Sven Trieflinger

Fault Tolerance

High-Performance

Modularity

Peer-to-Peer

Architecture

Isolation

# Desktop Grid Computing

Methods

Volatility

XMPP

Topologies

Heterogeneity

Applications

Capabilities

Satisfiability

Termination

Institute for Parallel and Distributed Systems (IPVS) • University of Stuttgart

# High Performance Peer-to-Peer Desktop Grid Computing

**Architecture, Methods, 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

Sven Trieflinger

aus Schramberg a.N.

|  |  |
|---|---|
| Hauptberichter: | Prof. Dr. rer. nat. Dr. h.c. Kurt Rothermel |
| Mitberichter: | Prof. Dr. rer. nat. Frank Leymann |
| Tag der mündlichen Prüfung: | 29. Oktober 2012 |

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2013

To my beloved wife and my amazing kids

# Acknowledgements

I would like to thank my Ph.D. supervisors Prof. Dr. Kurt Rothermel and Prof. Dr. Wolfgang Küchlin for giving me the opportunity to work in their groups. My special thanks go to my co-advisor Dr. Wolfgang Blochinger, most importantly for his guidance in scientific working and writing. Furthermore, I should like to thank Prof. Dr. Frank Leymann for accepting the task of being the second referee.

I also want to thank all the students who contributed to this thesis as part of their diploma theses, student research projects, and student assistant jobs. In particular I'd like to thank my coauthors Mathias Poths and Hannes Hannak – it has been a pleasure to work with you.

I am also grateful to the German National Science Foundation (Deutsche Forschungsgemeinschaft, DFG) for funding my work under grants BL 941/1-1 and BL 941/1-2 [Deu06].

Finally, I would like to express my deepest gratitude to my family for their invaluable and constant support, in particular to Helga Schulz for being the best mother I can imagine, to my wife Julia for her endless amount of patience and the inspiring technical discussions, and finally to my kids Thomas and Sarah for reminding me there are things in life even more enjoyable than work.

# Contents

# List of Figures

# List of Tables

# List of Modules, Protocols, and Algorithms

# List of Abbreviations and Acronyms

| | |
|---|---|
| (R-)OSGi | (Remote) Open Services Gateway initiative |
| ACL | Access Control Lists |
| ADHT | Autonomous Distributed Hash Table |
| ALEF | Adaptive Lemma Exchange Facility |
| AMF | Active Message Format |
| AML | Aggregation Management Layer |
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| BCNF | Bit-Packed Binary CNF |
| BOINC | Berkeley Open Infrastructure for Network Computing |
| BSP | Bulk Synchronous Parallel |
| C2S | Client-to-Server |
| CAN | Content Addressable Network |
| CCOF | Cluster Computing on the Fly |
| CDCL | Conflict Driven Clause Learning |
| CNF | Conjunctive Normal Form |
| COM | Component Object Model |
| CONFIIT | Computation Over Network for FIIT |
| COTS | Commercial-Off-The-Shelf |
| CPM | Compute Power Market |
| CPU | Central Processing Unit |
| CRCW | Concurrent Read, Concurrent Write |
| CX | Computational Exchange |
| DAG | Directed Acyclic Graph |
| DEGISCO | Desktop Grids for International Scientific Collaboration |
| DHT | Distributed Hash Table |
| DIMACS | Center for Discrete Mathematics and Theoretical Computer Science |
| DIRAC | Distributed Infrastructure with Remote Agent Control |
| DMZ | Demilitarized Zone |
| DPLL | Davis-Putnam-Logemann-Loveland |
| DSM | Distributed System Model |
| E2E | End-to-End |
| EBNF | Extended Backus-Naur Form |
| EC2 | Elastic Compute Cloud |
| EDGeS | Enabling Desktop Grids for e-Science |
| EGEE | Enabling Grids for E-sciencE |
| ERM | Equivalent Reference Machines |
| F2F | Friend-to-Friend |
| FI | Fast Infoset |
| FIIT | Finite number of Independent and Irregular Tasks |
| FLOPS | Floating Point Operations per Second |
| FLP | Fischer, Lynch, Paterson |
| GGF | Global Grid Forum |
| GNU | GNU's Not Unix |
| GP | Guiding Path |
| GPU | Graphics Processing Unit |
| GZIP | GNU zip |
| HPC | High Performance Computing |
| HTC | High Throughput Computing |
| HTTP | Hypertext Transfer Protocol |
| I/O | Input/Output |
| IaaS | Infrastructure as a Service |
| IAS | Isolation Admin Service |
| ICE | Interactive Connectivity Establishment |
| ID | Identifier |
| IDA | Information Dispersal Algorithm |
| IDE | Integrated Development Environment |
| IESP | International Exascale Software Project |
| IETF | Internet Engineering Task Force |
| IG | Iterated Greedy |
| ILDB | Inbound Lemma Database |

| | |
|---|---|
| IMMS | Isolating Module Management System |
| IP | Internet Protocol |
| IPL | Ibis Portability Layer |
| ISP | Irregulary Structured Problem |
| JavaGAT | Java Grid Application Toolkit |
| JEE | Java Enterprise Edition |
| JID | Jabber Identity |
| JIT | Just-In-Time |
| JMX | Java Management Extensions |
| JNI | Java Native Interface |
| JSR | Java Specification Request |
| JVM | Java Virtual Machine |
| JXTA | Juxtapose |
| LDAP | Lightweight Directory Access Protocol |
| MAF | Managed Add-In Framework |
| MAMS | Multi-Authority Modular System |
| MBean | Managed Bean |
| MISD | Multiple Instructions Multiple Data |
| MMS | Module Management System |
| MPI | Message Passing Interface |
| MPMD | Multiple Programs Multiple Data |
| MQL | Management Query Language |
| MTTF | Mean Time To Failure |
| MUC | Multi-User Chat |
| NA(P)T | Network Address (and Port) Translation |
| NP | Nondeterministic Polynomial |
| NUMA | Non-Uniform Memory Access |
| OG | Organic Grid |
| OGSA | Open Grid Services Architecture |
| OLDB | Outbound Lemma Database |
| OS | Operating System |
| P2P | Peer-to-Peer |
| P2PMP | Peer-to-Peer Management Protocol |
| P2PS | Peer-to-Peer Simplified |
| P3 | Personal Power Plant |
| PaaS | Platform as a Service |
| PB | Pseudo-Boolean |
| PC | Personal Computer |
| QoS | Quality of Service |
| RBT | Red-Black Tree |
| RFC | Request For Comments |
| RMI | Remote Method Invocation |
| RTP | Real-time Transport Protocol |
| SaaS | Software as a Service |
| SASL | Simple Authentication and Security Layer |
| SAT | Satisfiability |
| SDIMS | Scalable Distributed Information Management System |
| SIMD | Single Instruction Multiple Data |
| SOA | Service Oriented Architecture |
| SPEC | Standard Performance Evaluation Corporation |
| SPMD | Single Program Multiple Data |
| SpoVNet | Spontaneous Virtual Network |
| SQL | Structured Query Language |
| SRM | Scalable Reliable Multicast |
| SVM | Simple View Manager |
| SWAN | Sandboxing Without A Name |
| TAG | Tiny Aggregation (Service) |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| TSP | Traveling Sales Person |
| UDP | User Datagram Protocol |
| UML | Unified Modeling Language |
| URL | Uniform Resource Locator |
| USD | US Dollar |
| USL | Universal Scalability Law |
| V-Node | Virtual Node |
| VM | Virtual Machine |
| VO | Virtual Organization |
| VPN | Virtual Private Networking |
| VSIDS | Variable State Independent Decaying Sum |
| W3C | World Wide Web Consortium |
| WAN | Wide Area Network |
| WQR | Work Queue with Replication |
| X-MUX | eXtensible MUC |
| XEP | XMPP Extension Protocol |
| XMPP | eXtensible Messaging and Presence Protocol |

# Abstract

Although today's largest Desktop Grid harvests idle cycles from only 0.46‰ of the Personal Computers (PC) deployed world-wide, it is way ahead of the currently fastest supercomputer with respect to raw computing performance. If it were possible to attract roughly 7% of the world's PC owners to donate their resources, the resulting virtual supercomputer would *right now* punch through the exascale barrier expected to be broken by supercomputers not until around the year 2020.

However, the full potential of Desktop Grid Computing has not yet been unleashed in another respect: application support. Due to their centralized interaction model Desktop Grids are currently limited to embarrassingly parallel applications. By complementing the foundations of Desktop Grid Computing systems with Peer-to-Peer concepts and methods, their scope can be extended to non-trivial applications from the field of High-Performance Computing, like parallel search problems – including discrete optimization, constraint satisfaction, and satisfiability solving –, Raytracing, or N-Body simulations. These applications are all instances of a special class of parallel applications called *Irregularly Structured Problems* (ISP). Their computation and interaction patterns are input-dependent, unstructured, and evolving.

The incorporation of Peer-to-Peer methods has impact on many aspects of Desktop Grid Computing systems: Their architecture has to be retrofitted to support decentralized operation by multiple authorities in a secure and safe environment. The plethora of algorithmic alternatives available beyond Client/Server interaction requires the system to be designed for extensibility from the ground up. Solving task-parallel ISPs requires much more sophisticated platform support in the form of a distributed task pool that is able to perform dynamic decomposition, load balancing, and termination detection in a decentralized and fault-tolerant way. To support this decentralized execution model the underlying network substrate must provide efficient Peer-to-Peer unicast and multicast primitives and the ability to rapidly report available resources and their vanishing, both without seriously impairing scalability.

COHESION, the next generation Desktop Grid Computing platform described in this thesis, is an amalgamation of novel approaches designed to tackle these challenges. It's capacity to efficiently execute task-parallel ISPs in volatile and heterogeneous Desktop Grids is demonstrated by means of SATCIETY, a state-of-the-art distributed SAT solver build on top of COHESION.

# Zusammenfassung

Die Zahl der weltweit in Betrieb befindlichen Einzelplatzrechner steigt jedes Jahr um etwa 12%. Nach aktuellen Prognosen [Shi08] wird erwartet, dass deren Zahl bis zum Jahr 2014 zwei Milliarden übersteigen wird. Andere Studien [Mut92] belegen, dass die Rechner einen Großteil der Zeit unausgelastet, ja häufig sogar ungenutzt sind. Das sich daraus ergebende Potenzial ist enorm. In der Folge hat sich mit dem Desktop Grid Computing eine Disziplin des Grid Computing herausgebildet, in der die freien Ressourcen der unausgelasteten Rechner genutzt werden, um rechenintensive Probleme zu lösen. Bis in welche Leistungsdimensionen Desktop Grids vorstoßen können, zeigt das *Folding@home* [Sta10] Projekt, das mit einer maximalen (realen) Leistung von über 7 petaFLOPS den aktuell schnellsten Supercomputer *TianHe-1A* mit einer maximalen Leistung von rund 2.6 petaFLOPS an Leistung deutlich übertrifft. Obwohl diese Zahlen nur begrenzt vergleichbar sind, da unterschiedliche Anwendungen zur Ermittlung der Leistung herangezogen werden, kann man doch sagen, das Desktop Grids eine ernst zu nehmende Alternative zu klassischem Supercomputing darstellen. Ein ganz wesentlicher Aspekt sind dabei die deutlich geringeren Kosten, die seitens des Ressourcennutzers entstehen. Während für *TianHe-1A* Anschaffungskosten von 88 Millionen USD anfielen und jährlich Betriebskosten von deutlich über 10 Millionen USD anfallen, schlägt der Betrieb des *Folding@home* Desktop Grids mit lediglich einigen Hunderttausend USD zu Buche, die Anschaffungskosten sind mit einigen Zehntausend USD vernachlässigbar. Dies erklärt sich vor allem damit, dass sich die Besitzer der Einzelplatzrechner selbst um Anschaffung und Wartung ihrer Systeme kümmern.

Allerdings unterscheiden sich Desktop Grids in fundamentaler Weise von klassischen parallelen Systemen: Zunächst sind die Knoten des Grids nicht permanent verfügbar. Bedingt durch das Verhalten des Nutzers und sporadisch auftretende Fehlerzustände treten Knoten dem Grid in unvorhersehbarer Weise bei oder treten ebenso unvorhersehbar aus diesem aus. Dieses Phänomen wird *Volatilität* [BSV03, WSH99] genannt. Des Weiteren zeichnen sich die Ressourcen eines Desktop Grids durch ein hohes Maß an Heterogenität aus. So können beispielsweise die Taktfrequenzen der CPUs in einem Desktop Grid durchaus zwischen 179 MHz und 3 GHz variieren [KTB+04]. Schließlich werden Desktop Grids meist über Wide-Area Netzwerke betrieben, sodass die Kommunikationskosten uneinheitlich sind und die Konnektivität durch Firewalls und private Netzwerke eingeschränkt wird. Volatilität, Heterogenität und eingeschränkte Konnektivität in Verbindung mit uneinheitlichen Kommunikationskosten machen Desktop Grids zu einer der herausforderndsten parallelen Systemumgebungen. Trotzdem hohe Effizienz zu erzielen, stellt System- und Applikationsdesigner vor schwierige Aufgaben. Daher werden Desktop Grids heute überwiegend für die Lösung trivial-paralleler Probleme genutzt. Diese zeichnen sich dadurch aus, dass sie auf einfache Art und Weise im Vorfeld der eigentlichen Berechnung in einzelne unabhängige Teilprobleme zerlegt werden können, die dann zur Berechnung auf die Knoten des Desktop Grids verteilt werden. Allerdings erlaubt die Natur vieler wichtiger Applikationen keine derartige Vorgehensweise.

Dazu zählen unter anderem parallele Such- und Optimierungsprobleme sowie physikalische Kleider- und Vielkörpersimulationen. Um das volle Potenzial des Desktop Grid Computing auszuschöpfen, ist eine Erschließung weiterer Applikationsklassen jenseits der trivial parallelen Anwendungen notwendig. Eine wichtige Klasse von Problemen, die in diese Kategorie fallen, sind die sogenannten taskparallelen *irregulär-strukturierten Probleme* (ISP). Der Ablauf der Berechnung und die dabei auftretenden Interaktionsmuster sind bei diesen Problemen stark eingabeabhängig, unstrukturiert und dynamisch [SW03]. Insbesondere ist die Laufzeit der Teilprobleme unvorhersehbar. Dies macht ein dynamisches Vorgehen erforderlich, das die Zerlegung in Teilprobleme während der Berechnung bei Bedarf vorsieht. Bestehende Desktop Grid Systeme können dies aufgrund ihrer zentralen Organisation nicht leisten, da sie je nach Rechnerzahl und Problemgröße früher oder später an die Grenzen ihrer Skalierbarkeit stoßen.

Wie in Abbildung Z.1 dargestellt, leistet die vorliegende Arbeit Beiträge in mehreren Gebieten. Zentral ist dabei die systematische Erweiterung des Desktop Grid Ansatzes um Methoden aus dem Bereich der Peer-to-Peer Systeme, mit dem Ziel deren Anwendungsradius auf nicht-triviale Applikationen aus dem Bereich des High-Performance Computing auszuweiten. Die Auswirkungen dieses Schritts werden auf allen Ebenen des Systems analysiert und dieses, wo nötig, durch neue Ansätze ergänzt. Die dabei gewonnenen Erkenntnisse wurden im Rahmen einer *State-of-the-Art* Desktop Grid Computing Middleware namens COHESION umgesetzt und erprobt.

Die im Folgenden vorgestellten wissenschaftlichen Beiträge dieser Arbeit wurden in Zeitschriften [SBHD08, SBH09, SBP10, SB10b] veröffentlicht und auf internationalen Konferenzen [BDS06, SB07, SBP09, SB11] und Workshops [SB10a] präsentiert. Außerdem wurden große Teile COHESION's unter Open Source [ios] und Forschungslizenzen [coh] der Öffentlichkeit zugänglich gemacht.

## Mikrokern-basierte Erweiterbarkeit

Frühe Desktop Grid Computing Projekte, wie *SETI@home* [Uni] und *distributed.net* [Dis], waren als monolithische Systeme ausgelegt – Plattform- und Anwendungsfunktionalität waren untrennbar miteinander verbunden (vgl. Abbildung Z.2a). Die mangelnde Flexibilität dieses Ansatzes führte zu dessen Ablösung durch flexiblere Desktop Grid Computing Middleware-Lösungen, die eine simultane Ausführung mehrerer Anwendungen erlauben (vgl. Abbildung

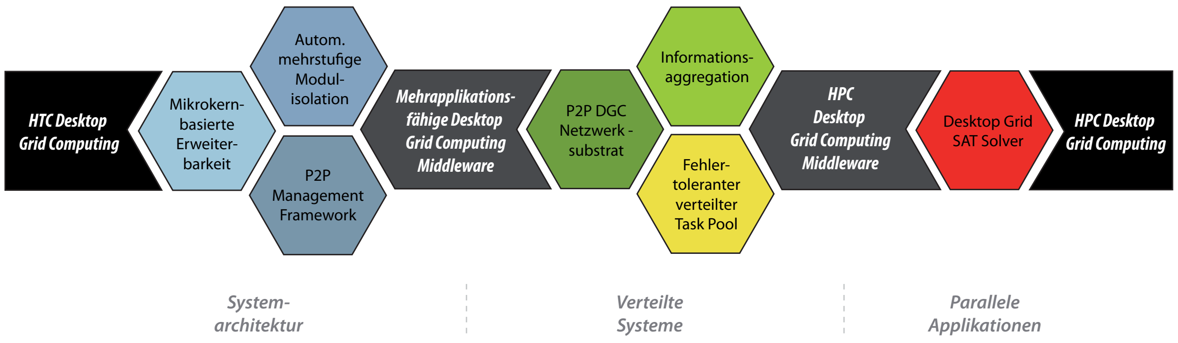
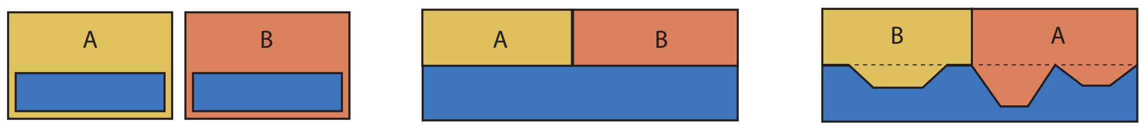

HTC Desktop Grid Computing
Mikrokern-basierte Erweiterbarkeit
Autom. mehrstufige Modul-isolation
P2P Management Framework
Mehrapplikations-fähige Desktop Grid Computing Middleware
P2P DGC Netzwerk-substrat
Informations-aggregation
Fehler-toleranter verteilter Task Pool
HPC Desktop Grid Computing Middleware
Desktop Grid SAT Solver
HPC Desktop Grid Computing
System-architektur
Verteilte Systeme
Parallele Applikationen

**Abbildung Z.1:** Beiträge der vorliegenden Arbeit



**(a)** Monolithisch **(b)** Middleware **(c)** Mikrokern

**Abbildung Z.2:** Entwicklung der Architektur von Desktop Grid Plattformen vom monolithischen System über Middleware-basierte Ansätze mit Unterstützung für die simultane Ausführung mehrerer Anwendungen (A,B) bis hin zu COHESION's mikrokern-basiertem Ansatz

Z.2b). Der wohl bekannteste und heute am weitesten verbreitete Vertreter dieses Ansatzes ist die *Berkeley Open Infrastructure for Network Computing* (BOINC) [Uni10]. Das sehr gut verstandene Client/Server-Interaktionsmodell dieser Plattformen ermöglicht die gesamthafte Bereitstellung von weitgehend applikationsunabhängiger Funktionalität durch die Plattform. Im Gegensatz dazu ergibt sich durch den Übergang zum Peer-to-Peer-Interaktionsmodell, wie er in dieser Arbeit vorgeschlagen wird, bezüglich jedes Systemaspekts eine Vielzahl möglicher Alternativen. Diese können unmöglich alle durch die Plattform bereitgestellt werden. Daher ist Erweiterbarkeit eine der wichtigsten Eigenschaften von Desktop Grid Plattformen der nächsten Generation. COHESION wird dieser Anforderung durch die Umsetzung des durch das Betriebssystem *Mach* [ABB+86] bekannt gewordenen Mikrokern-Architekturmusters gerecht [SBHD08].

## Peer-to-Peer Management und Automatische Mehrstufige Modulisolation

Neben seiner technischen Bedeutung steht der Begriff *Peer-to-Peer* auch für ein Organisationsprinzip, das allen an einem System partizipierenden Teilnehmern dieselben Rechte einräumt und dieselben Pflichten auferlegt. Eine unmittelbare Konsequenz der Anwendung dieses Prinzips auf das Desktop Grid Computing liegt darin, dass der Betrieb des Desktop Grids nicht mehr in der Hand einer einzelnen Autorität liegt, sondern zur gemeinsamen Aufgabe der Eigentümer der teilnehmenden Rechner, den Infrastrukturanbietern und den Applikationsbetreibern wird. Solch ein *Multi-Authority Environment* stellt neue Anforderungen an die Management- und Sicherheitsinfrastruktur. Diesbezüglich liegen die Beiträge der vorliegenden Arbeit in einem Isolationssystem für Modulsysteme namens I-OSGI und in einem Management Framework für große Systeme volatiler Ressourcen.

I-OSGI [SB11] implementiert einen neuartigen Ansatz zur Isolation von Modulen untereinander und zwischen Modulen und dem umgebenden Hostsystem. Ausgehend von implizit oder explizit spezifizierten Isolationsvorgaben berechnet I-OSGI automatisiert eine Systemkonfiguration, in der jedem Modul eine Isolationsumgebung derart zugeordnet ist, dass alle Isolationsvorgaben erfüllt sind. Der Grad der Isolation kann dabei durch den Einsatz unterschiedlicher Isolationstechniken variiert werden. Die Konfigurationen werden dabei über eine Kombination aus SAT-basierter Optimierung und Graphfärbealgorithmen mit dem Optimierungsziel minimaler Ressourcennutzung bestimmt. Die Skalierbarkeit des Systems wurde im Rahmen einer Serie von Mikrobenchmarks für Systeme nachgewiesen, die aus Hunderten von Modulen bestehen.

