisman, L., & Shrobe, H. (2001). Design Principles For Resource Management Systems For Intelligent Spaces. *2nd International Workshop on Self-Adaptive Software* , pp. 1-19.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Amsterdam, Netherlands: Addison-Wesley Longman.

Garlan, D., Siewiorek, D., Smailagic, A., & Steenkiste, P. (2002, April-June). Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing , 1* (2), pp. 22-31.

Gaschnig, J. (1977, August). A General Backtrack Algorithm that Eliminates Most Redundant Tests. *5th International Joint Conference on Artificial Intelligence* , p. 457.

Ginsberg, M. (1993, August). Dynamic Backtracking. *Journal of Artificial Intelligence Research , 1*, pp. 25-46.

Gray, C., & Cheriton, D. (1989). Leases: An Efficient and Fault-tolerant Mechanism for Distributed File Cache Consistency. *12th ACM Symposium on Operating Systems Principles* , pp. 202-210.

Grimm, R. (2004, July-September). One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing , 3* (3), pp. 22-30.

Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., et al. (2004, November). System Support for Pervasive Applications. *ACM Transactions on Computer Systems , 22* (4), pp. 421-486.

Hamadi, Y. (2002, May). Interleaved Backtracking in Distributed Constraint Networks. *International Journal on Artificial Intelligence Tools , 11* (2), pp. 167-188.

Hamadi, Y. (2002). Optimal Distributed Arc-Consistency. *Constraints* (7), pp. 367-385.

Handte, M., Becker, C., & Rothermel, K. (2005). Peer-based Automatic Configuration of Pervasive Applications. *International Conference on Pervasive Services 2005 (ICPS '05)*, (pp. 249-260). Santorini, Greece.

Handte, M., Becker, C., & Rothermel, K. (2005, December). Peer-based Automatic Configuration of Pervasive Applications. *Journal on Pervasive Computing and Communications (JPCC) , 1* (4), pp. 251-264.

Handte, M., Becker, C., & Schiele, G. (2003). Experiences: Minimalsim and Extensibility in BASE. *Workshop on System Support for Ubiquitous Computing (UbiSys'03)*, (pp. 1-8). Seattle, USA.

Handte, M., Herrmann, K., Schiele, G., & Becker, C. (2007). Supporting Pluggable Configuration Algorithms in PCOM. *Middleware Support for Pervaisve Computing Workshop (PerWare at PerCom'07)*, (pp. 472-476). New York, USA.

Handte, M., Herrmann, K., Schiele, G., Becker, C., & Rothermel, K. (2007). Automatic Reactive Adaptation of Pervasive Applications. *IEEE International Conference on Pervasive Services 2007 (ICPS'07)*, (pp. 214-222). Istanbul, Turkey.

Handte, M., Schiele, G., Urbanski, S., Becker, C., & Rothermel, K. (2005). Adaptation Support for Stateful Components in PCOM. *Workshop on Software Architectures for Self-Organization: Beyond Ad-Hoc Networking at Pervasive 2005*, (pp. 1-6). Munich, Germany.

Handte, M., Urbanski, S., Becker, C., Reinhardt, P., Engel, M., & Smith, M. (2006). 3PC/MarNET Pervasive Presenter. *Demonstration at 4th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'06)*, (pp. 1-2). Pisa, Italy.

Hirayama, K., & Yokoo, M. (2000). An Approach to Over-constrained Distributed Constraint Satisfaction Problems: Distributed Hierarchical Constraint Satisfaction. *4th International Conference on Multiagent Systems* , pp. 135-142.

Hirayama, K., & Yokoo, M. (1997). *Distributed Partial Constraint Satisfaction Problem.*

Hirayama, K., & Yokoo, M. (2000). The Effect of Nogood Learning in Distributed Constraint Satisfaction. *20th IEEE International Conference on Distributed Computing Systems* , pp. 169-177.

Hohl, F., Kubach, U., Leonhardi, A., Rothermel, K., & Schwehm, M. (1999). Next Century Challenges: Nexus - An Open Global Infrastructure for Spatial-Aware Applications. *Fifth Annual International Conference on Mobile Computing and Networking (MobiCom '99)*, (pp. 249-255). Seattle, WA, USA.

Institute of Electrical and Electronics Engineers. (2003, June). IEEE Standard 802.11, 1999 Edition (R2003). *http://standards.ieee.org* .

Johanson, B., & Fox, A. (2002). The Event Heap: A Coordination Infrastructure for Interactive Workspaces. *4th IEEE Workshop on Mobile Computing Systems and Applications*, (pp. 83-93). Callicoon, USA.