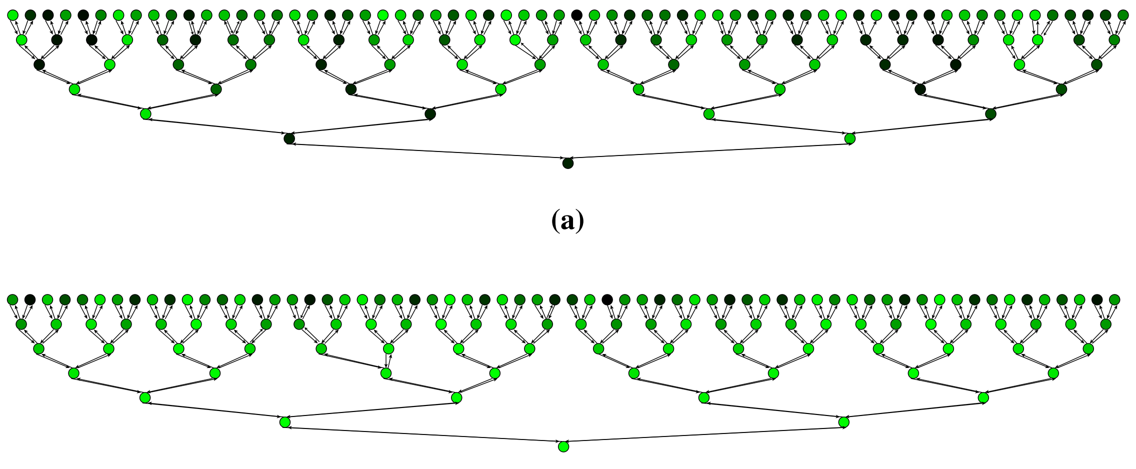
COHESION's Peer-to-Peer Management Framework [SB07] basiert auf dem weit verbreiteten *Java Management Extensions* (JMX) [Sunb] Standard. Multi-Authority Environments werden durch einen Rollen-basierten Zugriffsschutz unterstützt, der es nur dem Eigentümer eines Objektes gestattet, dessen Parameter zu verändern. Das Framework beinhaltet Werkzeuge, um eine große Zahl von Objekten virtuell auf einer Peer zusammenzuführen und deren Managementschnittstellen zusammenfassen, sodass die Einstellungen aller Objekte durch Ausführung einer einzigen Operation angepasst werden können. Auf diese Weise können Managementaufgaben effizient für eine große Anzahl von im System verteilten Objekten durchgeführt werden. Ein Dienst zur Ausführung von Management-Skripten erlaubt die Abarbeitung von wiederkehrenden Aufgaben. Skripte können darüber hinaus auch als Mobile Agenten entsandt werden, um Managementoperationen auf entfernten Peers durchzuführen. Dies auch dann, wenn die Peer zeitweise nicht verfügbar ist.

## Netzwerksubstrat für High-Performance Desktop Grid Computing

Die Anforderungen die High-Performance Computing Anwendungen an das Netzwerksubstrat stellen, unterscheiden sich fundamental von denen existierender Desktop Grid und Peer-to-Peer Anwendungen: Zunächst erfordert die engere Kopplung bei High-Performance Computing Anwendungen, dass Peers direkt miteinander kommunizieren können. Das zentrale Organisationsmodell bestehender Desktop Grid Plattformen, bei dem stets der Umweg über einen zentralen Masterknoten gegangen werden muss, ist hierfür ungeeignet. Des Weiteren muss die Verfügbarkeit bzw. der Ausfall einer Peer unverzüglich erkannt und gemeldet werden, sodass die Last effizient verteilt und Fehlerzustände so schnell wie möglich kompensiert werden können. Da zahlreiche verteilte Algorithmen erheblich von der Verfügbarkeit einer effizienten Multicast-Kommunikationsoperation profitieren, sollte das Netzwerksubstrat eine solche anbieten. Schließlich muss ein geeignetes Substrat effizient innerhalb von Wide-Area-Netzen arbeiten, die sich durch nicht-einheitliche Kommunikationskosten und eingeschränkte Konnektivität auszeichnen. Da bestehende Substrate diese Anforderungen nicht oder nur teilweise erfüllen, besteht ein signifikanter Beitrag dieser Arbeit in der Bereitstellung eines auf offenen Standards basierenden Netzwerksubstrates, das die genannten Anforderungen erfüllt: ORBWEB [SBP09, SBP10] gehört zur Klasse der hybriden Peer-to-Peer Netzwerke da es Teile der Funktionalität an ausgezeichnete besonders leistungsfähige Peers, sogenannte *Superpeers*, delegiert. Dazu gehört insbesondere das Management von Prozessgruppen, innerhalb derer ORBWEB verschiedene virtuelle Topologien zur Verfügung stellen kann. Dadurch wird der Trade-off zwischen dem Umfang der auf den Peers bereitgestellten Information über die Zusammensetzung der Gruppe und ihrer Skalierbarkeit einstellbar. ORBWEB basiert auf dem *eXtensible Messaging and Presence Protocol* (XMPP) [xsf] und kann so von der Dynamik einer großen und aktiven Community profitieren. ORBWEB's Leistungsfähigkeit wurde mittels einer umfassenden experimentellen Analyse bestätigt. Dabei wurden Gruppen bis zu einer Größe von über 10.000 Knoten bezüglich des Peer- und Superpeer-seitigen Ressourcenverbrauchs unter realen Bedingungen untersucht.

## Knoteneigenschaftsbewusste Informationsaggregation

Informationsaggregation ist die Zusammenfassung von Information innerhalb eines verteilten Systems. Um Skalierbarkeit zu erreichen, organisieren moderne Aggregationssysteme die Knoten des Systems in einem Aggregationsbaum entlang dessen Kanten die Information eingesammelt, zusammengefasst und mit unterschiedlichem Abstraktionsgrad – je nachdem, an welchem Knoten innerhalb des Baumes die Information abgenommen wird – angeboten wird. Zahlreiche fundamentale Problemstellungen in verteilten Systemen lassen sich über Aggregationsverfahren lösen [vR03]. Dazu gehören Leader Election, Dienst- und Ressourcenplatzierungprobleme und Wiederaufsetzungsverfahren nach Fehlerzuständen. Trotz seiner zentralen Bedeutung gibt es zahlreiche ungelöste Probleme bezüglich des Entwurfs und der Implementierung von Aggregationssystemen [RM06]. Speziell, die Kosten zur Rekonfiguration des Aggregationsbaumes und zur Evaluation komplexer Reduktionsfunktionen können weniger leistungsfähige Knoten überfordern und damit die Leistung des Gesamtsystems beeinträchtigen. COHESION stellt eine Lösung für dieses Problem bereit [SBH09], bei der Eigenschaften wie die Leistungsfähigkeit und die Stabilität eines Knotens bei seiner Platzierung innerhalb des Aggregationsbaumes berücksichtigt werden, sodass besonders leistungsfähige und stabile Knoten nahe der Wurzel und auf mehreren Ebenen des Aggregationsbaums platziert werden. Dadurch wird die durch die Aggregation entstehende Last nicht wie bei existierenden Systemen gleichmäßig, sondern entsprechend ihrer Leistungsfähigkeit und Stabilität über die Knoten des Systems verteilt. Es konnte gezeigt werden, dass die resultierende Besetzung des Aggregationsbaumes höhere Qualität aufweist, als dies bei agnostischen Verfahren der Fall ist (siehe Abbildung Z.3).



**(a)** / **(b)**

**Abbildung Z.3:** Vergleich der Allokationsqualität für ein Konstruktionsverfahren (a) ohne und (b) mit Berücksichtigung der Leistungsfähigkeit der teilnehmenden Rechner (helle Knoten stehen für leistungsstarke, dunkle Knoten für leistungsschwache Rechner)

## Fehlertoleranter Verteilter Taskpool

Die Problemdekomposition, also die Art und Weise, wie ein Problem in Teilprobleme zerlegt wird, spielt eine sehr wichtige Rolle in der Parallelisierung. Sie kann statisch vor der eigentlichen parallelen Berechnung oder dynamisch während der Berechnung erfolgen. Im letzteren Fall werden bei Bedarf sogenannte *Tasks* als Objekte erster Ordnung erzeugt und dynamisch freien Prozessoren zur Ausführung zugeteilt. Da die Laufzeit einer Task bei irregulär-strukturierten Problemen a priori nicht bekannt ist, führt statische Problemdekomposition zu signifikanten Effizienzeinbußen, die sich im Fall zu feiner Granularität aus dem Mehraufwand der Prozessierung unnötig vieler Einzeltasks, im Fall zu grober Granularität aus Leerlaufzeiten ergeben. Daher muss die Dekomposition dynamisch erfolgen. Im Task Pool Modell sind Problemdekomposition und Lastverteilung durch eine Datenstruktur entkoppelt, in der dynamisch erzeugte Tasks abgelegt werden können. Bestehende Desktop Grid Plattformen implementieren meist einen zentral organisierten Task Pool, der wenig komplex ist, sich dafür aber als ineffizient im Zusammenspiel mit dynamischer Dekomposition erweist. COHESION's Peer-to-Peer-Interaktionsmodell [SBHD08] ermöglicht die Umsetzung eines dezentralen Ausführungsmodells (siehe Abbildung Z.4), das auf einem verteilten Task Pool beruht. Dabei ist jedem Prozessor eine lokale Warteschlange für Tasks zur Seite gestellt mit deren Hilfe Problemdekomposition und Lastverteilung autonom durchgeführt werden können. Diese Dezentralisierung bringt aber auch einen deutlichen Anstieg der Komplexität mit sich, da die intrinsische Fehlertoleranz[1] des zentralen Modells verloren geht. COHESION verbirgt diese Komplexität hinter einer generischen Task Pool Abstraktion [SB10b]: Lastverteilung, Fehlertoleranzmaßnahmen und Terminierungserkennung werden dabei transparent durch die Plattform durchgeführt. Dabei wird die durch ORBWEB bereitgestellte Gruppenabstraktion

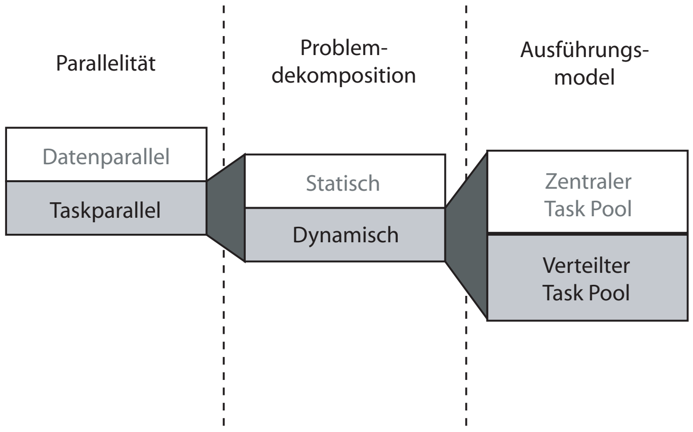


**Abbildung Z.4:** COHESION's Ausführungsmodell für taskparallele irregulär-strukturierte Probleme

1 Fehlertolerant ist ein zentral organisierter Taskpool nur solange die zentrale Koordinationsinstanz, die einen *Single Point of Failure* darstellt, nicht von einem Ausfall betroffen ist. Die Zuverlässigkeit des Gesamtsystems kann durch Replikation dieser kritischen Komponente über bestehende Verfahren, wie *Paxos* [Lam98], erhöht werden.

genutzt, um die gleichzeitige Verwendung mehrerer Task Pools durch unterschiedliche Applikationen innerhalb desselben Desktop Grids zu ermöglichen. Für die Terminierungserkennung wurde dabei ein neues Verfahren entwickelt, das unempfindlich gegen Taskduplikation ist, die aufgrund des asynchronen Verhaltens des zugrunde liegenden Systems auftreten kann. Dazu werden die skalaren Gewichte des klassischen Terminierungserkennungsalgorithmus von Mattern [Mat89] durch Gewichtsintervalle ersetzt, sodass deren Addition idempotenten Charakter erhält. Der Taskpool wurde einer umfangreichen experimentellen Analyse unterworfen. Dabei konnte eine sehr gute schwache Skalierbarkeit (96% parallele Effizienz bei 160 Knoten) unter realen Bedingungen für bis zu 160 Knoten nachgewiesen werden.

## SAT Solving im Desktop Grid

Das Erfüllbarkeitsproblem der Aussagenlogik (engl. *satisfiability*, SAT) ist das Entscheidungsproblem, ob zu einer gegebenen booleschen Formel eine erfüllende Belegung existiert. SAT war das erste Problem, für das NP-Vollständigkeit nachgewiesen wurde [Coo71]. Anwendungen finden sich in zahlreichen wichtigen Bereichen, wie dem Entwurf von logischen Schaltungen [VB01], der künstlichen Intelligenz [KS92], des Scheduling [CB94] und der Kryptography [MM00]. Trotz intensiver Forschung konnten seit der Erfindung der CDCL-Techniken um die Jahrtausendwende keine wesentlichen Fortschritte hinsichtlich der Beschleunigung bestehender SAT-Solving Verfahren mehr erzielt werden. So gibt es heute eine große Zahl von Problemen, die mit heutigen sequenziellen Techniken nicht zu lösen sind. Mit der Einführung von Multicore-Architekturen tauchten auch parallele Solver für gemeinsamen Speicher auf [HJS09a]. Signifikante Beschleunigungen sind aber erst bei Verfügbarkeit von Manycore-Architekturen mit einer erheblich höheren Zahl von Kernen als heute üblich zu erwarten. Der nächste logische Schritt sind deshalb massiv-parallele Ansätze für Architekturen mit verteiltem Speicher und Hunderten oder gar Tausenden von Prozessoren. SATCIETY [SB10a, SB10b] ist ein erster Vertreter dieser neuen Klasse von parallelen SAT-Solvern. SATCIETY nutzt COHESION's verteilten Taskpool und ORBWEB's Peer-to-Peer Kommunikationsmittel zur Umsetzung des bislang einzigen fehlertoleranten verteilten SAT-Solvers, der für das heterogene und volatile Umfeld der Desktop Grids geeignet ist und dabei dennoch in der Lage ist substantielle Beschleunigungen im Vergleich zu modernen sequenziellen SAT-Solvern zu erzielen. Dafür wurden zahlreiche Neuerungen in SATCIETY integriert. Dazu zählen ein adaptives Verfahren zum Wissensaustausch zwischen Solverkernen, das die Topologie des physikalischen Netzwerkes bei der Festlegung der Austauschraten berücksichtigt, die Bereitstellung der teilweise sehr großen Formeln ($> 100$ MB) über Peer-to-Peer Protokolle und ein kompaktes binäres Datenformat, sowie ein mehrstufiges Verfahren zum Speichermanagement, das die Vollständigkeit des Solvers – also die Fähigkeit zum Nachweis der Unerfüllbarkeit einer Formel – erhält. Die Leistungsfähigkeit wurde in einem Desktop Grid unter realen Bedingungen nachgewiesen. Dabei konnte trotz hoher Heterogenität der genutzten Ressourcen ein signifikanter mittlerer Speedup gegenüber dem schnellsten teilnehmenden Rechner von 14.5 bei 40 Prozessoren für langlaufende unerfüllbare Instanzen erzielt werden.

## Ausblick

Die Frage, ob das Desktop Grid Computing und die in diesem Bereich gewonnenen Erkenntnisse auch in 20 Jahren noch von Interesse sein werden, ist legitim. Vieles spricht dafür: Zunächst sind die hier vorgestellten Erkenntnisse bezüglich des Umgangs mit volatilen und heterogenen Ressourcen auch für andere aufstrebende Anwendungs- und Forschungsgebiete relevant. Dazu zählen Multi-/Manycore-Architekturen, Cloud Computing und Exascale Computing. Wie eingangs dieser Zusammenfassung ausgeführt, ist das Potenzial der Desktop Grids bei Weitem nicht ausgeschöpft. Das enorme Angebot an Rechenleistung steht einer stetig wachsenden Nachfrage gegenüber. Allein in Deutschland hat diese im Zeitraum von 2005 bis 2010 um das 80-fache zugenommen [BHL05]. Da dieser Zuwachs die von David House in einem Korollar zu *Moore's Gesetz* [Moo75] vorhergesagte Verdoppelung der Leistungsfähigkeit von Mikroprozessoren alle 18 Monate deutlich übersteigt, ist anzunehmen, dass Rechenleistung auch in Zukunft knapp bleiben wird. Vor diesem Hintergrund ist nicht anzunehmen, dass das enorme Potenzial des Desktop Grid Computing ungenutzt bleiben wird. Das Interesse an Desktop Grid Computing wird aber noch aus einem anderen Grund weiter steigen: *Butter's Law* [Teh00] prognostiziert eine Verdoppelung der Bandbreite von Glasfasern alle neun Monate. Hält diese Entwicklung an, werden Prozessoren in 20 Jahren 1000-mal, die Netzwerke jedoch eine Million Mal so schnell sein wie heute. Damit rücken weitere enger gekoppelte Applikationsklassen, die heute nur auf Supercomputern sinnvoll ausgeführt werden können, in die Reichweite der Desktop Grids.

# Part I

# Introduction

# 1 The Subject Matter

Today's pervasiveness of information technology results in a plethora of exploitable computing power. This is true for dedicated supercomputing systems, personal computers, and mobile devices: The performance of the best, the worst, and the aggregate computing power of all supercomputers listed in the TOP500 table [TOP10] has been growing exponentially for almost 20 years now. The number of personal computers in use worldwide is growing by 12% every year and is expected to exceed two billion in 2014 [Shi08]. The number of mobile phones is just as impressive – it reached 4.6 billion at the end of 2009 [UN 10].

For supercomputers this steady growth is stimulated by an ever increasing demand for computing power. This is particularly true in the areas of science and engineering where important *grand-challenge* problems exist that are not solvable with today's supercomputers. One example is accurate long-time climate modeling. It's goal is to predict the consequences of global warming by conducting large-scale long-term simulations with high spatial and temporal resolution that require supercomputers that are up to one million times faster than the currently fastest supercomputers [DeB05].

## 1.1 Grid Computing

*Grid Computing* has become a viable tool to narrow the gap between supply of and demand for computing power by aggregating networked compute resources across administrative domains. The first definition of the term *Grid Computing* was given by Ian Foster and Carl Kesselman [KF98] in 1998 before Grids as we know them today actually emerged:

> *»A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.«*

A few years later they refined their definition [FKT01] introducing so called *Virtual Organizations* (VO):

> »[Grid computing is] *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations.«*

They also gave a more precise notion of resource sharing:

> *»The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily,*

*highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization.«*

With Grid Computing becoming a hot topic and an increasing commercial appeal, a trend became apparent to call everything from cluster management systems to network file systems a Grid. In response to this development, Foster proposed a three item *Grid Checklist* [Fos02], according to which a Grid is a system that:

1. coordinates resources that are not subject to centralized control
2. uses standard, open, general-purpose protocols and interfaces
3. delivers nontrivial qualities of service (like response time, throughput, availability, security, and co-allocation of multiple resource types to meet complex user demands)

The early vision of Grid Computing to enable users to access computing power and other resources as simple as attaching a device to the power grid has not been fulfilled until now. Grid systems as envisioned by Foster *et al.* are usually far too complex to setup and maintain for non-expert individuals. Consequently, two distinct types of Grids have emerged: *Service Grids* and *Desktop Grids*.

### 1.1.1 Service Grids

Operating Service Grids is usually reserved for larger institutions that are able to employ professional system administrators that take care of the complex Grid environment consisting of hard-, middle-, and software. Today, an increasing number of middleware solutions for Service Grids (including the most widely used *Globus Toolkit* [Fos06] and *gLite* [Conb]) adhere to a common standard called the *Open Grid Services Architecture* (OGSA) [FKS+05]. OGSA is developed within the *Global Grid Forum* (GGF) and has been evolved from ideas of Foster *et al.* [FKNT02]. OGSA-based Service Grids are service-oriented grid computing environments composed of interoperable *Web Services* representing the available resources and subsystems. The OGSA specification defines a set of services for identity management, authentication and authorization, service level agreement negotiation and monitoring, management and communication within virtual organizations, integration of data resources into computations, managing and monitoring collections of services, etc. Prominent large-scale Service Grids are *D-Grid* [D-G], *NorduGrid* [SEE+03], and *EGEE* [Cona]. The latter is the largest Grid in the world consisting of over hundred thousand processors. However, deploying and maintaining a Grid Computing infrastructure is a complex and costly effort and is thus economically worthwhile only for large organizations like big companies, research laboratories, universities, and governments.

### 1.1.2 Desktop Grids

The other kind of Grid Computing is based on Desktop Grids. *Desktop Grid Computing* [CBK+08] aims at harnessing the idle resources of personal computers instead of supercomputers for tackling resource intensive problems. Due to its inherent deployment and maintenance cost, Service Grid Computing technology has not been adopted for Desktop Grid Computing. Instead, custom platforms with close to zero deployment and maintenance overhead have emerged that are used for both small-scale installations comprising the workstations of a department [CCEB03] for example, as well as for large-scale Internet-wide setups [And04].