Johanson, B., Fox, A., & Winograd, T. (2002, April-June). The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing* , pp. 67-74.

Joseph, A., Tauber, J., & Kaashoek, F. (1997, March). Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers , 46* (3), pp. 337-352.

Kiciman, E., & Fox, A. (2000). Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. *2nd International Symposium on Handheld and Ubiquitous Computing* , pp. 211-226.

Kon, F. (2000). *Automatic Configuration of Component-Based Distributed Systems.* Department of Computer Science, University of Illinois at Urbana Champaign: PhD Thesis.

Koulamas, C., Antony, S., & Jean, R. (1994). A Survey of Simulated Annealing Applications to Operations Research Problems. *Omega International Journal of Management Science , 22* (1), pp. 41-56.

Kumar, V. (1992). Algorithms for Constraint-Satisfaction Problems: A Survey. (A. A. Intelligence, Ed.) *AI Magazine , 13* (1), pp. 32-44.

Lai, T.-H., & Wu, L.-F. (1995, January). An (N-1)-Resilient Algorithm for Distributed Termination Detection. *IEEE Transactions on Parallel and Distributed Systems , 6* (1), pp. 63-78.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM , 21* (7), pp. 558-565.

Mackworth, A. (1977). Consistency in Networks of Relations. *Artificial Intelligence , 8* (1), pp. 99-118.

Martin, O., & Otto, S. (1993). Combining Simulated Annealing with Local Search Heuristics. (O. G. Engineering, Ed.) *Technical Report CSE-94-016* , pp. 1-15.

Matocha, J., & Camp, T. (1998, November). A Taxonomy of Distributed Termination Detection Algorithms. *Journal of Systems and Software , 43* (3), pp. 2007-221.

Mattern, F. (1989). Global Quiescence Detection Based on Credit Distribution and Recovery. *Information Processing Letters , 30* (4), pp. 195-200.

Mattern, F. (2001, June). Pervasive/Ubiquitous Computing. *Informatik-Spektrum , 24* (3), pp. 145-147.

Meisels, A., & Zivan, R. (2007, March). Asynchronous Forward-checking for DisCSPs. *Constraints* , pp. 131-150.

Microsoft Cooperation. (2000). *Understanding Universal Plug and Play: A White Paper.* UPnP Forum.

Microsoft Corporation. (1996, November). *DCOM Technical Overview.* Retrieved July 2007, from Microsoft Developer Network Library: http://msdn2.microsoft.com/en-us/library/ms809340.aspx

Microsoft Corporation. (1995, October). *The Component Object Model Specification.* Retrieved July 2007, from COM: Component Object Model Technology Website: http://www.microsoft.com/com

Modi, P. J., Shen, W.-M., Tambe, M., & Yokoo, M. (2005). ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence Journal , 161*, pp. 149-180.

Modi, P. J., Shen, W.-M., Tambe, M., & Yokoo, M. (2003, July). An Asynchronous Complete Method for Distributed Constraint Optimization. *Second International Joint Conference on Autonomous Agents and Multiagent Systems* , pp. 161-168.

Moore, G. (1965, April). Cramming more Components onto Integrated Circuits. *Electronics Magazine , 38* (8), pp. 114-117.

Nam, J., Shin, D., Hur, S., & Han, C. (2007). An ECA-based Mechanism of Non-blocking Device Coordination for a Ubiquitous Environment. *IEEE International Conference on Convergence Information Technology (ICCIT'07)* (pp. 573-579). Washington, USA: IEEE Computer Society.

Newman, M. W., Sedivy, J. Z., Neuwirth, C. M., Edwards, K. W., Hong, J. I., Izadi, S., et al. (2002). Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environment. *4th Conference on Designing Interactive Systems* , pp. 147-156.

Nguyen, V., Sam-Haroud, D., & Faltings, B. (2004, September). Dynamic Distributed Backjumping. *5th Workshop on Distributed Constraints Reasoning* , pp. 1-15.

Noble, B., & Satyanarayanan, M. (1999, December). Experience with Adaptive Mobile Applications in Odyssey. *Mobile Networks and Applications , 4* (4), pp. 245-254.

Noble, B., Satyanarayanan, M., Tilton, J., Jason, F., & Walker, K. (1997). Agile Application-Aware Adaptation for Mobility. *16th ACM Symposium on Operating Systems Principles*, (pp. 276-287). Saint Malo, France.