The potential of Desktop Grid Computing is enormous. Studies have revealed that personal computers are frequently idle or at least not fully utilized: While this had been true 75% of the time two decades ago [Mut92], this number has probably increased since multi-core CPUs and high performance GPUs have become standard for personal computers. As can be seen from Table 1.1, Desktop Grids are able to deliver enormous computing power significantly higher than that of today's most powerful supercomputers (for certain application classes) at a fraction of the costs. Moreover, as people buy or upgrade and maintain their hard- and software regularly, the resources of a Desktop Grid are not only largely self-financing, but also self-updating and self-maintaining. However, there are downsides: First, the FLOPS/Watt ratio of a commercial-off-the-shelf (COTS) computer is lower than that of a supercomputer built from optimized hard- and software components. As can be seen from the second row of Table 1.1, this fact translates to a significantly higher relative energy-consumption of

| | *Folding@home* | *TianHe-1A* |
|---|---|---|
| Processors | 460k active machines | 14k CPUs + 7k GPUs |
| Power Consumption | 2.8 MW/petaFLOPS (> 19.6 MW) | 1.6 MW/petaFLOPS (4 MW) |
| Initial Costs (operator/donator) | few $k (server farm) / $460 mil. ($1k/machine) | $88 mil. / - |
| Annual Costs (operator/donator) | few $100k (server farm) / $60 mil. ($130 for power per machine and year) | $2.7 mil. (power) + $10-$20 mil. (salaries for 200 operators) / - |
| Maximal Performance[1] | **> 7 petaFLOPS**[2] | **2.57 petaFLOPS** |

**Table 1.1:** Key fact comparison for *Folding@home* [Sta10], the world's largest Desktop Grid, and *TianHe-1A*, the leader of the TOP500 list [TOP10] of the fastest supercomputers in the world (FLOPS is an acronym for **FL**oating point **OP**erations per **S**econd)

1 The *maximal performance* is the maximal performance achieved on a real application. Note that the application is different for both systems (a protein folding application for *Folding@home* and *LINPACK* for the *TianHe-1A*).

2 The reported number is the number of *x86 FLOPS* that is computed by totaling the number of operations required on an x86 processor for each native operation. *x86 FLOPS* have been introduced by *Folding@home* [Sta10] when GPUs were added to the system that can perform some operations much more efficiently than an x86 CPU.

Desktop Grids. The Desktop Grid community is aware of this deficiency and initial efforts to tackle the problem [SE10] are underway as part of the *Desktop Grids for International Scientific Collaboration* (DEGISCO) project [conc]. Second, for Internet-wide Volunteer Computing projects like *Folding@home* resource donors have to be attracted and retained through public relations and incentive systems. Third, there are security issues threatening the host systems of resource donors and the Desktop Grid infrastructure servers.

However, in view of the fact that the enormous potential of the Desktop Grid approach is largely untapped – the largest Desktop Grid *Folding@home* today spans less than 0.46‰ of the PCs available worldwide in 2008[1] – these drawbacks seem negligible: If a project would be able to attract approximately 7% of the resources available in 2008 the resulting virtual supercomputer would approach the exascale barrier expected to be broken by supercomputers not until around the year 2020 [DBM+11]. According to Gartner [Shi08] a strong growth in the emerging markets will push the number of PCs in use world-wide well beyond two billion in 2014. Thus, Desktop Grid Computing projects will be able to harness resources with dozens of exaFLOPs of overall performance.

## 1.2 Characteristics of Desktop Grids

Exploiting the huge potential of Desktop Grid Computing for a broad spectrum of applications is difficult for two reasons: First, the ecosystem around Desktop Grid Computing has become increasingly fragmented in the past: There is a vast yet still growing number of Desktop Grid Computing projects (for a comprehensive list see [Pea]) and many of them had deployed their own ad-hoc infrastructure. As a result of this development resource contributors are likely to be put off by the need to pollute their system with clients for many different platforms. While the evolution of early Desktop Grid projects like *GIMPS* [Gim], *SETI@home* [ACK+02], and *Distributed.net* [Dis] into the *Berkeley Open Infrastructure for Network Computing* (*BOINC*) middleware [Uni10] with support for multiple applications has been an important step towards resolving this issue, its lack of support for applications beyond embarrassingly parallel computations makes it unlikely that *BOINC* becomes a *universal* platform for Desktop Grid Computing in the same way as the *Globus Toolkit* became for Grid Computing.

Second, Desktop Grids differ significantly from other types of parallel systems. Particularly, the aggregated resources join and leave the Grid in an unpredictable manner. This phenomenon is called *volatility* [BSV03, WSH99] and is much more pronounced than in other kinds of parallel systems like compute clusters. The reasons which result in a multiplication of possible error sources are manifold: non-dedication of resources, reduced system isolation, and typically lower reliability of hardware and software system components due to lack of redundancy. Hence, a Desktop Grid system not only needs to handle occasional error conditions but must be explicitly tailored to cope with a constant flux in resource availability. Another major difficulty is heterogeneity: While cluster nodes are most often virtually identical, the nodes of a Desktop Grid are different concerning hard- and software

1 460k of over a billion PCs [Shi08]

configuration. The actual differences can be substantial: The CPU clock rates within an Institutional Desktop Grid at the *San Diego Supercomputer Center* have been reported to range from 179 MHz up to 3 GHz [KTB+04]. To complicate matters further, resource usage may be constrained by the host owner. Finally, Desktop Grids are typically operated over wide area networks (WAN) or the Internet. This causes non-uniform communication costs and often comes with restricted connectivity between the participating hosts due to network address (and port) translating (NA(P)T) devices and restrictive firewalls.

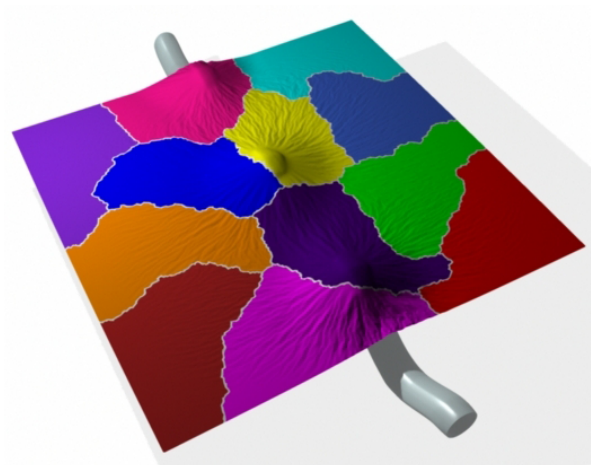
In summary, the system properties of Service and Desktop Grids are fundamentally different. Hence, approaches trying to unify both by dissolving one into the other have little prospect of success. However, recent research [FHL+08, FKBG10] is concerned with integrating Service and Desktop Grids into a single system by providing bridges both from Service to Desktop Grids and vice versa (see Chapter 6).

## 1.3 High-Performance Desktop Grid Computing

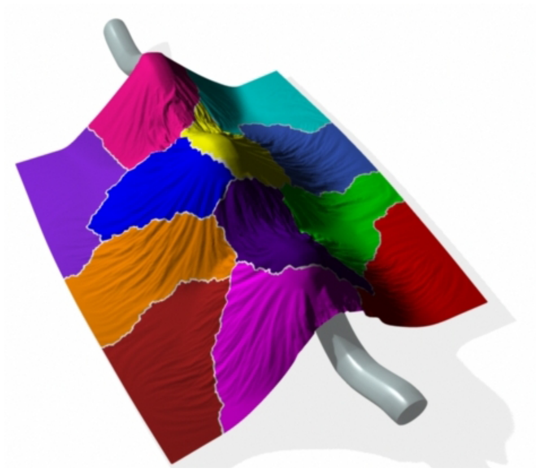
Volatility, heterogeneity, and partially restricted connectivity with non-uniform communication costs turn Desktop Grids into one of the most challenging environments for parallel computing. Delivering sustained computing power in this setting poses enormous challenges to system and application designers. As a consequence, existing Desktop Grid applications are most often *embarrassingly parallel*. The input to such applications can be decomposed into independent subproblems, which can be farmed out for computation without further communication among the hosts processing individual subproblems. Hence, Desktop Grid Computing has been a tool mainly used for *High Throughput Computing* (HTC) that is about delivering large amounts of processing capacity over an extended period of time without efficiency being a major concern. However, many important applications are different in so far as they cannot be solved (efficiently) without direct inter-host communication. Examples are parallel search and optimization problems or parallel N-Body simulations. In contrast to the embarrassingly parallel class of applications, they belong to the class of *High Performance Computing* (HPC) applications. This discipline of parallel computing is about delivering large amounts of computing power for short periods of time with high efficiency. Extending the scope of Desktop Grid Computing towards HPC applications would further increase the utility of the approach and would make Desktop Grids qualify as a competitive alternative to expensive supercomputing and Grid resources.

Not every HPC application can be efficiently executed on top of a Desktop Grid. A decisive factor is the degree of coupling. Tightly-coupled payloads do not match well with the (comparatively) low bandwidth high-latency network link characteristics in WANs and the Internet. More suitable are loosely-coupled yet not embarrassingly parallel applications. An important representative of this kind of application are task-parallel *Irregularly Structured Problems* (ISP). ISPs are parallel applications whose computation and interaction patterns are input-dependent, unstructured, and evolving [SW03]. Prominent examples are all kinds of parallel search problems – including discrete optimization, constraint satisfaction, and satisfiability solving –, raytracing, N-Body problems, and physically-based cloth simulation.

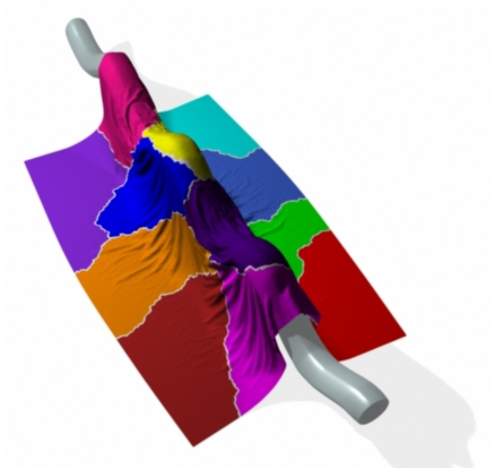
However, the environmental (volatile and heterogeneous resources, restricted connectivity, and non-uniform communication costs) and ISP-specific (unpredictable task size and com-



**(a)**



**(b)**

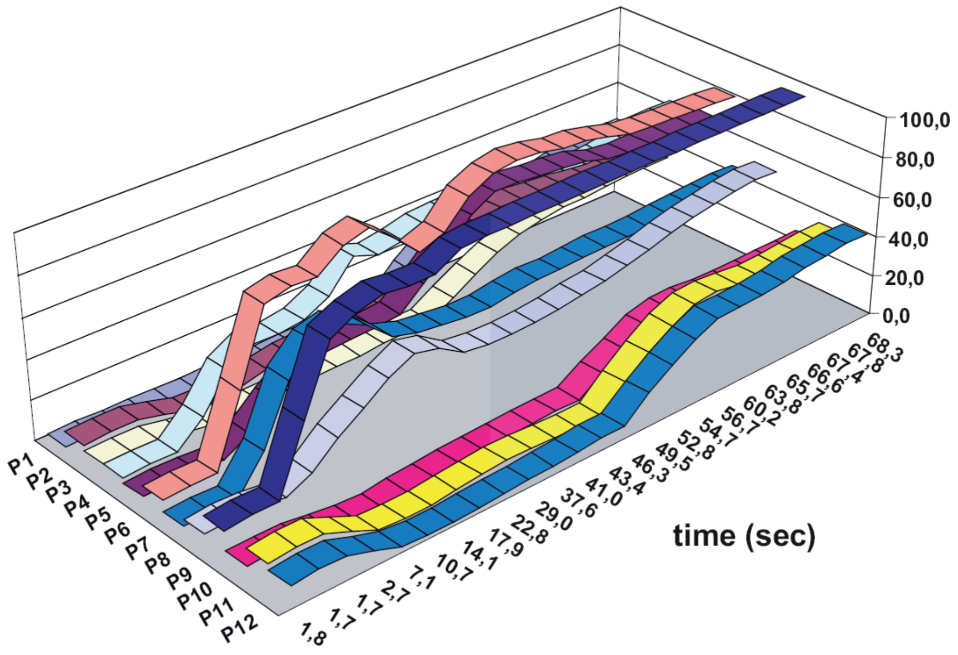


**(c)**

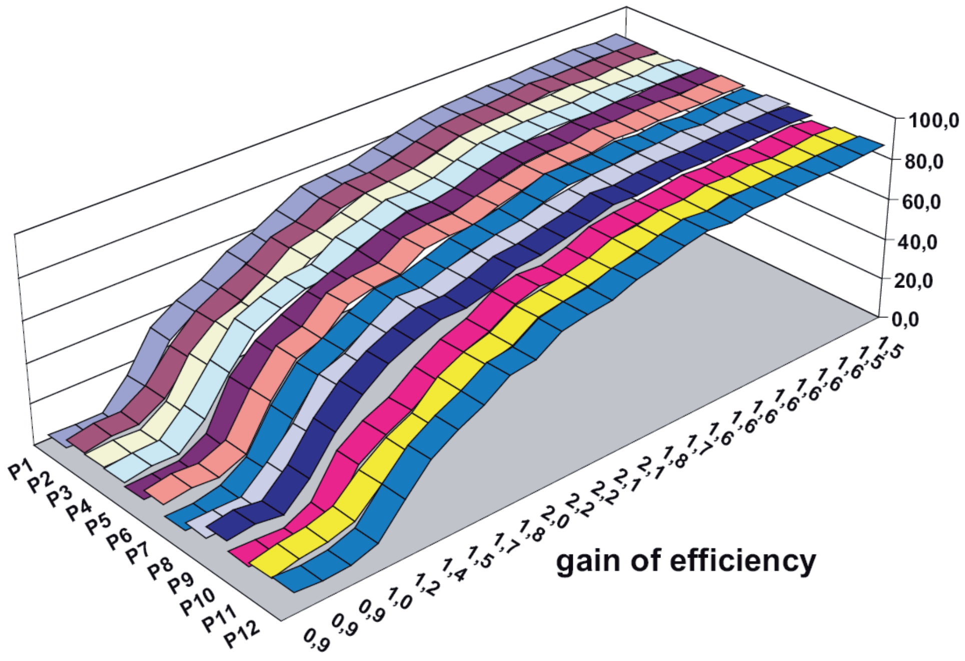
**Figure 1.1:** Multiple stages of a physically-based simulation of a square piece of cloth with 15K vertices draping over an undulated bar (colors in Figures (a)-(c) indicate mesh partitioning for parallel physical modeling).

munication patterns, inter-task dependencies) characteristics make solving ISPs on Desktop Grids a challenging task (see Figure 1.3) that requires much more sophisticated platform support than required for solving embarrassingly parallel applications: Due to the fact that the execution time of a subproblem is no longer related to the size of the input, subproblems can no longer be created in advance and scheduled trivially. Instead, a technique called *dynamic load balancing* [GGKK03] has to be applied that redistributes the workload as the computation progresses. This way processors that become idle can be assigned new work that is taken from busy processors.

Creating tasks in advance is also problematic as finding the optimal decomposition depth in the face of unpredictable subproblem runtime, fluctuating resource availability, and resource heterogeneity is impossible. Hence, some depth has to be chosen that most likely won't



**(a)**



**(b)**

**Figure 1.2:** Effect of dynamic problem decomposition and load balancing on the parallel efficiency of the collision handling phase (x-axis $\hat{=}$ $i$-th processor, z-axis $\hat{=}$ CPU usage in %).

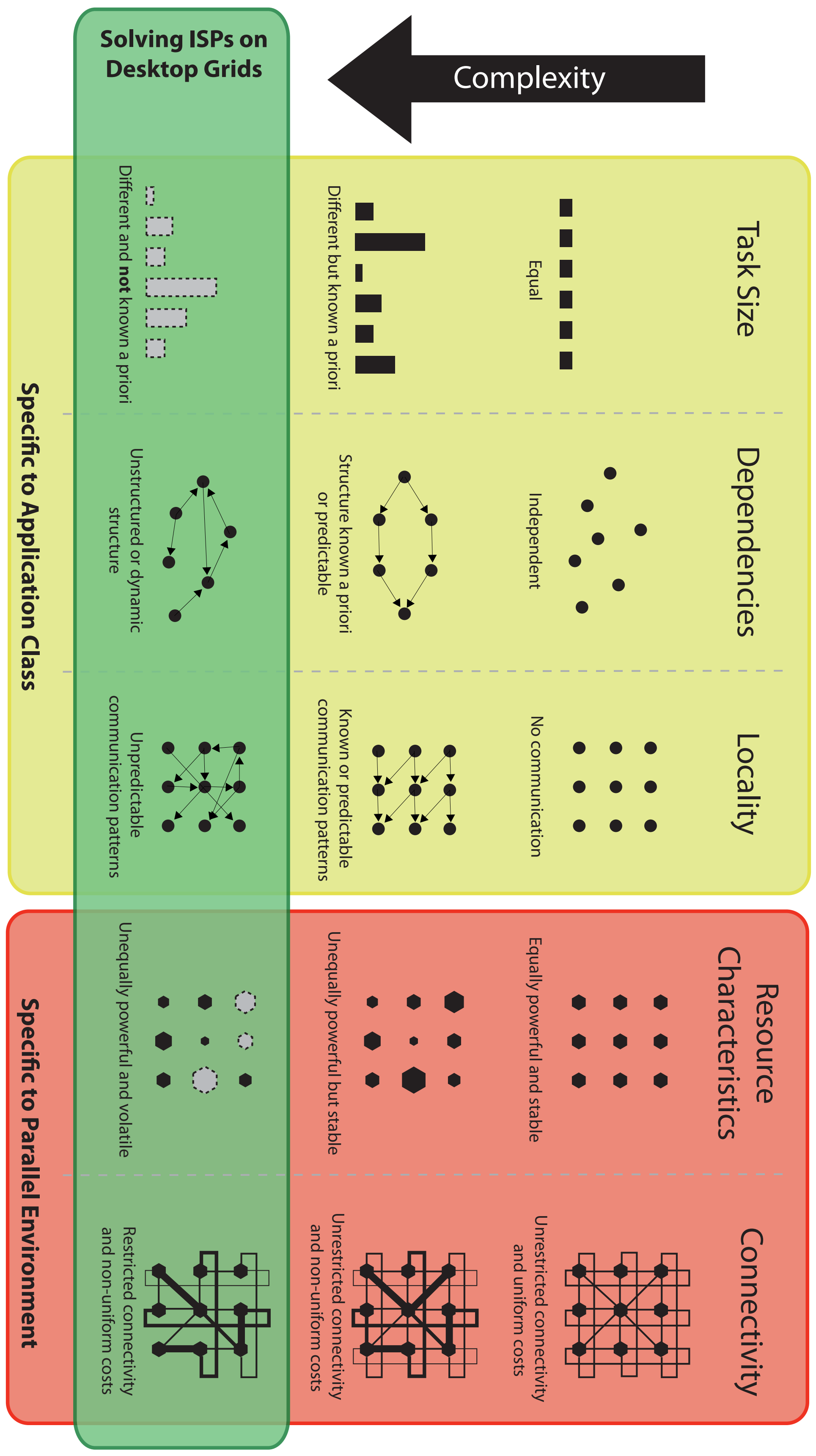

Complexity
Solving ISPs on Desktop Grids
Task Size
Dependencies
Locality
Resource Characteristics
Connectivity
Equal
Different but known a priori
Different and **not** known a priori
Independent
Structure known a priori or predictable
Unstructured or dynamic structure
No communication
Known or predictable communication patterns
Unpredictable communication patterns
Equally powerful and stable
Unequally powerful but stable
Unequally powerful and volatile
Unrestricted connectivity and uniform costs
Unrestricted connectivity and non-uniform costs
Restricted connectivity and non-uniform costs
Specific to Application Class
Specific to Parallel Environment

**Figure 1.3:** The combination of volatile and heterogeneous processors with partially restricted connectivity and non-uniform communication costs with unknown subproblem size, unpredictable communication patterns, and a dynamic dependency structure makes solving ISPs on Desktop Grids very challenging.

be optimal. In case too few tasks are created, processors become idle when the number of available tasks drops below the number of processors. Creating too many tasks, in contrast, induces overhead caused by task migration between processors. Both conditions have a negative impact on efficiency. Thus, a second technique called *dynamic decomposition* [GGKK03] is applied that consists in decomposing the problem on-demand according to the workload distribution during runtime.

The consequences of ignoring the fundamentally different nature of solving ISPs become evident when we look at the collision detection phase of a parallel physically-based cloth simulation[1] [TB07]. As can be seen from Figure 1.1, the density and spatial distribution of rigid collisions and self-collisions evolve over time. This results in pronounced imbalance between processors when static decomposition is employed (see Figure 1.2a). By applying dynamic decomposition and dynamic load balancing, significant efficiency gains can be achieved (see Figure 1.2b).

The requirement for dynamic load balancing has an immediate consequence: As subproblems may be large, direct communication is mandatory to migrate subproblems efficiently. The most severe limitation of existing Desktop Grid Computing middlewares in this regard is their centralized *interaction model*. The term *interaction model* refers to the way the nodes of a Desktop Grid interact when executing a parallel application employing certain parallel algorithms and parallel programming models. *BOINC* – like most of today's Desktop Grid platforms – is tailored to applications with a Client/Server interaction model, which is characterized by a fixed star-shaped communication topology that is suitable for trivial Master/Worker-style parallel applications with independent tasks such as bag-of-tasks, parameter sweep/study, and monte carlo simulation. Although applications with a significant amount of inter-node communication could be executed on top of this topology in principle as well, messages exchanged between two clients or as part of collective communication operations among multiple clients had to be relayed by the server, leading to a bottleneck that severely limits efficiency and scalability.

Hence, a pivotal step towards HPC-enabled Desktop Grid Computing capable of solving ISPs is the replacement of the centralized interaction model of existing platforms with a more powerful *Peer-to-Peer*[2] (P2P) interaction model. As discussed in the following chapter, this involves reconsidering design decisions on all levels of the parallel system. In this thesis, we present the results of this reconsideration paving the way for the transition to a new breed of HPC-enabled Peer-to-Peer Desktop Grid Computing platforms.

1 Parallel collision detection is done using the *strict multi-threading* programming model [BK03] of the parallel system platform *DOTS* [BKWb98].

2 In our context, the term *Peer-to-Peer* refers to the ability of the system to collectively provide a service by direct interaction between peers, with limited use of intermediary or coordinating entities. According to the definitions given in [Sch01], this kind of architecture/interaction model is more precisely termed *hybrid Peer-to-Peer* (cf. Chapter 13).

# 2 Contributions

As depicted in Figure 2.1, this thesis makes contributions in several areas, in particular systems architecture, distributed systems, and parallel applications. Most importantly, it describes how the foundations of Desktop Grid Computing can be complemented with Peer-to-Peer concepts to extend its scope to non-trivial applications from the field of High Performance Computing. The impact of this transition is analyzed on all layers of the system and novel approaches are proposed where existing approaches fall short. The results of this analysis are amalgamated in COHESION a state-of-the-art P2P Desktop Grid Computing middleware stack depicted in Figure 2.2.

The individual scientific contributions summarized subsequently and discussed in detail in the following chapters have been published in scientific journals [SBHD08, SBH09, SBP10, SB10b], and have been presented at international conferences [BDS06, SB07, SBP09, SB11] and workshops [SB10a]. In addition, large parts of COHESION's codebase have been made available to the public under Open Source [ios] and research licenses [coh].

## 2.1 Microkernel-Based Extensible System Core

Early Desktop Grid Computing applications like *SETI@home* [Uni] and *distributed.net* [Dis] were monolithic systems mixing up platform and application functionality in a non-detachable manner (cf. Figure 2.3a). Due to the resulting inflexibility and an emerging trend towards fragmentation of the Desktop Grid ecosystem, these early systems were superseded by more flexible Desktop Grid Computing middleware – most notably *BOINC* [And04] – designed to support multiple concurrently executing applications (cf. Figure 2.3b). The well understood Client/Server interaction model of applications deployed on these platforms allows to provide a comprehensive set of platform functionality that is largely application independent. The transition to a Peer-to-Peer interaction model as proposed in this thesis in contrast results in a plethora of options on every layer of the system. For example, the selection of an appropriate groupcast algorithm for a given application depends on a large number of factors, including network topology, expected communication load, and required quality of service (QoS) properties. Generally, the design space for Peer-to-Peer Desktop Grid systems and applications becomes highly multidimensional and thus is considerably larger than the design space of platforms with a Client/Server interaction model. Consequently, providing a comprehensive toolbox serving all conceivable application requirements is no longer possible. Hence, a key purpose of next generation Desktop Grid platforms must be to provide a set of generic reusable components for common application aspects that may be replaced and supplemented with extensions contributed by and tailored to the needs of a specific application. Similar to the evolution in the field of operating systems, next generation Peer-to-Peer computing systems must be specifically designed to cope with this kind of

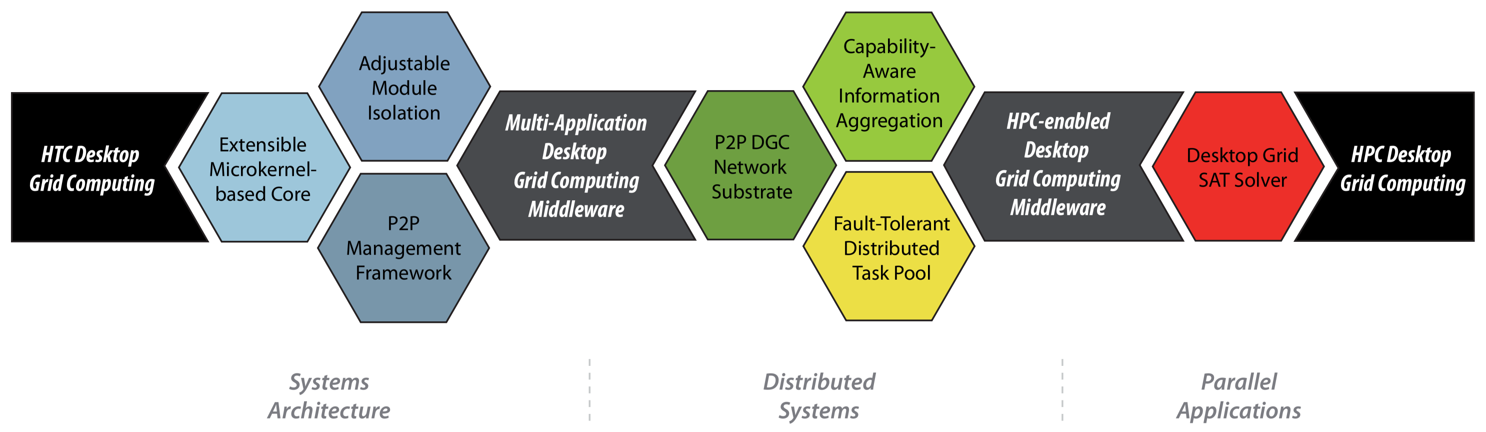

HTC Desktop Grid Computing
Extensible Microkernel-based Core
Adjustable Module Isolation
P2P Management Framework
Multi-Application Desktop Grid Computing Middleware
P2P DGC Network Substrate
Capability-Aware Information Aggregation
Fault-Tolerant Distributed Task Pool
HPC-enabled Desktop Grid Computing Middleware
Desktop Grid SAT Solver
HPC Desktop Grid Computing
Systems Architecture
Distributed Systems
Parallel Applications

**Figure 2.1:** The contributions of this thesis regarding the paradigm shift from Client/Server to Peer-to-Peer models

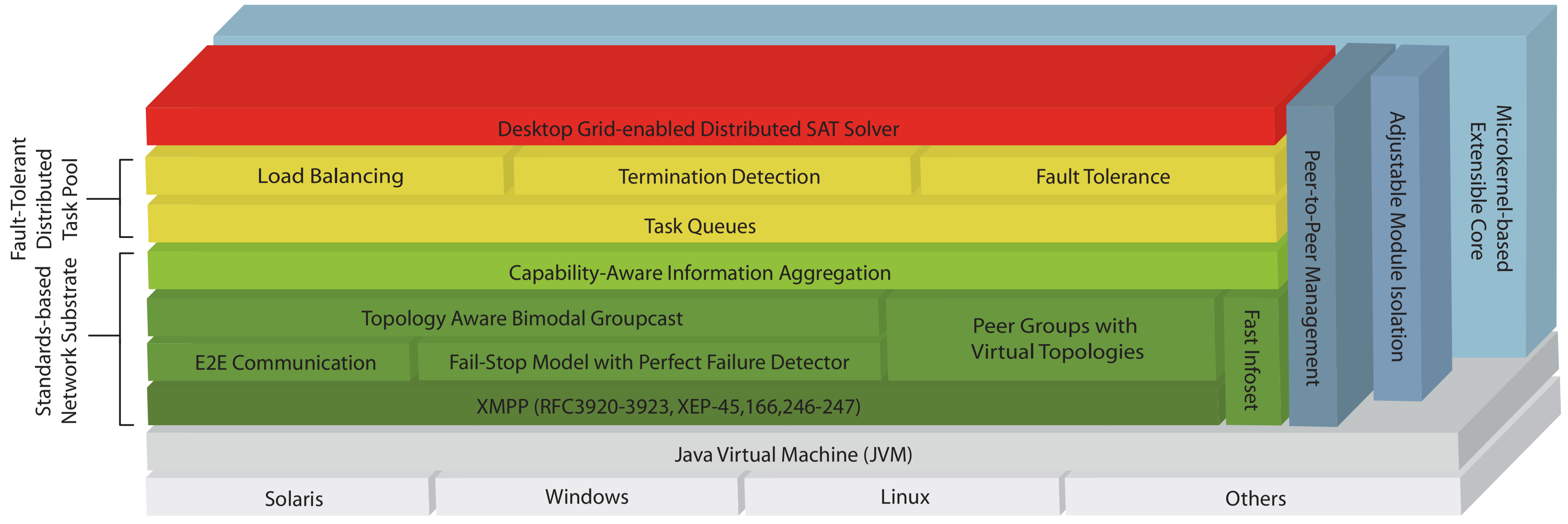
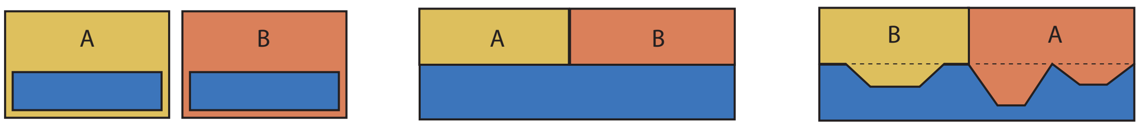

Fault-Tolerant Distributed Task Pool
Standards-based Network Substrate
Desktop Grid-enabled Distributed SAT Solver
Load Balancing
Termination Detection
Fault Tolerance
Task Queues
Capability-Aware Information Aggregation
Topology Aware Bimodal Groupcast
Peer Groups with Virtual Topologies
E2E Communication
Fail-Stop Model with Perfect Failure Detector
XMPP (RFC3920-3923, XEP-45,166,246-247)
Fast Infoset
Peer-to-Peer Management
Adjustable Module Isolation
Microkernel-based Extensible Core
Java Virtual Machine (JVM)
Solaris
Windows
Linux
Others

**Figure 2.2:** The layered architecture of the COHESION Peer-to-Peer Desktop Grid Computing middleware



**(a)** Monolithic **(b)** Middleware **(c)** Microkernel

**Figure 2.3:** Evolution of the software architecture of Desktop Grid platforms from monolithic system designs over middleware approaches with multi-application support to COHESION's microkernel-based approach.

application specific customization at the system level. Hence, by adopting the *mircokernel* architectural pattern known from the *Mach* operating system [ABB+86], COHESION brings modularity – which is considered a cornerstone in modern software architecture – to the core of Desktop Grid Computing platforms (cf. Figure 2.3c) [SBHD08].

## 2.2 Peer-to-Peer Management and Adjustable Module Isolation

The term *Peer-to-Peer* – besides its technical meaning – stands for an organizational principle granting participating entities the same rights and imposing the same duties. A direct consequence of adopting this principle for Desktop Grid Computing is that operating the Desktop Grid is no longer a task performed by a single authority, but becomes a collaborative effort of host owners, infrastructure providers, and application providers. This kind of multi-authority scenario introduces new challenges in the fields of application manageability, security, and safety that have not yet been addressed in the context of P2P Desktop Grid Computing. In this regard, the contributions of this thesis are a sophisticated isolation system called I-OSGI and a management approach for large-scale systems based on established standards.

I-OSGI [SB11] allows for adjustable inter-module isolation providing a tunable trade-off between isolation degree and resource consumption. System configurations with minimal resource usage and inter-module communication overhead are computed for up to hundreds of modules in a fully automated way by leveraging state-of-the-art optimization techniques. The novel approach of I-OSGI supersedes existing application sandboxing approaches as it is able to protect the host system *and* modules from malicious or erroneous collocated modules with minimal consumption of host system resources.

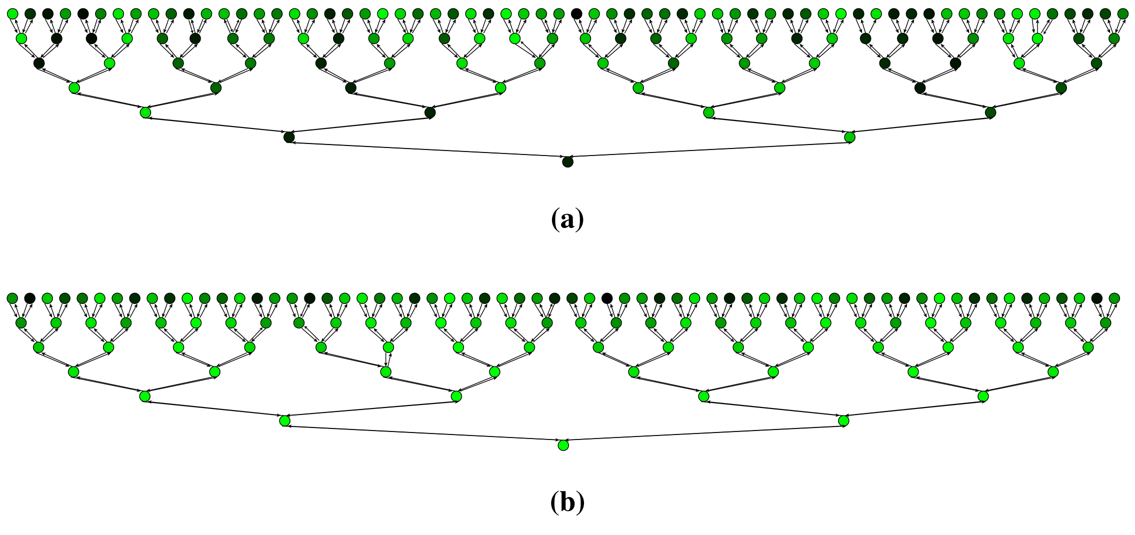
COHESION's Peer-to-Peer management framework [SB07] is based on the industrial grade *Java Management Extensions* (JMX) [Sunb] management standard. Multi-authority environments are supported through a role-based management approach that allows for restricting access of a party to its own manageable objects. The framework provides tools to transparently cascade and aggregate the management interfaces of large numbers of remote objects and allows for managing offline peers through scripting. The management services are made accessible over an integrated management workbench based on the *Eclipse* IDE [Ecl].

## 2.3 Standards-Based Network Substrate

The requirements of HPC applications on the network layer are fundamentally different from those of both existing Desktop Grid and traditional P2P applications: First, the tighter coupling between subproblems in HPC applications necessitates that peers are able to communicate directly, avoiding the bottlenecks of existing Desktop Grid Computing middleware with centralized interaction models. Second, instant updates concerning resource availability are required to efficiently distribute load among the peers and to allow for quick compensation of the effects of peer failures. Third, implementing advanced distributed algorithms requires extended communication mechanisms most importantly an efficient one-to-many (multicast) communication primitive. Finally, the substrate has to provide its entire functionality within WANs or over the Internet with heterogeneous resources (links and hosts) and restricted connectivity. As existing network substrates fall short in satisfying these requirements, a significant contribution of this thesis is ORBWEB [SBP09, SBP10], a standards-based hybrid Peer-to-Peer network substrate centered around a powerful peer group abstraction with a tunable trade-off between the extent of information provided to peers about the structure of the group and its scalability. By adopting and adapting the IETF *eXtensible Messaging and Presence Protocol* (XMPP) [xsf] standard for this purpose, Desktop Grid Computing can gain momentum from the impetus of a large and influential community.

## 2.4 Capability-Aware Information Aggregation

Information aggregation is the process of summarizing information across the nodes of a distributed system. Existing hierarchical approaches based on aggregation trees can provide information with different levels of detail by progressively summarizing data along the edges



(a)

(b)

**Figure 2.4:** Comparison of the allocation quality for (a) capability-agnostic and (b) capability-aware aggregation tree maintenance (lighter/darker nodes represent more/less capable nodes)

of a spanning tree. Many fundamental aspects of distributed systems can be realized using information aggregation [vR03], including leader election, voting, service and resource placement, multicast tree formation, and error recovery. Despite its importance, there are numerous open challenges in the design and implementation of aggregation systems that are considered worthy of future research [RM06]. In particular, the cost of reconfigurations caused by high node volatility and frequent evaluation of complex reduction functions can overwhelm less powerful nodes. COHESION includes a solution to this problem [SBH09] that embraces the heterogeneity across nodes prevalent in Desktop Grids by distributing the onus of aggregation *not* uniformly across the nodes of the system as done in existing approaches but according to their capabilities with respect to performance and stability. We have shown experimentally that the resulting allocation quality is superior to that of capability-agnostic approaches (see Figure 2.4). The many-to-one communication (reduction) primitive provided by the aggregation system complements the primitives provided by ORBWEB.

## 2.5 Fault-Tolerant Distributed Task Pool

Problem decomposition plays a central role in the design of parallel applications as it determines how the problem is divided into (sub-)tasks to be executed in parallel. Basically, problem decomposition can be carried out statically before the actual application execution or in a dynamic manner creating tasks on demand at runtime. In the latter case, tasks are first-class objects that can be dynamically assigned to idle processors for execution. As for ISPs a task's runtime cannot be determined *a priori*, a static decomposition approach can result in significant processor idling. This has been demonstrated by means of a parallel cloth simulation application in Chapter 1. Thus, dynamic problem decomposition is mandatory. This in turn necessitates explicit load balancing. The *task pool model* decouples problem decomposition and load balancing by a data structure that stores task objects created by dynamic decomposition operations. Existing Desktop Grid Computing middlewares use a centralized task pool that is easy to implement but of limited scalability and efficiency when used in conjunction with dynamic problem decomposition. COHESION's Peer-to-Peer

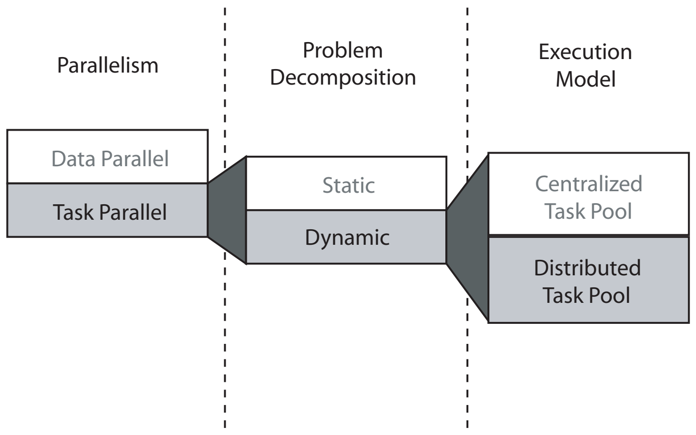


**Figure 2.5:** Execution model for task-parallel ISP applications

interaction model [SBHD08] allows for realizing a decentralized execution model based on the distributed task pool model (see Figure 2.5) with a task queue at each peer performing problem decomposition and load balancing autonomously. However, distributing the task pool comes at the price of losing the intrinsic[1] fault-tolerance of centralized approaches. COHESION provides a generic distributed task pool abstraction [SB10b] that transparently handles load balancing, fault tolerance, and termination detection. The task pool leverages the features of ORBWEB and can be instantiated multiple times to support concurrent execution of multiple applications within the same Desktop Grid. Termination is detected using a novel *weight-throwing* algorithm that is resilient to task duplication. This is achieved by replacing the scalar weights of Mattern's classical termination detection algorithm [Mat89] with weight *intervals* rendering the addition of weights idempotent.

## 2.6 Distributed SAT Solver for Desktop Grids

The *Satisfiability* (SAT) problem is the decision problem whether there is a satisfying solution to a given boolean formula. SAT solving was the first problem shown to be NP-complete in 1971 by Cook [Coo71] and today is an enabling technology for many important application domains. Prominent examples are electronic design automation [VB01], artificial intelligence [KS92], scheduling [CB94], and cryptography [MM00]. Despite this success there has been little progress since the invention of the conflict-driven clause-learning SAT solving algorithm around the turn of the millennium and there are still many problems that can't be solved by today's state-of-the-art sequential solvers. With the advent of multicore CPUs, shared memory parallel SAT solvers have emerged [HJS09a]. However, until manycore CPUs with a significant number of cores are common, they are able to achieve moderate speedups only.

The next frontier in parallel SAT solving are massively parallel approaches that employ hundreds or even thousands of processors with distributed memory to open up new performance dimensions and to tackle problems that are way beyond the capability of today's mostly centrally organized parallel solvers. SATCIETY [SB10a, SB10b] is a first step towards this new breed of parallel SAT solvers able to operate within the highly demanding Desktop Grid environment with heterogeneous and volatile resources yet still yielding substantial speedups compared to a state-of-the-art sequential solver. SATCIETY leverages COHESION's distributed task pool and ORBWEB's Peer-to-Peer communication facilities to realize the first fault-tolerant Desktop Grid-enabled distributed SAT solver with inter-peer adaptive and topology-aware knowledge exchange.

1 As long as the central system components are not affected by a failure.

# 3 Structure of this Thesis

The rest of this thesis is structured as follows: Part II discusses general related work. It is complemented with focused treatments of related work concerning the respective topic in the Parts III-VI, and in Part VIII. Part III describes i-OSGi the adjustable module isolation system which is part of Cohesion's extensible core. Part IV summarizes the Peer-to-Peer management framework used to tackle the challenges related to the management of the large number of entities in multi-authority Desktop Grid environments. Part V is devoted to Orbweb the novel hybrid P2P network substrate. The fundamental concepts of peer groups and virtual peer topologies are introduced and their realization on top of the mature XMPP standard is described. The scalability and efficiency of Orbweb is substantiated with a thorough performance evaluation. Part VI summarizes the information aggregation method that takes the capabilities of peers into account when distributing the onus of performing system-wide information aggregation at a large scale. In Part VII Cohesion's fault-tolerant distributed task pool is discussed that lays the foundations for performing large-scale task-parallel computations in Desktop Grids. Part VIII describes Satciety, a distributed SAT solver realized on top of Cohesion. The part details how the services of the platform are leveraged to tackle various domain-specific challenges, first and foremost scalable knowledge exchange. Satciety represents the state-of-the-art in distributed SAT solving. Finally, Part IX summarizes the results of this thesis and identifies opportunities for future research.