Object Management Group. (2004, March). *Common Object Request Broker Architecture: Core Specification.* Retrieved July 2007, from Object Management Group Homepage: http://www.omg.org/docs/formal/04-03-12.pdf

Object Management Group. (2006, April). *CORBA Component Model Specification.* Retrieved July 2007, from Object Management Group Homepage: http://www.omg.org/docs/formal/06-04-01.pdf

Object Management Group. (2002, August). *Minimum Corba Specification (Version 1.0).* Retrieved January 2008, from Object Management Group Homepage: http://www.omg.org/cgi-bin/apps/doc?formal/02-08-01.pdf

OSGi Alliance. (2007, April). *OSGi Service Platform Core Specification.* Retrieved July 2007, from OSGi Alliance Specifications Website: http://www2.osgi.org/Specifications/HomePage

OSGi Alliance. (2007, April). *OSGi Service Platform Service Compendium.* Retrieved July 2007, from OSGi Specifications Website: http://www2.osgi.org/Specifications/HomePage

Paluska, J., Pham, H., Saif, U., Chau, G., & Ward, S. (2008, March). Structured Decomposition of Adaptive Applications. *6th Annual IEEE International Conference on Pervasive Computing and Communications* , pp. 1-10.

Ponnekanti, S., Johanson, B., Kiciman, E., & Fox, A. (2003, March). Portability, Extensibility and Robustness in iROS. *1st IEEE International Conference on Pervasive Computing and Communications* , pp. 11-19.

Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., & Winograd, T. (2001). ICrafter: A Service Framework for Ubiquitous Computing Environments. *3rd International Conference on Ubiquitous Computing* , pp. 56-75.

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R., & Mickunas, D. (2005, March). Olympus: A High-Level Programming Model for Pervasive Computing Environments. *3rd IEEE International Conference on Pervasive Computing and Communications* , pp. 7-16.

Rivera, W. (2001). Scalable Parallel Genetic Algorithms. (K. A. Publishers, Ed.) *Artificial Intelligence Review , 16* (2), pp. 153-168.

Roman, E., Siganesh, R., & Brose, G. (2005). *Mastering Enterprise JavaBeans (3rd Edition).* Indianapolis, USA: Wiley Publishing.

Roman, M., & Campbell, R. (2000, September). Gaia: Enabling Active Spaces. *9th ACM SIGOPS European Workshop* , pp. 229-234.

Roman, M., & Campbell, R. (2001). Unified Object Bus: Providing Support for Dynamic Management of Heterogeneous Components. *UIUC Technical Report, UIUCDCS-R-2001-2222* , pp. 1-18.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., & Nahrstedt, K. (2002, October-December). Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing , 1* (4), pp. 74-83.

Roman, M., Kon, F., & Campbell, R. (2001, July). Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online , 2* (5).

Roman, M., Singhai, A., Carvalho, D., Hess, C., & Campbell, R. (1999). Integrating PDAs into Distributed Systems: 2K and PalmORB. *1st International Symposium on Handheld and Ubiquitous Computing* , pp. 137-149.

Rossi, F., Perie, C., & Dhar, V. (1990). On the Equivalence of Constraint Satisfaction Problems. *9th European Conference on Artificial Intelligence* , pp. 550-556.

Rothermel, K., Barth, I., & Helbig, T. (1994). Cinema - An Architecture for Distributed Multimedia Applications. In O. Spaniol, A. Danthine, & W. Effelsberg, *Architecture and Protocols for High-Speed Networks* (pp. 253-271). Kluwer Academic Publishers.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach (Second Edition, Chapter 5).* Prentice Hall International.

Saif, U., Pham, H., Paluska, J., Waterman, J., Terman, C., & Ward, S. (2003, October). A Case for Goal-oriented Programming Semantics. *System Support for Ubiquitous Computing Workshop, 5th Annual Conference on Ubiquitous Computing* , pp. 1-8.

Satyanarayanan, M. (2002, May). The Evolution of Coda. (ACM, Ed.) *ACM Transactions on Computer Systems , 20* (2), pp. 85-124.

Schiele, G. (2007). *System Support for Spontaneous Pervasive Computing Environments.* Dissertation, Universität Stuttgart.

Schiele, G., Becker, C., & Rothermel, K. (2004). Energy-Efficient Cluster-based Service Discovery for Ubiquitous Computing. *11th ACM SIGOPS European Workshop*, (pp. 1-5). Leuven, Belgium.

Schiele, G., Handte, M., & Becker, C. (2007). Good Manners for Pervasive Applications - An Approach Based on the Ambient Calculus. *WIP-Track, 5th IEEE International Conference on Pervasive Computing and Communications (PerCom'07)*, (pp. 585-588). New York, USA.

Schneider, F. (1990, December). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys , 22* (4), pp. 299-319.

Selman, B., Kautz, H., & Cohen, B. (1994, July). Noise Strategies for Improving Local Search. *Twelfth National Conference on Artificial Intelligence (AAAI'94)* , pp. 337--343.