As outlined above the contributions of this thesis are manifold. Due to length restrictions not all contributions can be discussed in full detail. Hence, Part IV and Part VI are summaries of the respective topic. The interested reader is referred to the publications listed on the cover page of the respective part.

For readers not familiar with scalability theory there is a primer in Appendix A. Module definitions and protocol specifications in Parts V and VII use the model and notation introduced by Cachin *et al.* in [CRG11]. A condensed summary of both can be found in Appendix B.

# Part II

# Related Work

During the last 15 years a plethora of Desktop Grid Computing systems have been created and their number is still growing rapidly. The discussion of related work presented in this part is limited to Desktop Grid Computing systems that have at least one aspect or goal in common with COHESION. In particular, we do not cover plain Client/Server-based platforms [CCEB03, And04, BWKW05, CDF+05, KKF+09]. The reader interested in details on this kind of systems is referred to one of the many surveys on the topic [Sar01, CME04, CKB+07, FHL+08, VC08, ZLL11]. The most comprehensive yet still incomplete survey we are aware of proposes a taxonomy based on 28 properties and classifies 16 Desktop Grid Computing systems [CBK+08].

The systems of interest within the context of this thesis covered throughout the three chapters[1] of this part ...

... open up new application classes beyond embarrassingly parallel (also known as trivially parallel) applications. Supported application classes and programming models are fully-strict multi-threading [BBB96], distributed shared memory with custom semantics [BKKW99], *Bulk Synchronous Parallel* applications [Sar99], applications consisting of a finite number of independent and irregular tasks [FKS10], general [WTK07, NH09] and DAG-style [AB07, ACJ10] workflows, as well as *Branch-and-Bound* [NC05] and *Divide-and-Conquer* [vNWJB10] applications. Some systems [KSS05, GR07, LAGS09, LSG10] even try to implement support for tightly-coupled parallel applications using the MPI programming model. However, as indicated by various performance studies, the synchronous nature of MPI doesn't fit well with the volatility and heterogeneity of Desktop Grid environments. Finally, there are research efforts [FHC08, CSKT08] aimed at providing support for data-intensive applications.

... improve scalability by incorporating methods and techniques from the realm of Peer-to-Peer systems. The main weakness of classical systems with respect to scalability is centralized scheduling. Thus, hierarchical [ELvD+96, VNRS02, KSS05, TB05], superpeer-based [TS10, MCT+09, BZH06], and decentralized [MMB03, DJW+03, MK05, LZZ+05, CBL07, FFM06, KNM+07, FFM06, KNM+07, ZYX09, DWH09] scheduling techniques for Desktop Grid Computing systems have been devised. Some approaches [BEDSV04, CBG+06, ZL06] are sensitive to environmental conditions like the degree of volatility or heterogeneity.

... employ Peer-to-Peer concepts on the organizational layer [CM02], implement market-oriented approaches [RN00, LK00, BV01] to attract a larger number of resource donors and foster interoperability between different Desktop Grids [ACJ09], between Desktop Grids and Service Grids [FHL+08, FKBG10], and between Desktop Grids and Clusters [vNWJB10, SBB+11].

COHESION is unique among these projects for several reasons: First of all, it is the only approach capable of solving ISPs in highly volatile environments. While systems like *Atlas*

1 Note that some of the systems described hereinafter contribute in several of these areas. Nevertheless, they appear only once in the following chapters. However, their contributions in the other areas are discussed as well.

[BBB96] and *Javelin* [NC05] support dynamic problem decomposition and load balancing, they are not designed to cope with high volatility. *CONFIIT* [FKS10] on the other hand in principal supports solving irregular problems under heavy churn but with fixed decomposition granularity only and using global knowledge. Second, Cohesion approaches distributed computing in Desktop Grids from an end-to-end perspective addressing challenges on the whole software stack. In this regard, *Ibis* [BMvN+10] and *ProActive* [CDDCL06] are most similar. However, they are not explicitly designed for Desktop Grids and hence do not address specific issues like efficient isolated application execution in multi-authority environments. Finally, Cohesion fosters interoperability from the ground up by facilitating technology based on open standards whenever possible instead of deploying proprietary solutions.

# 4 Application Support

## 4.1 Task-Parallel Applications

*Bayanihan* [Sar98, SH99] is a framework based on a Master/Worker execution model for conducting Desktop Grid related research. It is designed to be extensible on all levels of the system but is *no* multi-application middleware. Using *Bayanihan* an embarrassingly parallel application is reported [Sar98] to be executed with a promising efficiency of 92.8%, however, on a small ensemble of only eight processors. Efficiencies for a fractal image computing application are between 1% and 91% in a 16-node setup [SH99]. This wide range is due to different task granularities that are not dynamically adapted in *Bayanihan*. Later, the *Bulk Synchronous Parallel* (BSP) programming model was implemented on top of *Bayanihan* [Sar99]. BSP applications are structured as a sequence of parallel *supersteps* separated by barrier synchronization. The *Bayanihan* system was complemented with eager scheduling, checkpointing, and sabotage tolerance mechanisms based on two techniques called *majority voting* and *spot-checking* proposed by the same authors [SH99]. Majority voting schedules tasks multiple times and accepts a result only if it is reported by a majority of the processors. Spot checking is a probabilistic method to verify the trustworthiness of workers by precomputing randomly chosen tasks and comparing that result to those returned by the workers. If a mismatch occurs the offending worker is blacklisted. The BSP model was applied to several applications with different degrees of coupling ranging from embarrassingly parallel to tightly-coupled. The resulting efficiencies for the embarrassingly parallel application class are $\approx$ 85% for fractal image computing and $\approx$ 45% for parallel matrix multiplication. Performing a tightly-coupled *Jacobi iteration*[1] in parallel resulted in a 40-fold slowdown.

*Charlotte* [BKKW99] is a parallel virtual machine implementing a shared memory architecture with *Concurrent Read, Concurrent Write Common* (CRCW-Common) semantics. Parallelism is expressed using *routines* which are code snippets embedded into sequential code that are to be executed as tasks in parallel. *CRCW-Common* means that all routines may read from any memory location and write to any common location as long as all writing routines write the same value. Reads are done at the beginning of a parallel step, writes become visible at the end. Routines are assigned to processors using *self-scheduling* in *bunches*. Self-scheduling systems are able to adapt to changing resource availability by having processors actively fetch outstanding tasks in a self service manner. *Bunching*, i.e., the aggregation of several tasks to be handled as a single one with respect to scheduling is used to reduce task assignment overhead, to hide communication latency, and to allow

1 The *Jacobi Method* is an algorithm from the field of numerical linear algebra for solving a certain type of systems of linear equations.

for writing fine-grained applications. (N-1)-resiliency is achieved using *eager scheduling* [KPS90]. *Charlotte* does not allow for dynamic decomposition. The reported efficiency for an embarrassingly parallel application on a setup of 16 stable nodes communicating over a (compared to the power of the compute nodes) high performance network is 91%. The efficiency for four volatile nodes is reported to be 88%.

*Computation Over Network for FIIT* (CONFIIT) [FKS10] is a purely decentralized (N-1)-resilient Peer-to-Peer Computing middleware. It is tailored to applications that are composed of a *Finite number of Independent and Irregular Tasks* (FIIT). Note that FITT applications differ from task-parallel ISP applications in two respects: First, FITT tasks are independent while ISP tasks are not. Second, FIIT tasks are of finite number, while ISP tasks are decomposed dynamically into subtasks as needed. CONFIIT nodes build a logical ring using an existing ring maintenance protocol [PDH00] enhanced with (N-1)-resiliency. A token is routed along the ring that is used for coordination and communication. Like *Bayanihan*, CONFITT does not allow for dynamic decomposition. All tasks of the computation are generated in advance and made available on all nodes. Hence, the impact of task granularity is significant: while generating too many tasks introduces excess computation, generating to few causes processor idling. Guessing a *good* granularity poses a serious challenge to application developers. This is especially true in the context of FITT applications where tasks are irregular with run times that are hard or even impossible to predict. A per-node local scheduler assigns open tasks randomly. When a task is done, the result is sent around the ring using the token, which removes the task from the list of open tasks on all nodes. As a task may be scheduled by multiple nodes concurrently, CONFIIT implements a kind of *randomized eager scheduling*. Termination detection is trivial as all nodes maintain a list of all open tasks. This N-fold replication scheme provides for CONFIIT's (N-1)-resiliency. Obviously, the performance of CONFITT crucially depends on how well the nodes' positions in the overlay matches their geographic distribution. The impact of the quality of this mapping on application performance is evaluated in [SBB+11] on 60 nodes of the *Grid'5000* testbed [BCC+06] for a worst case and an optimum mapping. The authors find that this impact for an application solving the *Langford problem* [Knu08] is negligible. They attribute this interesting fact to CONFIIT's ability to overlap computation and communication enabled by the strategy to immediately schedule a new task when a task is finished without waiting for the token to return.

### 4.1.1 Dynamic Decomposition

*Atlas* [BBB96] is an offspring of the *Cilk-NOW* system [BL97] and the first system that combined adaptive parallelism and resiliency. *Atlas* adopts the *fully-strict multi-threading* also known as the *fork/join* programming model from *Cilk* [BJK+95] in which a program consists of at least one procedure that is comprised of a sequence of tasks. Tasks can spawn *child tasks* and *successor tasks*. The former are eligible to be farmed out for computation by other nodes, the latter are executed locally after all input arguments are available. Random stealing is used to balance load among the nodes of the system. Resiliency is achieved using fully distributed checkpointing that guarantees eventual completion of all tasks but may cause significant overheads as losing a task triggers restoration of the whole subtree rooted at that task.

*Javelin* [NC05] is a distributed computing system for Branch-and-Bound applications like the *Traveling Sales Person* (TSP) problem. Branch-and-Bound algorithms search for an optimum within a set of feasible solutions with exponential cardinality by *pruning* unexplored parts of the search space with solutions having costs above the currently best known optimal solution. Pruning results in irregularity, making Branch-and-Bound applications belong to the class of ISPs. *Javelin* supports adaptively parallel computing, i.e., can cope with volatile resources, by employing a number of previously known techniques: First, tasks are decomposed dynamically until an *atomicity threshold* is reached. Limiting the decomposition depth ensures that the overhead of task management does not impair performance. Second, load is balanced using structured work-stealing along the edges of a centrally managed dynamically repaired host tree. Third, eager scheduling tailored to the specifics of Branch-and-Bound applications is used when a host is unable to get work from its neighbors. Besides host tree management, the node on which the job has been submitted is also responsible for maintaining the currently best bound. On change, the new bound is broadcast by propagating the new bound along the edges of the host tree. A novel aspect of *Javelin* is *dynamic depth expansion* which increases the maximum decomposition depth over time such that hard tasks running for a long time can be decomposed into smaller subtasks towards the end of the computation. The authors report an efficiency of $\approx 56\%$ for 1024 nodes. According to the authors the system is not suitable for highly volatile Desktop Grid environments as one of its fundamental assumptions is that the number of failures/departures is small compared to the number of processors.

### 4.1.2 Divide-and-Conquer

*Ibis* [BMvN+10] is a distributed programming and deployment system that consists of two layers: a high performance communication library called the *Ibis Portability Layer* (IPL) on top of which several programming models have been implemented. And a middleware interoperability layer called the *Java Grid Application Toolkit* (JavaGAT) that is used for deploying and managing distributed applications. *Satin* [vNWJB10] – one of the programming models provided by *Ibis* – supports the execution of divide-and-conquer applications in volatile and heterogeneous environments. Load balancing is done using *cluster-aware random stealing* [vNKB01] which is variant of random stealing that avoids high cross-site bandwidth consumption by emitting a steal request only once to a single remote cluster when a node becomes idle. Fault-tolerance is achieved using a combination of checkpointing and a mechanism called *orphan work saving* [WvNMB05]. An *orphan* is a subtask that has lost its parent task, i.e., the task it was split off from. *Satin* restores parent/child relationships between orphans and their parent and thus preserves the work done for processing orphan tasks using a lightweight recovery protocol. The efficiency of *Satin* has been substantiated in a whole string of experiments [vNKB01, WvNMB05, vNWJB10]. However, *Satin* cannot be used for solving ISPs efficiently. This is because problem decomposition in divide-and-conquer systems is continued until the subproblems become simple enough to be solved by a single processor. When this condition holds is unfortunately undecidable for ISPs.

## 4.2 MPI

*P2P-MPI* [GR07] is a distributed computing platform for fault-tolerant execution of *Message Passing Interface* (MPI) [Pac96] applications on large-scale volatile systems. In contrast to other systems targeted to the same combination of application class and execution environment, *P2P-MPI* fault management [GJR09] is based on active replication [Sch90] rather than checkpointing. Replication is done by having a group of processes act as a single *logical process*. Members of a process group proceed in synchrony controlled by a master node. The master node is replaced by one of the backup nodes in case of failure. The overhead induced by synchronous execution leads to longer runtime which in turn increases the probability for the occurrence of fault during the computation. Based on this observation, the authors describe how the optimum replication degree can be computed when the system's failure trace and some application-dependent metrics are known. Unfortunately, no experimental comparison to a system based on a checkpointing approach is given. However, the fact that $n$-fold replication on $p$ processors limits the achievable speedup to $p/n$ for replication-based systems makes it likely that checkpointing approaches are superior.

*VolpexPyMPI* [LSG10] is a Python-based MPI library claimed to be capable of executing unaltered MPI programs in Desktop Grid environments. This is made possible by replicating processes and by using pull-based communication with message logging. With slowdowns between 1.5 and 6.5 the system performs badly compared to the C-based *VolpexMPI* [LAGS09] implementation in case of unreplicated execution. Adding replication results in an additional penalty of up to 50% for threefold replication. The evaluation is flawed in several ways: First, no comparison to standard MPI has been conducted. Second, the results have been obtained in a cluster with homogenous high performance nodes and a high performance interconnect. Hence, the utility of the system within a Desktop Grid environment has to be doubted.

## 4.3 Distributed Objects

*ProActive* [CDDCL06] is a general framework for parallel and distributed computing based on a distributed object programming model with weak migration support and location independent addressing. *ProActive* deploys an unstructured pure Peer-to-Peer network maintained by a custom protocol that establishes a peer list of configurable length. The protocol uses heartbeating to remove failed peers from the list and on-demand random neighborhood exploration to find new peers. Fault-tolerance is based on *rollback-recovery*. Applications can choose from *distributed checkpointing* [CL85] and *pessimistic message logging*[1] depending on their requirements and the properties of the environment they are executed in. Load balancing of active objects is done using a variant of random pushing that balances load between processors only if the randomly selected victim peer[2] is underloaded and its *rank*, that quantifies its relative processing speed, is similar or higher [BJCCP05].

1 A *pessimistic* message logging protocol is one in which no process $p$ ever sends a message $m$ until it knows that all messages delivered before sending $m$ are logged [AM98].

2 The *victim* peer is the peer to which work is pushed.

*ProActive*'s scalability and performance has been substantiated in various experimentations with up-to 2700 processors in Desktop Grid, Service Grid [CDDCL06], and mixed setups [CCM07].

## 4.4 Workflows

*XtremWeb-CH* (XWCH) [AB07] is an upgraded version of *XtremWeb* [CDF+05]. It strives for providing a high performance Peer-to-Peer platform for executing applications that can be expressed as a *Directed Acyclic Graph* (DAG). While the system allows for direct and relayed inter-worker communication and hence allows for executing applications with dependent tasks, the scheduling algorithm is centralized and neither resilient to node failures nor to node departures. Hence, the system's suitability for Desktop Grid environments is limited

*PastryGrid* [ACJ10] broadens the application support of institutional Desktop Grids to tasks with precedences. As the name indicates the system employs a *Pastry* [RD01b] overlay for scalable operation. Scheduling of jobs described by a DAG is done in a decentralized way by having peers processing a given task schedule this task's successors autonomously. The system supports static task graphs only and thus is not suitable for applications with dynamic irregular structure. Fault-tolerance is achieved by a passive replication approach based on *PAST* [RD01a]. The scalability provided on the network layer is not preserved on the coordination layer as *PastryGrid* uses a centralized yet per application different failure detector based on probing. However, the impact of this flaw is not that significant as institutional Desktop Grids are hardly larger than a few thousand nodes – a size centralized architectures can easily handle. Unfortunately, no meaningful performance analysis for the system without failures was conducted. The impact of faults affecting different parts of the system is substantial [ACJM10]: The runtime of an application consisting of 128 parallel jobs (no dependencies) on a testbed comprised of 200 machines is reported to increase by 40% if a single worker peer, 57% if the rendezvous peer, and 15% if the peer performing failure detection fails.

The Peer-to-Peer Desktop Grid system of Wang *et al.* [WTK07] aims at increasing the *dependability* for general workflow-based computations. Dependability in their context is the ratio of the performance on a given problem with and without failures. Their approach is based on a centralized workflow orchestrator that is responsible for translating the workflow into individual tasks that are scheduled as soon as their dependencies are satisfied. Dependability is achieved by a redundant dispatch strategy that assigns workflow tasks multiple times to eliminate performance degradation in case a task gets lost due to a peer failure. The originality of the approach lays in a scheme to dynamically adjust the redundancy level based on the current system state, in particular, the number of available workers, the number of incomplete tasks, and the average failure rate.

*Aneka* [CNJ+07] is a modular service-oriented platform for Desktop Grid Computing. *Aneka* supersedes the *Alchemi* [LBRV05] platform that was restricted to the Master/Worker execution model and a distributed object programming model. In contrast, *Aneka* is designed to be able to balance various types of workloads and to support several different programming models including distributed threads, bag-of-tasks, MPI, and general workflows. It supports multi-application scenarios and achieves extensibility through a configurable container for pluggable services. Beyond these architectural concepts, not much information about the system is available. The parallel efficiencies achieved for a protein-folding bag-of-tasks and a matrix multiplication application on a student lab testbed consisting of 30 non-volatile nodes with a Fast Ethernet interconnect are ≈ 56% and ≈ 53% respectively. These figures may however result from a badly chosen task granularity for the bag-of-tasks application and a general mismatch between tightly-coupled applications like matrix multiplication and the Desktop Grid paradigm.

## 4.5 Data-Intensive Applications

*BitDew* [FHC08] is a data management system that strives for extending the scope of Desktop Grids towards data-intensive applications. *BitDew* supersedes ad-hoc centralized data management solutions of existing Desktop Grid Computing platforms with a scalable fault-tolerant solution for data indexing and distribution based on Peer-to-Peer methods and protocols. *BitDew* provides a framework and an API to transparently utilize existing lookup techniques and data distribution protocols. Key aspects of how to provision a data item are controlled by attached metadata. This includes the replication level, fault-tolerance, i.e., whether to create a new replica when an existing fails, lifetime, affinity, i.e., if a data item should be collocated with another one, and hints about the protocol to be used for distribution. *BitDew* uses a DHT-based data catalog for data indexing and lookup. For a Desktop Grid spanning 250 nodes, the authors report on a 5-fold speedup for an embarrassingly parallel application from the field of biology with a large gene database (≈ 2.5 GB) required on each node. However, this improvement is largely due to the efficiency of the *BitTorrent* [bit, LPP07] protocol used to distribute the database.

Costa *et al.* [CSKT08] compare pure Peer-to-Peer data distribution using *BitTorrent* to a (now discontinued) superpeer-based data caching architecture called *P2P-ADICS*. They argue that whilst the former is better with respect to available aggregate bandwidth, superpeer-based approaches provide superior security as only trusted peers may be promoted to become data caching superpeers. This renders denial-of-service attacks more difficult to stage.

# 5 Scalable Scheduling

## 5.1 Hierarchical Approaches

*JXTA* [TAA+03] is a protocol suite for Peer-to-Peer applications[1] that provides a powerful *peer group* abstraction that can be used to build hierarchical systems. A whole string of projects have been built on top of *JXTA* including *Jalapeno* [TB05], *JNGI* [VNRS02], *Personal Power Plant* [KSS05], *P2P-MPI* [GR07], and *OurGrid* [BAV+07]. The last two have migrated to other network technologies for reasons discussed in Chapter 23. They do not implement a hierarchical approach and are thus presented in Chapter 4 and Section 5.2.

*Jalapeno* [TB05] is a hierarchical approach that uses JXTA peer groups to setup clusters of medium size (up-to 100 peers) called *worker groups* that are managed by a single manager peer. Manager peers are responsible for decomposing tasks in a two step process: First, the job is split into a set of subjobs of which all but one are forwarded to other managers. Second, the subjob kept local is decomposed into tasks to be handled by workers attached to the manager. However, both decomposition steps are not performed on demand but up to a preconfigured granularity in a similar way as in *Javelin*. Work stealing is performed to balance work within worker groups. The authors report on very good parallel efficiency of 91% for a RC5 brute-force key cracking application and a moderate efficiency of 70% for ray-tracing. However, both benchmarks were run in a small setup of eight nodes and thus are not qualified to assess the scalability of the system. The efficiency of the ray-tracing application shows that the lack of support for dynamic decomposition renders *Jalapeno* less suitable for solving ISPs.

The *Personal Power Plant* (P3) [KSS05] adds a thin layer on top of JXTA to simplify access to network-related functionality. On top of this facade different distributed computing libraries are deployed. This includes libraries with support for MPI-style message passing and embarrassingly parallel applications. The platform provides no fault-tolerance mechanisms and is thus not suitable for Desktop Grid Computing. The system achieves a 20-fold speedup in a 32 node cluster on a RC5 brute-force key cracking application. However, this moderate result is due to a task granularity of 1.4 seconds, which has been chosen by at least an order of magnitude to low by the authors. Unfortunately, as in *Jalapeno* and *CONFIIT* task granularity is static. Thus extensive experimentation is required to find a suitable value.

*JNGI* [VNRS02] is a generic distributed computing platform. Scalability is achieved by organizing peers in a hierarchy of arbitrary depth by means of *JXTA*'s peer group abstraction. The assignment of peers to groups can be done according to similarity [BEDSV04] with respect to qualitative and quantitative metrics. This concept called *similarity groups* can be used to transparently group peers according to their network distance in wide-area setups.

1 JXTA is introduced in detail in Chapter 23.

As peers interact intensively only when they are member of the same group, doing so results in a significant rise in efficiency due to increased bandwidth and reduced latency.

*Condor* [LLM88] is a centralized HTC system using COTS components. *Condor* nodes are organized into pools with a manually configured central manager responsible for scheduling jobs using a matchmaking mechanism [RLS00] that finds matching resources for submitted jobs. *Condor* supports job migration and checkpointing [LS99] and thus can handle resource volatility. *Condor* pools can be federated by a *flocking* mechanism [ELvD+96] that allows for cross-pool resource sharing. However, flocking relies on manual configuration.

## 5.2 Superpeer-Based Scheduling

*OurGrid* [BAV+07] is a hierarchical superpeer-based Peer-to-Peer Desktop Grid targeted to bag-of-tasks applications. The architecture consists of three layers: The *working machines* on the lowest layer are responsible for actually executing tasks pushed to them by their single associated superpeer acting as a cluster-head. The superpeer performs *Work Queue with Replication* (WQR) [dSCB03] scheduling. WQR is essentially the same as eager scheduling: It replicates and concurrently executes each task multiple times to provide fault-tolerance and to compensate for heterogeneity. Thus, WQR-based systems do not achieve high parallel efficiency. As task replicas are monitored by superpeers in order to reschedule abnormally terminated tasks, the system could have implemented reactive fault-tolerance as well to reschedule lost tasks on demand. Superpeers connect to an ultrapeer – called the *Core Peer* in *OurGrid* – that provides a registry of available superpeers queried by a superpeer on job scheduling. *OurGrid* provides a XEN-based sandbox called *Sandboxing Without A Name* (SWAN) [CAG+06] for isolated task execution.

*ShareGrid* [ACG10] extends *OurGrid* by providing firewall traversal based on *Virtual Private Networking* (VPN), a portal server for web-based user access, and a storage server for persisting input and output data. *ShareGrid* refrains from using SWAN because of its high execution overhead, its limited number of supported platforms, and the requirement for administrative privileges to be setup. Despite the most simple bag-of-tasks application type and very low rates of abnormal task termination, the reported application level parallel efficiency for a range of applications – although based on the *Equivalent Reference Machines* (ERM) [KTB+04] metric – is comparatively low.

*ad hoc Grid* [TS10] strives to remove the necessity for centralized administration efforts in the *OurGrid* system. This is achieved by using IP multicast communication to dynamically elect and announce *OurGrid* superpeers. While this improves resiliency, the limited adoption of IP multicast may hamper deployment. The impact on computing performance with no failures is 1.5% and thus negligible. Unfortunately, no meaningful results are available for setups with failures.

The Peer-to-Peer Desktop Grid system for bag-of-tasks applications proposed by Mastroianni *et al.* [MCT+09] employs an unstructured superpeer network to match descriptions of and requests for tasks and the data required for their execution. The system relies on flooding to locate jobs and data descriptions matching a given request. Whilst being conceptually simple, the flooding approach is of limited scalability and does not guarantee that a satisfiable request is actually satisfied. To increase locality and fault-tolerance, data is replicated throughout the super-peer network. The simulation-based parallel efficiency for a job consisting of 1000 tasks with an average execution time of 500 seconds and a size of 7.2 MB in a network consisting of 1000 workers, 100 superpeers of which 49 are used as data caches, and a single data source is 6%. This low efficiency is caused by 10-fold task replication the system assumes to be required by applications to enhance statistical accuracy or to minimize the effects caused by faulty or malicious workers.

Butt *et al.* [BZH06] extend *Condor*'s flocking mechanism to become self-organizing using the *Pastry* [RD01b] overlay. The role of the central manager is assigned dynamically to the node whose identifier is numerically closest to a certain identifier in the *Pastry* overlay. By replicating state with its $n$ closest neighbors the central manager becomes $(N-1)$-resilient. For the purpose of flocking, the central managers of all *Condor* pools form another overlay instance on top of which descriptions of idle resources are made available to remote managers. Performance numbers given by the authors substantiate the scalability of their approach. However, no evaluation concerning the attainable efficiency neither in stable nor in volatile setups has been conducted.

## 5.3 Decentralized Scheduling

### 5.3.1 Probabilistic Scheduling

*Paradropper* [DJW+03] strives to provide a novel Peer-to-Peer network substrate for Global Computing. However, the originality of the proposed overlay construction algorithm is limited, as it is very similar to that of the *SCAMP* [GKM01] group membership service. *Paradropper*'s load balancing algorithm can be deployed on any overlay network featuring a higher clustering than a random network of the same size. It is based on the idea of pushing surplus tasks to that neighbor with the lowest load level for a finite number of times. This way a task is eventually transferred to an underloaded node with high probability. The authors do not provide a meaningful experimental analysis as they compare the efficiency of their algorithm to itself on a random network only, but not to other approaches.

Di *et al.* report on a gossip-based load balancing approach for unstructured Peer-to-Peer networks [DWH09] based on the *Newscast* [VJvS03] protocol that maintains a random network by periodically dropping existing connections and connecting to other peers selected at random. Although the system is not yet fully implemented, load balancing is supposed to be done by process migration. The proposed algorithm tries to achieve optimal load balance with minimal migration cost. Detection of over-/underloaded peers is done by comparison to the globally aggregated average load computed by a gossip-based aggregation protocol. Each overloaded peer periodically selects underloaded neighbors to negotiate transferal of load. To avoid *reassignment conflicts*, i.e., several peers migrate processes to the same target peer resulting in an even greater imbalance, an overloaded peer migrates processes to a peer that is underloaded to the same extent with respect to some metric. Thus, the probability of two peers selecting the same target peer is substantially reduced. Simulation results show that the proposed system achieves very good load balance while causing very low average process migration costs that are independent of system size.

*P2P-Tuple* [NH09] deploys a fully distributed *tuple space* [Gel85] based on the *PAST*

[RD01a] Peer-to-Peer distributed storage system to store task-related data of bag-of-tasks applications. The infrastructure is multiplexed by using a *SCRIBE* [CDKR02] multicast group for each application. Fault-tolerance is approached in a probabilistic way using replicas and erasure codes based on Rabin's *Information Dispersal Algorithm* (IDA) [Rab89]. The systems resiliency to node failures can be tuned by selecting a replication factor. Eightfold replication, for example, ensures task survival under churn for up to two days. Hence, responsibility of eventual completion for long running jobs is shifted to the submitting client. The system is evaluated under real-world conditions where it achieves a moderate overall efficiency of 74% for an embarrassingly parallel application with independent tasks in a 19-node cluster setup and 66% in a 72-node *PlanetLab* [CCR⁺03] setup. The comparatively low parallel efficiency is caused by a randomized scheduling strategy that creates an overhead of approximately 17%. While *P2P-Tuple* can be used to execute workflows, it is not suitable for ISPs or other applications that require for dynamic task decomposition.

### 5.3.2 Autonomic Scheduling

The *Organic Grid* (OG) [CBL05] adopts *autonomic scheduling* [KCCF03] to overcome the restriction of state-of-the-art Master/Worker-based Desktop Grid systems to embarrassingly parallel applications. Autonomic scheduling is a decentralized scheduling scheme that only uses information that is locally available to take a scheduling decision. Thus, the scalability limitations of *metaschedulers* used in Grid Computing systems is eliminated, while still supporting non-trivially parallel applications. OG adapts the idea of autonomic scheduling to support highly dynamic systems by constantly reconfiguring its tree overlay network that is used to distribute tasks and collect results. This reconfiguration is done in a way that ensures that those peers with high task throughput are eventually located near the root of the tree. As tasks and results travel along the edges of the tree, this scheme minimizes the average path length between the root and the best performing peers. The tree overlay is also used for termination detection by propagating results towards the root and fault-tolerance by having peers store task replicas for all tasks delivered to any child peer. Hence, OG is in principle suitable for executing diffusing computations with dynamic decomposition. However, for ISPs a situation may arise where most work gets propagated into a leaf of the tree where it is no longer available for load balancing. Although, this situation can be prevented easily by using an alternative performance metric, the time required for performing the resulting tree reconfigurations – the authors report on slow work diffusion that takes 5 minutes for a small network of 18 nodes – may induce long periods of work imbalance and thus would seriously impair the system's overall performance. Although stated by the authors, the approach is *not* fully decentralized as the peer where a new task is injected has to be available until the computation is finished. The authors do not provide any results for volatile setups although the impact of node failures on a regular application where studied elsewhere [CBL07].

### 5.3.3 Agent-based Scheduling

Choi *et al.* propose an adaptive scheduling approach based on *mobile agents* [CBG⁺06]. The originality of their approach is to group peers according to some metrics (like failure rate, peer availability, or peer credibility) and to apply a scheduling technique within each of these groups that best matches the prevailing conditions. While the establishment of scheduling groups is accomplished by a single dedicated master peer, scheduling is performed in a decentralized way by *scheduling agents* that create and distribute *task agents* within their associated group. Fault-tolerance is achieved by heartbeating for scheduling agents and by replication for task agents. According to a performance evaluation on a testbed of 200 nodes, their approach is superior to centralized eager scheduling and yields increases of 3% to 20% in task throughput for a non-volatile setup and of 14% to 120% for a volatile setup with session times between 5 and 50 minutes.

*PPVC* [ZYX09] is a Peer-to-Peer Desktop Grid system with agent-based dynamic decomposition and thus is in principal suitable for ISP execution. However, crucial aspects of the system including details on the type and structure of the underlying Peer-to-Peer network and the employed termination detection algorithm have not been published. Furthermore, the simplistic fault-tolerance mechanism proposed by the authors induces significant inefficiencies when an intermediary task in the spawn tree gets lost as the whole subtree of descendant tasks is restored. The parallel efficiency for the N-Queen problem on a testbed consisting of three nodes is – despite peer failures are not considered – well below 90%.

*Messor* [MMB03] is a biologically inspired Peer-to-Peer Grid Computing system based on mobile agents. The natural model of *Messor* is a particular ant species called *Messor Sancta* whose colonies are able to pile objects into clusters. This *emergent behavior* results from very simple behavioral patterns of individual ants. While building on the same principles, *Messor* agents balance load across the nodes of the system by randomly migrating through the system searching for overloaded nodes. Once found, the agent again performs a random walk trying to locate an underloaded node. On success, the agent drops contact information and starts over. The authors present preliminary test results showing fast dispersion of load. However, no comparison to other load balancing techniques has been conducted.

### 5.3.4 Overlay-Based Scheduling

*Cluster Computing on the Fly* (CCOF) [LZZ+05] aims at creating a scheduling architecture for Desktop Grids tailored to different kinds of applications. In particular, CCOF announced support for tree-based search problems – however, contributions in this field are still pending. A unique feature of CCOF is its *timezone-aware scheduler* [ZL06]. The scheduler exploits the fact that Desktop Grids often span hosts located in different time zones and that hosts are idle at night with high probability. By proactively migrating tasks such that they are always located at a host in a time zone where it is night, CCOF outperforms classic timezone-oblivious scheduling approaches as tasks are more likely to be run to completion without being interrupted. The scheduler is realized using a self-organizing overlay network based on the *Content Addressable Network* (CAN) [RFH+01] in which hosts organize themselves in a way such that those with small time differences are located nearby in the overlay network. However, the benefit of the approach compared to decentralized systems employing load balancing and checkpointing techniques has to be doubted as idle cycles during daytime can be scavenged as well by these systems and progress made on a task is not (completely) lost when a host departs.

*G2:P2P* [MK05] is an extension to a previously centrally organized Desktop Grid Computing system called *G2:Classic* [KRS02]. *G2:P2P* implements a distributed object model, however with the restriction that each remote method invocation may execute for a short period of time only. This restriction is due to the system's weak object migration model that requires an object to become idle before migration may be initiated. To support volatile systems in which objects have to be migrated away from departing hosts, object addressing is realized in a location independent way by using the identifiers of a DHT-based network overlay based on *Pastry* [RD01b]. Load balancing is achieved using the well-known DHT approach of mapping from object identifiers to host identifiers by applying a hash function. To cope with suboptimal identifier distributions, the authors propose an optimization to level local imbalances by allowing for placing objects in the direct neighborhood of their target host. However, this introduces additional overhead to the already expensive multi-hop message routing scheme of DHTs as method invocation messages routed to a host potentially have to be forwarded to the real location of the target object which adds another hop to the routing path. Fault-tolerance in *G2:P2P* is transparent, fully automated, and adjustable to satisfy the requirements imposed by the underlying system. However, only one of the three available schemes assumes a realistic failure model, where hosts crash or leave without notice and (may) never return. The scheme is based on a combination of checkpointing and message logging but suffers from high complexity induced by the distributed object model. Whether it is still able to deliver high efficiency is unclear as no evaluation is given.

Fischer *et al.* propose a scheduling middleware [FFM06] that provides different methods to distribute partial task lists for bag-of-tasks applications within a Peer-to-Peer Desktop Grid. A job is decomposed on submission and the resulting tasks are scheduled for execution right afterwards. The proposed methods are gossiping, *Chord*-based broadcast, and a combination of both. Fault-tolerance is achieved using timeout-based task restarts performed on the initiating peer that is thus required to be stable until the computation is done. The authors present results of an experimental evaluation using a synthetic benchmark with uniform subtasks on a testbed of 160 non-volatile middleware instances running on a 16-node cluster with a Gigabit Ethernet interconnect. The results indicate that the hybrid approach is superior to the pure approaches. The achieved parallel efficiencies however are low, ranging from $\approx 37\%$ for 64 subtasks to $\approx 76\%$ for 256 substasks. These moderate results can be attributed to the system's inability to dynamically decompose tasks: For the setup with 64 subtasks not all nodes are taking part in the computation. For the setup with 256 subtasks any schedule that assigns more than one task to a node puts the execution on that node on the critical path. In both cases many nodes are idle producing substantial losses in efficiency.

Kim *et al.* propose a Peer-to-Peer Desktop Grid system that employs DHT overlays to solve the problem of finding a peer within the system that satisfies the requirements to solve a given task. This scheduling problem is called the *matchmaking problem*. The authors propose two different solving strategies: The first employs tree-based hierarchical aggregation of peer information to quickly determine whether a matching peer is contained within the subtree routed at the given peer [KNM+07]. If this is the case the request is routed to the respective children. Otherwise, the parent peer is visited. This way a matching peer can be found in $O(logN)$ steps in a network consisting of $N$ peers if such a peer exists. The second approach puts peers into a *Content Addressable Network* (CAN) [RFH+01] at coordinates derived from the capabilities of the peer [KKM+07]. A task can be routed to a matching peer by simply mapping the task's requirements to a location within the overlay.

# 6 Miscellaneous

## 6.1 Meta- and Integration Approaches

*BonjourGrid* [ACJ09] is a meta-approach to autonomously orchestrate multiple instances of existing Institutional Desktop Grid Computing middleware systems based on the Master/-Worker execution model. *BonjourGrid* provides multi-application support with per application fault containment. The system implements a fully decentralized resource discovery service based on the *Bonjour* protocol[1]. The scalability and performance of the discovery subsystem has been substantiated in a large-scale setup of 300 nodes [AD09]. While fault-tolerance for workers is handled by the respective Desktop Grid Computing middleware, master nodes are made resilient using passive replication and virtualization [ACJS10]. One limitation of *BonjourGrid* is its lack of support for wide area setups.

*Enabling Desktop Grids for e-Science* (EDGeS) [FHL+08] is a European effort to make Desktop and Service Grids interoperable by allowing to seamlessly process jobs from Service Grids in Desktop Grids and vice versa. Technically this is done using bridging technology [FKBG10] that uses either Service Grid resources as Desktop Grid nodes or the other way around. As of now, the proposed approach is only suitable for embarrassingly parallel applications as communication across the boundaries of a Service Grid is problematic and the schedulers of the Grids do not cooperate. Which additional application classes can be supported is yet an open research question.

## 6.2 Market-Based Approaches

Volunteer Computing has proven to be able to attract a large number of enthusiasts. However, to tap the full potential of Desktop Grid Computing the set of contributors must be extended beyond enthusiasts. One approach to attain this goal is to have clients pay for the resources they consume. This idea led to the emergence of market-based systems like *Popcorn* [RN00], *JaWS* [LK00], the *Compute Power Market* (CPM) [BV01], and the *Computational Exchange* (CX) [CM02]. Common to all these systems is their approach to trade resources between providers and consumers by means of auction systems and brokers.

1 *Bonjour* is Apples implementation of the *ZeroConf* protocol [SC05].

# Part III

# Modularization

Desktop Grid Computing platforms, as well as Enterprise Mashups and Cloud Computing infrastructures, are representatives of a new breed of systems that leverage the modularity paradigm to assemble large-scale dynamic applications from modules contributed by different, possibly untrustworthy providers. Faulty and malicious modules, incompatibility issues, and lack of per-module resource accounting are major challenges for assembling and operating such systems. In this part, we describe how these problems are solved by retrofitting module management systems with the ability to deploy modules to execution environments with adjustable degree of isolation. We give a formal definition of the underlying hierarchical *Module Isolation Problem* and devise an online algorithm to solve it in an incremental fashion. We discuss how to apply our approach to the state-of-the-art OSGi module management system that is the nucleus of COHESION and demonstrate its effectiveness by an experimental evaluation.

## RELATED PUBLICATIONS

[SB11] SCHULZ, Sven and BLOCHINGER, Wolfgang: **Adjustable Module Isolation for Distributed Computing Infrastructures**, In: *The 12th IEEE/ACM International Conference on Grid Computing (Grid 2011), In press.*

Modularity is the paradigm of building complex systems from smaller loosely-coupled subsystems that can be designed and implemented independently, yet function together as a whole. As it fosters flexibility, adaptability, and reusability, modularity has become a cornerstone in software engineering and represents an important measure to tackle the ever increasing complexity of today's IT systems. Since the groundbreaking work of Parnas [Par72] in the early 1970s, support for modularity has constantly evolved from build-time approaches offered by programming languages, like Modula, to sophisticated *Module Management Systems* that allow for changing the composition and configuration of running systems on the fly. A major advantage of such systems is a significantly increased availability as downtime caused by maintenance outages can be minimized.

Today, module management systems are often employed in multi-authority environments, where modules come from different, possibly untrusted sources and are used concurrently by multiple independent parties. Apart from Desktop Grid systems, Enterprise Mashups [mas] and Cloud Computing infrastructures [RDA09, APG+10] are examples for this new kind of systems. Ensuring essential system properties like system security, process safety, resource accountability, and consistency of configuration is very challenging for these computing environments and has not been solved satisfactorily yet.

We propose to tackle these challenges by providing customized isolation environments acting both as a security sandbox and as an ad-hoc, platform-independent runtime environment for modules. The distinguishing features of our language and framework agnostic approach are fully automated operation, tunable degree of isolation between modules, efficient use of host system resources, sufficient scalability to support a large number of modules, flexible specification of isolation constraints, and support for multiple authorities that can specify such constraints. Further contributions are a formal definition of the *(Minimum) Module Isolation Problem* and the application of our approach to the OSGi module management system, which is the de facto standard for modular applications in the Java universe.

The rest of this part is organized as follows: In Chapter 7, we substantiate the idea of an *Isolating Module Management System* by giving an informal definition, identifying key requirements, and demonstrating the relevance of our work by discussing typical application scenarios. In Chapter 8, we give a formal definition of the *Module Isolation Problem*, show how the degree of isolation between individual modules can be made adjustable, and present an online algorithm that can be efficiently employed in highly dynamic environments. Chapter 9 describes the application of our approach to OSGi and discusses interesting aspects of our prototypical implementation called I-OSGI, in particular a performance evaluation. Related work is discussed in Chapter 10.

# 7 Isolating Module Management System

## 7.1 Definition and Requirements

A *Module Management System* (MMS) consists of a set of modules and functionality to add and remove modules from this set at runtime. An *Isolating Module Management System* (IMMS) is an MMS that maintains a dynamic set of isolation environments to which modules are deployed such that a set of isolation constraints is satisfied. The most simple IMMS would create a dedicated isolation environment for each module. As discussed in Chapter 10, existing solutions work this way. Our approach is more sophisticated as it is designed to satisfy the following six requirements:

**R1** Isolating modules entails overhead. In particular, inter-module interaction becomes more expensive when modules no longer share the same address space. Thus, the *degree of isolation between modules must be adjustable to allow for balancing out overhead and isolation requirements*.

**R2** Isolation environments consume host system resources, e.g., main memory and file handles. Moreover, they must be configured, launched, and – when no longer required – destroyed. These management operations incur additional costs. Thus, *the number of isolation environments should be minimized by sharing them whenever possible*.

**R3** Large modular applications can consist of hundreds of modules. This complexity renders manual configuration of an isolation system infeasible and requires a *fully automated solution to configure the isolation environment*. Configuration must be done in an efficient and scalable way to *support a sufficiently large number of modules*.

**R4** The mechanism to define isolation constraints must be flexible and expressive enough to *support a broad range of constraint types*.

**R5** MMS-based approaches are typically used in multi-authority environments. Thus, an IMMS has to *allow for multiple parties to contribute constraints*.

**R6** The fact that a module runs within an isolated environment *must be transparent to both the module itself and the remainder of the system*. This non-intrusiveness ensures that module development is independent of the details of later isolation.