Selman, B., Levesque, H., & Mitchell, D. (1992). A New Method for Solving Hard Satisfiability Problems. *10th National Conference on Artificial Intelligence* , pp. 440-446.

Silaghi, M. C., & Yokoo, M. (2006, May). Nogood based Asynchronous Distributed Optimization. *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* , pp. 1389 - 1396 .

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2000). Asynchronous Search with Aggreations. *7th National Conference on Artificial Intelligence/12th Conference on Innovative Applications on Artificial Intelligence* , pp. 917-922.

Sousa, J. P., & Garlan, D. (2002, August). AURA: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. *3rd IEEE/IFIP Conference on Software Architecture* , pp. 29-43.

Sun Microsystems. (1999, September 12). *Code Conventions for the Java Programming Language.* Retrieved October 2007, from Sun Developer Network Homepage: http://java.sun.com/docs/codeconv/

Sun Microsystems. (2004). *Java Remote Method Invocation Specification.* Retrieved July 2007, from Java Technology Website: http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf

Sun Microsystems. (1997, August). *JavaBeans Specification Version 1.01.* Retrieved January 2008, from Sun Developer Network Homepage: http://java.sun.com/products/javabeans/docs/spec.html

Sun Microsystems. (2001, December). *Jini Technology Core Platform Specification.* Retrieved July 2007, from Java Technology Website.

Sun Microsystems. (2007, October). *JXTA Protocols Specification (Version 2.0).* Retrieved April 9, 2007, from JXTA Community Homepage: https://jxta-spec.dev.java.net/

Szyperski, C. (1997). *Component Software Beyond Object-Oriented Programming.* Addison-Wesley.

Tel, G., & Mattern, F. (1993, January). The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems , 15* (1), pp. 1-35.

Testa, C., & Dearie, D. (1974). Human Factors Design Criteria in Man-Computer Interaction. *ACM 74: 1974 Annual Conference* , pp. 61-65.

Tsang, E., & Voudouris, C. (1995). Fast Local Search and Guided Local Search and Their Application to British Telecom's Workforce Scheduling Problem. (U. o. Essex, Ed.) *Technical Report CSM-246* , pp. 1-15.

Tsang, E., & Voudouris, C. (1999, March). Guided Local Search and its application to the Travelling Salesman Problem. (A. Publishing, Ed.) *European Journal of Operational Research , 113* (2), pp. 469-499 .

Tseng, Y.-C., & Tan, C.-C. (2001, June). Termination Detection Protocols for Mobile Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems , 12* (6).

Weis, T., Handte, M., Knoll, M., & Becker, C. (2006). Customizable Pervasive Applications. *4th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'06)*, (pp. 239-244). Pisa, Italy.

Weiser, M. (1991, February). The computer for the 21st century. *Scientific American , 265* (3), pp. 66-75.

World Wide Web Consortium. (2007, April). SOAP (Version 1.2). *W3C Recommendation* .

World Wide Web Consortium. (2007, June). Web Service Description Language (Version 2.0). *W3C Recommendation* .

World Wide Web Consortium. (2002). *Web Services*. Retrieved January 2008, from http://www.w3.org/2002/ws/

Xu, D., Nahrstedt, K., & Wichadakul, D. (2001, April). QoS and Contention-aware Multi-Resource Reservation. *Cluster Computing , 4* (2), pp. 95-107.

Yeoh, W., Felner, A., & Koenig, S. (2008, May). BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)* , pp. 591-598.

Yokoo, M. (1995). Asynchronous Weak-Commitment Search for Solving Distributed Constraint Satisfaction Problems. *1st International Conference on Principles and Practice of Constraint Programming* , pp. 88-102.

Yokoo, M., & Hirayama, K. (2000). Algorithms for Distributed Constraint Satisfaction: A Review. *Aoutonomous Agents and Multi-Agent Systems , 2*, pp. 198-212.

Yokoo, M., Durfee, E., Ishida, T., & Kuwabara, K. (1992). Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *12th IEEE International Conference on Distributed Computing Systems* , pp. 614-621.

Yokoo, M., Durfee, E., Ishida, T., & Kuwabara, K. (1998, September/October). The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering , 10* (5), pp. 673-685.

Zivan, R., & Meisels, A. (2003, December). Synchronous vs. Asynchronous search on DisCSPs. *1st European Workshop on Multi Agent System* , pp. 1-11.

10 Indexes

10.1 Index of Figures

10.2 Index of Formulas

10.3 Index of Algorithms

10.4 Index of Tables

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

Marcus Handte