## 7.2 Application Scenarios

To substantiate the usefulness of our IMMS approach, we discuss a number of scenarios where the ability to isolate modules is particularly advantageous.

### 7.2.1 Security/Safety

A common technique for ensuring security and safety in modular systems is trust. However, relying on trust-based mechanisms is not an ideal solution: First, there are bugs in each and every software system. Although the issuer of a module is trusted, a yet unknown bug may seriously impair the host system. Second, there may be a need for using a module despite its source being untrustworthy, because for example there are no alternative implementations of the required functionality available and a reimplementation is too expensive. On the other hand, permission-based systems are often fine-grained, rendering the task of specifying what a specific module may and may not do very time-consuming and error-prone. Moreover, the resource provider may be no IT expert and simply is not able to decide whether a given set of permissions poses a threat to his system or not. Even if the execution environment provides additional security mechanisms, as it is for example the case for the *Java Virtual Machine* (JVM), the protection may be incomplete. An example is native code attached over the *Java Native Interface* (JNI) which runs outside the security sandbox, exposing the system to serious threats like buffer overflow attacks or format string attacks.

In all discussed cases an IMMS can be used to put the potentially dangerous module under quarantine. Auditing security related actions within an isolated environment with limited or controlled access to the host system can then be used to decide whether the module should be released from quarantine after some time. In case of an incident the malicious module can be terminated and removed without harming other parts of the system. This approach allows using the module while keeping the risk of being damaged low.

### 7.2.2 Compatibility

Today's IT world is highly heterogeneous. Integrating components written in different languages and/or for different platforms has become an inevitable yet expensive aspect of software development. Java modules incorporating native code for example are platform dependent and thus have to be explicitly ported to all target systems. Using our IMMS such modules can be run in a customized environment with a compatible setup causing no additional development effort. Another source of incompatibilities are version conflicts in the dependency sets of modules. Although some MMS allow for deploying several versions of a module concurrently, the resulting duplication wastes resources. Even worse modules may publish or consume singleton resources rendering parallel deployment impossible.

### 7.2.3 Resource Accounting

Accounting for the consumption of resources, such as CPU time, memory, or persistent storage is a vital prerequisite for many real-world applications in particular in the field of Cloud Computing. It is necessary for billing as well as for preventing malicious or accidental resource overuse, such as denial-of-service attacks. However, predominant execution platforms, like the Java or Python, lack support for mature resource accounting mechanisms. Existing approaches either require a modified JVM [CDT03] or contribute considerably to the overall execution time [HB08]. With module isolation, lightweight accounting facilities of modern operating systems can be leveraged by running specific modules in a separate OS instance.

# 8 Adjustable Module Isolation

In this chapter, we present an algorithmic framework to incrementally compute the mapping of modules to isolation environments. At its core lies the basic *Module Isolation Problem* (MIP), for which we first give a formal definition. Subsequently, we describe, how we can achieve a variable degree of isolation between modules (R1,R2) employing a hierarchical approach which is based on solving the corresponding minimum MIP on multiple layers. Finally, we derive an online algorithm that is able to compute solutions incrementally every time the system composition is changed by adding or removing a module. We also show that the basic MIP is – and hence all derived problems are – NP-complete. In Chapter 9, we deal with selecting suitable approximations and heuristics for implementing our algorithmic framework that allow us to handle several hundred modules (R3).

## 8.1 The Module Isolation Problem

Let $\mathcal{S}$ be a *multi-authority modular system* (MAMS) defined by the triple $(\mathcal{M}, \mathcal{A}, \perp)$ with:

1. A set of modules $\mathcal{M} = \{m_1, \ldots, m_n\}$.
2. A set of authorities $\mathcal{A} = \left\{a_1, \ldots, a_p\right\}$, like the infrastructure or module provider.
3. A set $\perp$ defining isolation constraints. It contains a binary relation $\perp_i \subseteq \mathcal{M}^2$ for each authority $a_i \in \mathcal{A}$. $(m_k, m_l) \in \perp_i$ indicates that according to authority $a_i$ modules $m_k$ and $m_l$ have to be deployed to different isolation environments.

Let $\mathcal{I} = \{i_1, \ldots, i_m\}$ be a set of isolation environments to each of which one or more modules can be deployed.

**Definition 8.1.1.** ***Module Isolation Problem (k-MIP)*** *Given an MAMS $\mathcal{S} = (\mathcal{M}, \mathcal{A}, \perp)$,* k-MIP *is the decision problem whether there is a set $\mathcal{I}$ of isolation environments with $|\mathcal{I}| \leq k$ and a mapping $f : \mathcal{M} \mapsto \mathcal{I}$ such that all relations in $\perp$ hold.*

To prove that k-MIP is NP-complete, we show that k-MIP is in NP, i.e., its solutions can be verified in polynomial time, and that it is NP-hard, i.e., it is at least as hard as the hardest problems in NP.

**Lemma 8.1.1.** *k-MIP is in NP.*

*Proof.* Consider the following procedure for verifying solutions:

```
1: procedure isValid-k-MIP($\mathcal{S}, f$)
2:   for all $\perp' \in \perp$ do
```

```
3:   for all $(m,m') \in \perp'$ do
4:     if $f(m) = f(m')$ then return $false$
5:   return $true$
```

Obviously, the procedure isValid-k-MIP exhibits polynomial run-time. ■

**Lemma 8.1.2.** *k-MIP is NP-hard.*

*Proof.* We show that k-coloring reduces to k-MIP. Let $G = (V,E)$ be an undirected graph with vertices $V$ and edges $E$. *K-coloring* is the decision problem whether colors can be assigned to the vertices $V$ such that only $k$ different colors are used and no adjacent vertices have the same color. The straightforward polynomial time mapping between instances of k-MIP and k-coloring is as follows

1. Each vertex $v_i \in V$ becomes a module $m_i \in \mathcal{M}$ and vice versa
2. Each edge $(v_i, v_j) \in E$ becomes a tuple $(m_i, m_j) \in \perp$ and vice versa

We now show that $G$ is k-colorable, iff a solution to the corresponding k-MIP instance for $\mathcal{S} = (\mathcal{M}, \{a\}, \perp)$ and some authority $a$ exists:

„⇒“: Let $c : V \mapsto [1,k]$ be a k-coloring of $G$ and $[i] = \{v \in V | c(v) = i\}$ denote all vertices with color $i$. By construction, there are no edges between the vertices in $[i]$. Hence, a solution to k-MIP is $\mathcal{I} = \{i_1, \ldots, i_k\}$ and $f$ maps $m_j$ to that $i_l$ for which $v_j \in [l]$ holds.

„⇐“: Let $\mathcal{I} = \{i_1, \ldots, i_k\}$ and $f$ be a solution to the k-MIP instance. With $[j] = \{m \in \mathcal{M} | f(m) = i_j\}$ being the modules mapped to the isolation environment $i_j$ by $f$. Note, that by construction $(m_i, m_j)$ is not in $\perp$, if $m_i$ and $m_j$ are both in the same isolation environment. Hence, if we assign a unique color $l$ to all modules in a given isolation environment $i_l$, there are no edges between the corresponding vertices of the k-coloring instance. ■

**Theorem 8.1.1.** *k-MIP is NP-complete.*

*Proof.* Follows directly from Lemmas 8.1.1 and 8.1.2 ■

Keeping the resource usage and thus the number of isolation environments as small as possible is one of our design goals. Hence, we are interested in the minimum number of isolation environments necessary for a given MAMS. The resulting optimization problem is the

**Definition 8.1.2.** ***Minimum Module Isolation Problem (Minimum-MIP)*** *Given an MAMS $\mathcal{S} = (\mathcal{M}, \mathcal{A}, \perp)$,* Minimum-MIP *is the optimization problem of finding a minimum set $\mathcal{I}$ of isolation environments such that a solution to k-MIP exists for $\mathcal{S}$.*

**Theorem 8.1.2.** *Minimum-MIP is an NP-optimization problem.*

*Proof.* Follows directly from the fact that k-MIP is the corresponding canonical decision problem for Minimum-MIP. ■

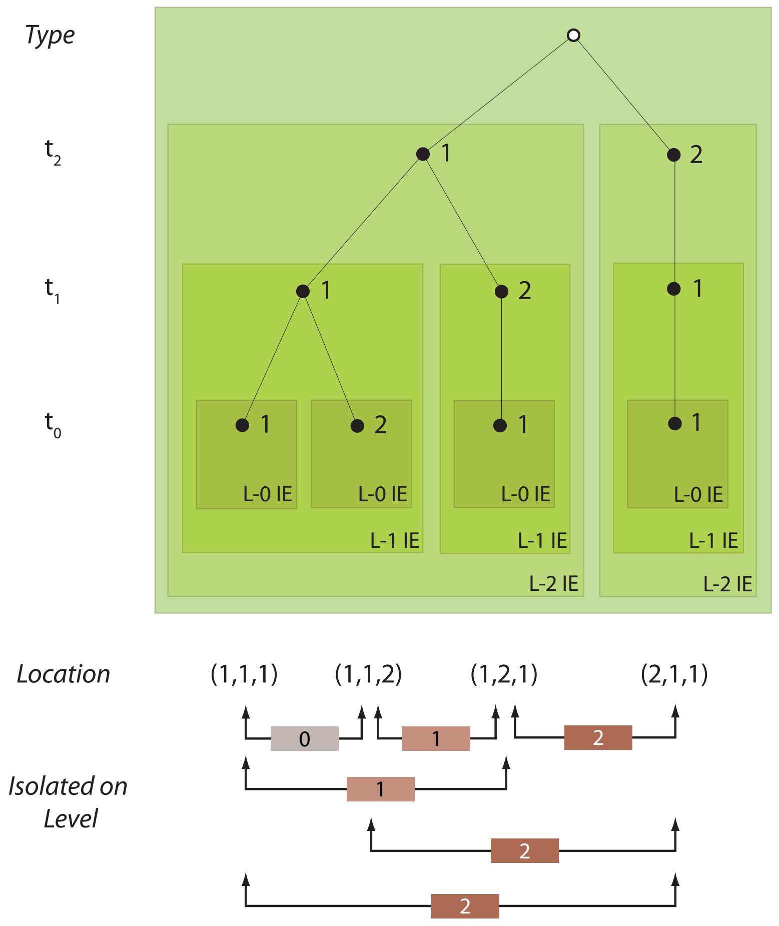


**Figure 8.1:** Nested isolation environments (L-*i* IE denotes a level-*i* isolation environment)

## 8.2 Adjustability

Let $\mathcal{T} = \{t_0, \ldots, t_n\}$ be a *type hierarchy*, which is a totally ordered set of *isolation environment types* abbreviated as *types* in the following. The order is defined by a containment relation, i.e., for two types $t_i, t_j \in \mathcal{T}$, $t_i < t_j$ holds if $t_j$ can contain *instances* of $t_i$ (see Figure 8.1). An instance of a type $t_i$ is called a *level-i* isolation environment. Modules are deployed to level-0 instances. Child environments of an isolation environment are indexed consecutively. $\mathcal{I}_k \subseteq \mathcal{I}$ denotes the set of all level-$k$ isolation environments.

The *location* of a module $m$ is defined as $L_m = \left(l_{|\mathcal{T}|-1}, \ldots, l_0\right)$, where $l_i$ represents the index of a specific isolation environment instance on level $i$. Two modules are said to be isolated on *level* $|\mathcal{T}| - i - 1$ when their locations share a common prefix of length $i$. For example, in Figure 8.1 modules deployed to locations $(1,1,1)$ and $(1,2,1)$ are isolated on level 1.

To incorporate type hierarchies, we extend our previous definition of $\perp$, which now denotes a set that consists of a binary relation $\perp_i^j$ for each tuple $\left(t_i, a_j\right) \in \mathcal{T} \times \mathcal{A}$. A tuple $(m_k, m_l) \in \perp_i^j$ indicates that according to authority $a_j$ two modules $m_k, m_l \in \mathcal{M}$ have to be deployed to different instances of $t_i$, i.e., should be isolated on level $i$.

With these definitions, we can specify the

**Definition 8.2.1.** ***Hierarchical Module Isolation Problem*** *(****Hierarchical-k-MIP****) Given an MAMS $\mathcal{S} = (\mathcal{M}, \mathcal{A}, \perp)$ and a type hierarchy $\mathcal{T} = \{t_0, \ldots, t_n\}$, the Hierarchical-k-MIP is the decision problem whether there is a set $\mathcal{I}$ of isolation environments with $\left|\mathcal{I}_j\right| \leq k$, $\forall j \in \{1, \ldots, |\mathcal{T}|\}$ and a mapping $f^* : \mathcal{M} \mapsto \mathbb{N}^{|\mathcal{T}|}$ from modules to locations such that all relations in $\perp$ hold.*

We first show that

**Lemma 8.2.1.** *Hierarchical-k-MIP is in NP.*

*Proof.* A solution to Hierarchical-k-MIP can be verified with the following procedure:

```
1: procedure isValid-H-k-MIP($\mathcal{S}$,$\mathcal{T}$,$f^*$)
2:   for all $i \in \{1, \ldots, |\mathcal{T}|\}$ do
3:     $\mathcal{S}' \leftarrow \left(\mathcal{M}, \mathcal{A}, \cup_j \perp_i^j\right)$
4:     if $\neg$ isValid-k-MIP($\mathcal{S}'$,$f^*$) then return $false$
5:   return $true$
```

With Lemma 8.1.1 it is obvious that isValid-H-k-MIP is a polynomial time procedure. Hence, Hierarchical-k-MIP is in NP. ■

**Theorem 8.2.1.** *Hierarchical-k-MIP is NP-complete.*

*Proof.* Follows immediately from Lemma 8.2.1 and the fact that k-MIP is NP-hard (Lemma 8.1.2) and a specialization of Hierarchical-k-MIP (for $|\mathcal{T}| = 1$). ■

As in the case of k-MIP and Minimum-MIP we can define the corresponding optimization problem, namely the

**Definition 8.2.2.** ***Minimum Hierarchical Module Isolation Problem*** *(****Minimum-Hierarchical-MIP****) Given an MAMS $\mathcal{S} = (\mathcal{M}, \mathcal{A}, \perp)$ and a type hierarchy $\mathcal{T} = \{t_0, \ldots, t_n\}$, the Minimum-Hierarchical-MIP is the optimization problem of finding a minimum set $\mathcal{I}$ of isolation environments such that a solution to Hierarchical-k-MIP exists for $\mathcal{S}$ and $\mathcal{T}$.*

**Theorem 8.2.2.** *Minimum-Hierarchical-MIP is an NP-optimization problem.*

*Proof.* Follows directly from the fact that Hierarchical-k-MIP is the corresponding canonical decision problem for Minimum-Hierarchical-MIP. ■

Let $\textsc{Min-MIP}(\mathcal{S})$ be a (approximation) procedure that solves the Minimum-MIP for a given MAMS $\mathcal{S} = (\mathcal{M}, \mathcal{A}, \perp)$ and let $\circ$ be the *append* operator on tuples

$$\begin{aligned} \circ : & I^n \times I^m \to I^{n+m}, \\ & (i_1, \ldots, i_n) \circ (j_1, \ldots, j_m) \mapsto (i_1, \ldots, i_n, j_1, \ldots, j_m) . \end{aligned} \tag{8.1}$$

Then a call to the following recursive procedure with initially $k = |\mathcal{T}|$ solves the Minimum-Hierarchical-MIP:

```
1: procedure $\textsc{Min-H-MIP}(\mathcal{S},\mathcal{T},k)$
2:   $\bot \leftarrow \bigcup_j \bot_k^j$
3:   $(\mathcal{I},f) \leftarrow \textsc{Min-MIP}(\mathcal{M},\mathcal{A},\bot)$
4:   for all $m \in \mathcal{M}$ do
5:     $L_m \leftarrow L_m \circ f(m)$
6:   if $k > 0$ then
7:     for all $i \in \mathcal{I}$ do
8:       $\textsc{Min-H-MIP}(([i]_f,\mathcal{A},\bot),\mathcal{T},k-1)$
```

In line 2 an aggregated set of isolation constraints for level-$k$ isolation environments is computed, which is fed into the $\textsc{Min-MIP}$ procedure to compute a solution to Minimum-MIP (line 3). Note that Minimum-MIP always has a solution, as it corresponds to the vertex coloring problem using a minimum number of colors, which is always possible as any graph $G$ can be trivially $|V|$-colored where $V$ is the node set of $G$. In lines 4 and 5 the locations of the modules are constructed incrementally according to this solution. Finally, if level 0 hasn't been reached yet, the procedure is called recursively to compute the child isolation environments for each set of modules located in the same isolation environment on the current level (lines 6–8).

## 8.3 Online Processing

$\textsc{Min-H-MIP}$ is an offline algorithm, i.e., the whole input has to be available from the beginning. However, in an IMMS modules are added and removed at runtime. Thus, an online algorithm is required.

Let $\sigma$ be the *suffix* operator on tuples

$$\begin{aligned} &\sigma : I^n \times \{0,\ldots,n\} \to I^m, \\ &\quad \sigma((i_1,\ldots,i_n),k) \mapsto (i_{n-(k+1)},\ldots,i_n). \end{aligned} \tag{8.2}$$

and let $L'$ be the set of module locations from a previous invocation (defined by $f'$), then a call to the following procedure with initially $k = |\mathcal{T}|$ computes an incremental solution to $\mathcal{S} = (\mathcal{M},\mathcal{A},\bot)$ with $\mathcal{M}$ being either $\mathcal{M}' \cup \{m\}$ (module $m$ added) or $\mathcal{M}' \setminus \{m\}$ (module $m$ removed):

```
 1: procedure $\textsc{Inc-Min-H-MIP}(\mathcal{S},\mathcal{T},k,L')$
 2:   $\bot \leftarrow \bigcup_j \bot_k^j$
 3:   $(\mathcal{I},f) \leftarrow \textsc{Match}(\textsc{Min-MIP}((\mathcal{M},\mathcal{A},\bot),L'))$
 4:   for all $m \in \mathcal{M}$ do
 5:     $L_m \leftarrow L_m \circ f(m)$
 6:   if $k > 0$ then
 7:     for all $i \in \mathcal{I}$ do
 8:       if $[i]_f \neq [i]_{f'}$ then
 9:         $\textsc{Inc-Min-H-MIP}(([i]_f,\mathcal{A},\bot),\mathcal{T},k-1,L')$
10:       else
11:         for all $m \in [i]_f$ do
```

```
12:        $L_m \leftarrow L_m \circ \sigma(L_m,k)$
```

Basically, the algorithm is a variation of $\textsc{Min-H-MIP}$ that only descends into isolation environments that have been subject to a change. Line 2 and lines 4 and 5 are the same as in $\textsc{Min-H-MIP}$. In line 3 $\textsc{Min-MIP}$ is invoked as above but the result is further optimized by the procedure $\textsc{Match}$ (to be discussed below) taking into account the previous module locations $L'$. Line 8 checks whether any module has been added or removed from isolation environment $i$ ($[i]_{f'}$ can be computed trivially from $L'$). If this is the case, the algorithm descends into that isolation environment in line 9. Otherwise, the locations of all modules in $i$ are completed by the respective parts of the previous locations (lines 11 and 12).

$\textsc{Match}$ computes a least-cost matching between $(\mathcal{I},f)$ returned by $\textsc{Min-MIP}$ and $(\mathcal{I}',f')$ implicitly defined by $L'$. This is necessary as the solution to the Minimum-MIP for the new configuration of the MAMS may be considerably different from the solution for the original MAMS. The underlying problem is called the *linear assignment problem*. With $A = \left\{[i]_f \,|\, i \in \mathcal{I}\right\}$ and $B = \left\{[i]_{f'} \,|\, i \in \mathcal{I}'\right\}$, we have to find a bijection[1] $g : A \rightarrow B$ such that the *cost function*

$$\sum_{i \in \mathcal{I}} \delta\left([i]_f, g\left([i]_f\right)\right) \tag{8.3}$$

is minimized. The *weight function* $\delta : A \times B \rightarrow \mathbb{N}$ is the edit distance between two unordered sets with respect to element addition and removal. Thus, the (per level) overall number of module migrations between isolation environments is minimized by $\textsc{Match}$.

1 If $|I| \neq |I'|$ empty dummy environments have to be added to the smaller set.

# 9 i-OSGi – An IMMS on top of OSGi

This chapter describes how the IMMS concept can be realized on top of an industrial strength MMS from the Java universe – OSGi. i-OSGi's architecture is especially designed to satisfy the requirements identified in Chapter 7 and is used as the isolation system in the extensible core of Cohesion. The algorithmic framework for module isolation presented in the last chapter is translated into a fine-grained isolation environment type hierarchy and a versatile mechanism for both manually providing and automatically deducing isolation constraints. The performance of our i-OSGi implementation, which employs two different approaches to solve the underlying Minimum-MIP, is substantiated by an experimental evaluation.

## 9.1 Prerequisites

### 9.1.1 OSGi

OSGi [OSG] – an acronym for *Open Services Gateway interface* – is a module management system for Java. The OSGi specification is maintained by the *OSGi Alliance*, a growing community of leading IT companies. OSGi has matured over a decade since its inception in 2000 and is awaiting the 4.3 release at the time of this writing. There are a whole host of open-source and commercial OSGi implementations that are used in many large projects including industrial strength IDEs and JEE application servers.

The *OSGi framework* exhibits a layered architecture consisting of the following four layers:

- The *Execution Environment* comprises the set of classes and methods available on a specific platform.
- The *Module and Lifecycle Layer* provides a dynamic module system which allows for hot-deployment, hot-undeployment, and inter-dependency management of modules, called *bundles* in OSGi jargon. OSGi has built-in support for native code extensions by providing the ability to automatically load platform-dependent shared libraries. However, as with standard Java, native code attached over the *Java Native Interface* (JNI) [Lia99] is executed outside the security sandbox.
- The *Service Layer* implements a system-wide registry exposing a publish-find-bind model for services. An OSGi *service* is a Java interface that is decorated with a set of service properties encoded as key/value-pairs on publication. Service consumers can query the registry using LDAP-style (RFC 1960) filter predicates and bind to returned *service references*. The indirection over service references allows for handling

the inherent volatility[1] induced by the dynamism of the module system, i.e., a service disappears as soon as the providing bundle is stopped.

- The *Security Layer* extends the Java security model with module- and service-specific functionality.

### 9.1.2 Distributed OSGi

The OSGi service model was scoped to purely local OSGi frameworks prior to version 4.2. Stimulated by a growing number of enterprise use cases, OSGi's service model has been extended to distributed scenarios by allowing provisioning and consumption of services across framework borders. This extension called *Distributed OSGi* is based on two new entities: the *distribution provider* and the *discovery service*. The former is responsible for handling the communication between service providers and consumers by creating protocol-specific service endpoints. The discovery service is used to publish and retrieve the service descriptions together with their service endpoints. When a remote service is imported to the local framework, the distribution provider crafts and deploys a service proxy that can be used by the consumer like a local service object.

### 9.1.3 Isolation Techniques

In the context of OSGi the following isolation techniques are of particular relevance (see Figure 9.1):

**Child Frameworks.** OSGi RFC 138 [osg10] specifies a mechanism to create *Child Frameworks* of an OSGi framework. Child frameworks provide the lowest degree of isolation including class-space and service isolation.

**Isolates.** The Java *Application Isolation API* (JSR-121) [Pal06] specification introduces *Isolates* as a mechanism to run several Java applications within the same JVM in isolation. Isolates provide object-space isolation (as long as no object references are exchanged across isolate borders) while at the same time preserving the performance of direct method invocation. Each isolate has for example its own system properties, classpath, security manager, shutdown hooks, and garbage collector. Additionally, an experimental resource accounting framework has been proposed [CHS+03].

**Processes.** All modern operating systems support the concept of processes. Process-based isolation ensures fault containment and allows for leveraging all per-process resource accounting and control facilities provided by the operating system.

1 Note that the term *volatility* refers to the unpredictable coming and going of OSGi services in this context and is not related to the host volatility in Desktop Grids.

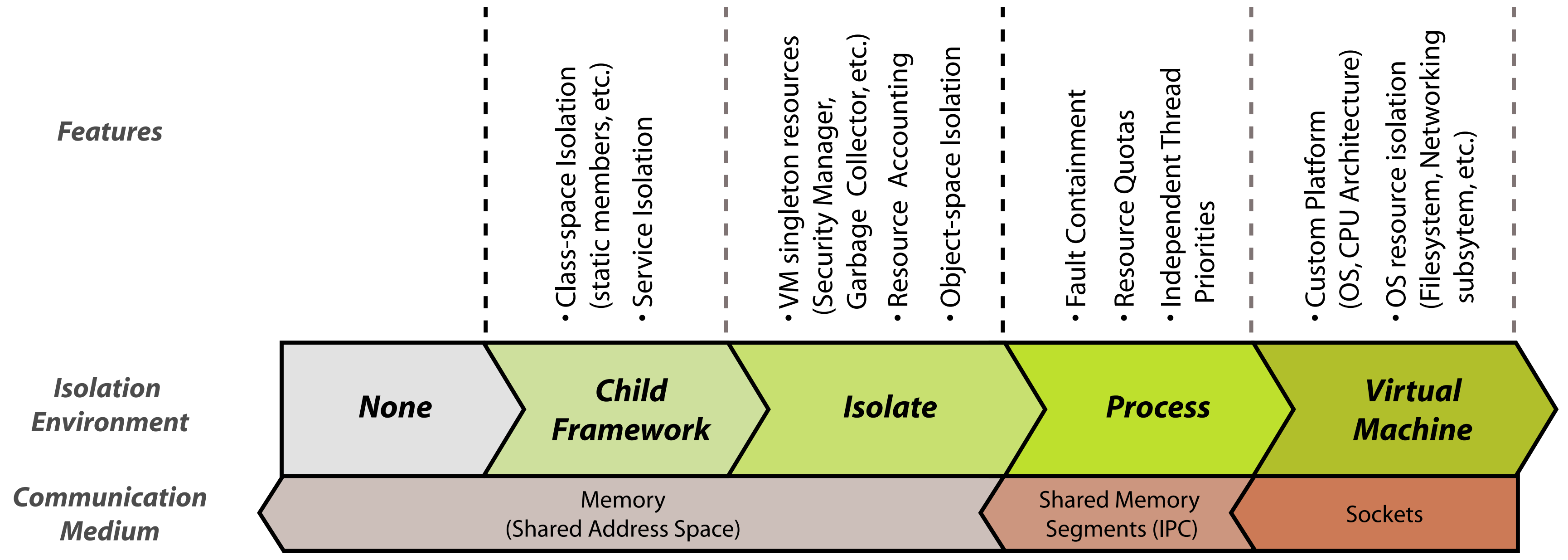

Features
Isolation Environment
Communication Medium
• Class-space Isolation (static members, etc.)
• Service Isolation
• VM singleton resources (Security Manager, Garbage Collector, etc.)
• Resource Accounting
• Object-space Isolation
• Fault Containment
• Resource Quotas
• Independent Thread Priorities
• Custom Platform (OS, CPU Architecture)
• OS resource isolation (Filesystem, Networking subsytem, etc.)
None
Child Framework
Isolate
Process
Virtual Machine
Memory (Shared Address Space)
Shared Memory Segments (IPC)
Sockets

**Figure 9.1:** Overview of relevant isolation techniques (For each technique the isolation properties/additional features and the most efficient communication mechanism are given.)

**Virtual Machines.** Virtual machines provide the highest degree of isolation supported by i-OSGi. The isolation covers resources managed by the operating system including the filesystem and the networking subsystem. Bundles requiring a given OS/Architecture combination can be deployed by creating a suitable virtual machine.

## 9.2 Architecture

Figure 9.2 depicts the component-based architecture of i-OSGi. The fundament of i-OSGi are hierarchical *Isolation Environments* as introduced in Chapter 8. The architecture is specifically designed to support a broad range of isolation strategies, in particular the techniques discussed in Section 9.1.3. Together with an efficient implementation of our algorithmic framework presented in Chapter 8, this flexibility ensures that the degree of isolation can be fine-tuned to the requirements of many conceivable scenarios (R1[1]).

Bundles are installed to *Isolated Frameworks*. An isolated framework is an OSGi framework which runs exclusively in its own level-0 isolation environment. This allows us to isolate unmodified OSGi bundles. Level-0 isolation environments expose a *Framework Management Service* that is used to perform management operations on the contained isolated framework, like installing, updating, and removing bundles. A *Shared Service Registry* spans all isolated frameworks. Every service registered with the local service registry of any isolated framework is transparently mapped to all other registries. Together, these features realize transparency of isolation (R6).

Isolation environments may contain an arbitrary number of subordinate environments[2] and isolated frameworks may host an arbitrary number of bundles. This ensures that resources are used efficiently (R2).

The *Isolated Framework Builder* employs *Environment Factories* to incrementally create the environment chain for a new isolated framework according to a given set of properties. For example, in case of native code extensions this includes the required operating system and CPU architecture. Initially, there is only a single framework available, which is called the *primordial framework*. Besides regular bundles, it also hosts the i-OSGi runtime system. For each isolation environment, except for those on level 0, an environment factory is made available through the shared service registry that is used by the framework builder to create subordinate environments.

A shared *Constraint Registry* service for *Isolation Constraints* is available to all isolated frameworks (R5) and can be used to register simple constraints of the form $(a,b,l)$ meaning that bundle $a$ should be isolated from bundle $b$ on level $l$ (cf. Chapter 8). Unprivileged bundles are not allowed to register constraints for other bundles, i.e., $a$ or $b$ must be the calling bundle itself. Although constraints are intentionally kept very simple, more complex constraints can be realized by transformations (R4). Examples are discussed in Section 9.3.

The *Isolation Orchestrator* is the core of the i-OSGi runtime system. It implements the algorithmic framework discussed in Chapter 8 to deploy bundles in accordance to the

1 cf. Section 7.1
2 There may be restrictions enforced by the isolation technology, e.g., the number of processes is usually limited by the operating system.

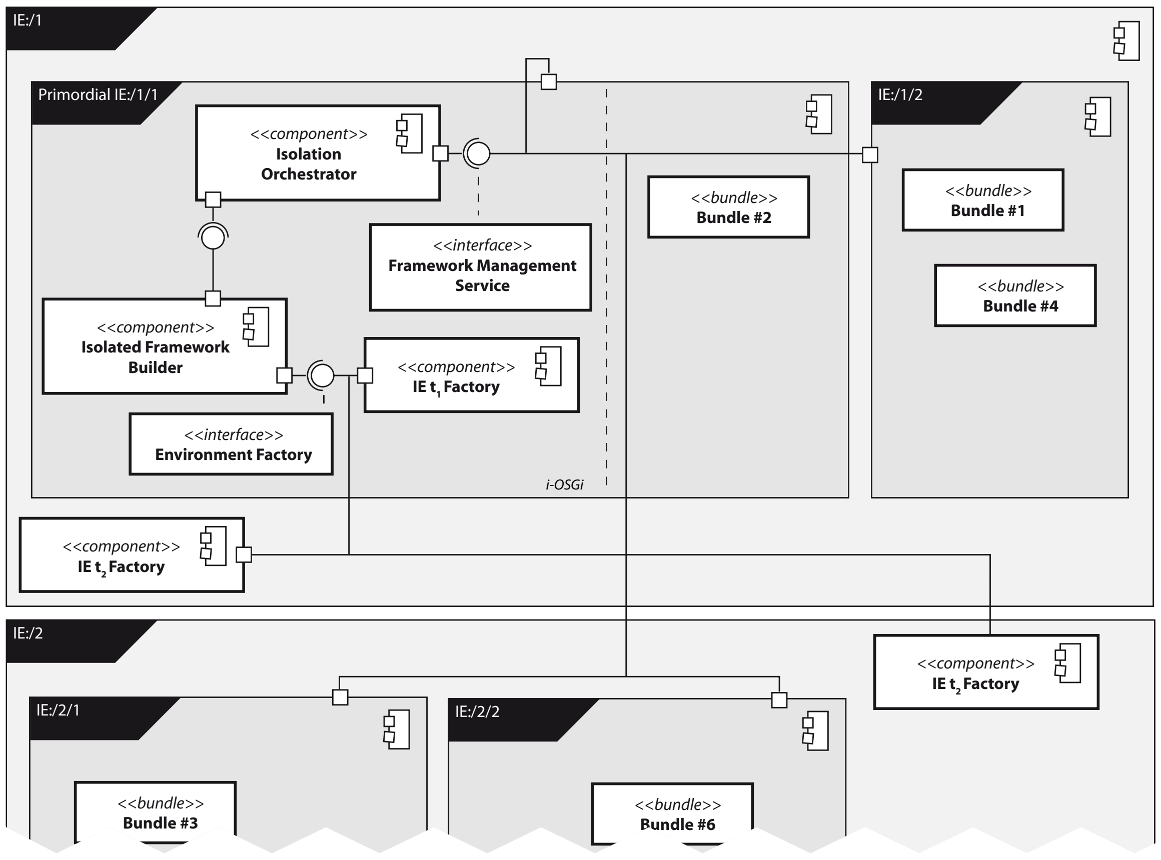

IE:/1
Primordial IE:/1/1
<<component>> Isolation Orchestrator
<<interface>> Framework Management Service
<<component>> Isolated Framework Builder
<<component>> IE t1 Factory
<<interface>> Environment Factory
i-OSGi
<<component>> IE t2 Factory
<<bundle>> Bundle #2
IE:/1/2
<<bundle>> Bundle #1
<<bundle>> Bundle #4
IE:/2
<<component>> IE t2 Factory
IE:/2/1
<<bundle>> Bundle #3
IE:/2/2
<<bundle>> Bundle #6

**Figure 9.2:** The architecture of I-OSGI depicted as an UML component diagram. For reasons of clarity, the shared registries for constraints and services are omitted. (IE = isolation environment)

registered constraints in a fully automated way (R3). To be able to accomplish this task, the orchestrator is aware of all existing isolated frameworks, intercepts bundle management operations, and is notified of changes to the constraint registry.

Interaction with the I-OSGI subsystems is accomplished by a façade service called the *Isolation Admin Service* (IAS).

## 9.3 Isolation Constraints

In I-OSGI there are two principal authorities who define isolation constraints: On the one hand, *bundle constraints* are defined by bundle issuers. On the other hand, I-OSGI allows infrastructure providers to register arbitrary *host constraints* through the IAS in the primordial isolation framework. This way a more restrictive isolation regime than the one implied by the bundle constraints can be enforced.

While host constraints are defined manually, bundle constraints are not: OSGi bundles use a file known as the *manifest* to declare all kinds of metadata. It contains key/value-pairs called *OSGi headers*. I-OSGI both adopts this practice by introducing a new header that allows bundle issuers to declare explicit bundle constraints and exploits the fact that metadata exists that can be analyzed to deduce implicit bundle constraints.

### 9.3.1 Explicit Bundle Constraints

To support the declaration of explicit bundle constraints, I-OSGI introduces the `Bundle-Isolation` OSGi header. The header must conform to the following EBNF grammar

```
Bundle-Isolation ::=  directive
                      {',' directive} ;
directive        ::=  'level' ':=' level
                      ';' filter ;
```

with `level` being an isolation level and `filter` being an LDAP-style filter predicate. A `directive` declares that the declaring bundle should be isolated on the given level from all bundles (except the declaring bundle) matching the given predicate. Predicates from directives targeting the same isolation level are combined using a logical conjunction (OR). I-OSGI translates a directive for bundle $b$ with level $l$ and predicate $p$ to a constraint by evaluating $p$ against the properties of bundle $b_i$ for each pair $(b,b_i) \in \mathcal{M} \setminus \{b\}$ adding a constraint $(b,b_i,l)$ via the IAS if $p$ is satisfied.

### 9.3.2 Implicit Bundle Constraint Deduction

I-OSGI interprets existing OSGi headers to automatically deduce bundle constraints. Subsequently, we discuss two use-cases:

**Singleton Bundles**. A bundle is identified by the header `Bundle-SymbolicName` which can have a `singleton` property indicating that only a single bundle with the given symbolic name can be active in the framework at any given point in time. To ensure that only a single singleton bundle is deployed to a framework, I-OSGI collects all bundles $\mathcal{S} = \{b_1,\ldots,b_n\}$ with a given symbolic name that are flagged as singletons and adds a constraint $\left(b_i,b_j,0\right)$ for all pairs $\left(b_i,b_j\right) \in \mathcal{S}^2$.

**Native Code Extensions**. `Bundle-NativeCode` headers encode dependencies on native code libraries for bundles with native code extensions. They associate the location of a library with a number of attributes which are matched against the hosting environment. The attributes relevant to I-OSGI are the name and version of the operating system, and the processor architecture, collectively called the *platform*. Let $\mathcal{N} = \{b_1,\ldots,b_n\}$ be the set of bundles with a `Bundle-NativeCode` header. To ensure that bundles with conflicting requirements on the host platform are not deployed within the same virtual machine, I-OSGI computes for every pair $\left(b_i,b_j\right) \in \mathcal{N}^2$ whether their requirements on the platform are compatible, i.e., whether there exists an intersection between the sets of allowed platforms. If not, a constraint $\left(b_i,b_j,3\right)$ is added that ensures that the bundles are deployed to different virtual machines.

## 9.4 Implementation

The open source i-OSGi implementation [ios] used in Cohesion is based on the *Apache Felix* v3.0.9 [fel] OSGi framework and the *R-OSGi* v1.0.0.RC4 [RAR07] Distributed OSGi implementation. The isolation environment types based on Child Frameworks and Isolates are currently not implemented as OSGi RFC 138 is not yet finalized and JSR-121 has not been integrated into any but an experimental JVM implementation called *Barcelona* [CDT03] that is only available for Sun OS. However, thanks to the extensible architecture of our implementation both can be easily integrated as soon as they become available.

The implementation of the Virtual Machine (VM) isolation environment type is based on Oracle *VirtualBox* v4.0 [vir] as it is remote controllable via web services. Moreover, it is very memory efficient by supporting to share common memory pages among (*Page Fusion*) and dynamic memory hand over between (*Memory Ballooning*) virtual machines. VM isolation environment instances are created from preconfigured virtual appliances. Out-of-the-box, i-OSGi provides low-footprint Linux (*Tiny Core Linux/Microcore* v3.5 [tin]) and Windows (*Windows PE* v3.0 [Mic]) virtual appliances.

### 9.4.1 Inter-Isolated Framework Communication

The i-OSGi shared service registry is implemented using *Apache Zookeeper* [RJ08]. Zookeeper is a high performance fault-tolerant centralized coordination system for distributed systems. *Zookeeper* provides a shared hierarchical namespace that is modeled in analogy to a filesystem. i-OSGi intercepts local service publications in isolated frameworks and creates a corresponding *Zookeeper* node containing the properties of the service and its *R-OSGi* endpoint, which is required to remotely invoke the service. Isolated frameworks listen for newly created service nodes and make the respective remote services available locally through a proxying mechanism provided by *R-OSGi*.

### 9.4.2 Isolation Engine

The isolation orchestrator of i-OSGi delegates solving the Minimum-MIP to an isolation engine that implements a parallel multi-strategy approach to cope with the NP-hardness of the problem. Multi-core CPUs are exploited by running two orthogonal strategies concurrently stopping either on success of one of the strategies or after a configurable timeout period. Both strategies produce intermediary solutions of increasing quality. On timeout the best solution found so far is used. For both strategies, the linear assignment problem encapsulated by the Match procedure (cf. Section 8.3) is solved using the polynomial time *Hungarian Algorithm* [Kuh55].

The first strategy is to apply an approximation algorithm for graph coloring called *Iterated Greedy* (IG) [CL96]. The translation of an MIP to a graph coloring instance has been described in the proof of Lemma 8.1.2. IG uses a *greedy* graph coloring algorithm repeatedly to discover better solutions in an incremental manner. The greedy algorithm visits the vertices of the input graph in a specific order and assigns the smallest color not already assigned to a neighbor vertex or a new color if no such color exists. IG permutes the order in which the vertices are visited in each iteration such that independent sets (vertices of the same color) discovered in the previous iteration remain adjacent. The exit condition for the algorithm is the number of iterations $i_{max}$ of the greedy algorithm performed after the last improvement (with respect to the number of colors used in the coloring) was detected. i-OSGi invokes the IG algorithm [HN] in an infinite loop with a geometric progression for $i_{max}$, i.e., $i_{max}(n) = 2^{n-1}$ for the $n$-th invocation, and uses the output of the previous invocation as the initial coloring.

The second strategy produces exact solutions (if run to termination) and consists in translating the MIP to a *Pseudo-Boolean* (PB) problem and to solve it using *Sat4J* [LBP10], a solver for satisfiability-based (SAT) problems. This approach is motivated by the hypothesis that the isolation constraints will probably be similar to real world trust relationships. The related PB problem thus will be structured. Since the invention of conflict-driven backtracking and dynamic clause learning, SAT solvers have become exceptionally efficient in solving structured problems from many real-world application domains [BHvMW09] (see also Part VIII). The mapping from MIP instances to PB problems is described subsequently.

### 9.4.3 Minimum-MIP as a Pseudo-Boolean Problem

A *linear Pseudo-Boolean (PB) problem* is an optimization problem over $n$ Boolean variables $\{x_1,\ldots,x_n\}$ of the form

$$\begin{aligned} \min \quad & c^T x \\ & Ax \geq b \\ & x \in \{0,1\}^n \end{aligned}$$

where $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$. $c^T x$ is called the *objective function* and $Ax \geq b$ are the *linear constraints*.

**Lemma 9.4.1.** *Every Minimum-MIP instance can be translated to an equivalent linear PB problem.*

*Proof.* Let $\mathcal{I} = \{e_1,\ldots,e_k\}$ be a set of isolation environments, and $\mathcal{S} = (\mathcal{M},\mathcal{A},\perp)$ an MAMS. Then the associated Minimum-MIP can be translated to a Pseudo-Boolean problem as follows (for an application of a similar encoding for exact graph coloring see [RMSA06]):

For each module $m_i \in \mathcal{M}$, we introduce $k$ Boolean *indicator variables* $x_i^1,\ldots,x_i^k$. Module $m_i$ is assigned to isolation environment $e_j$, iff $x_i^j = 1$. To guarantee that each module is assigned to exactly one isolation environment, we add for each module $m_i \in \mathcal{M}$ the two Pseudo-Boolean constraints

$$\sum_{j=1}^{k} x_i^j \geq 1 \text{ and } \sum_{j=1}^{k} -x_i^j \geq -1 \left( \leftrightarrow \sum_{j=1}^{k} x_i^j = 1 \right). \tag{9.1}$$

Each isolation constraint $(m_a, m_b) \in \perp$ is represented in *conjunctive normal form* (CNF)[1]:

$$\bigwedge_{j=1}^{k} \left(\neg x_a^j \vee \neg x_b^j\right). \tag{9.2}$$

This ensures that no modules that should be isolated are assigned to the same isolation environment.

Our goal is to minimize the number of isolation environments. A prerequisite to model this goal is to track to which environments at least one module has been assigned. Thus, we introduce $k$ *tracking variables* $y_1, \ldots, y_k$ and force the solver to set $y_i = 1$ iff at least one module has been assigned to the corresponding isolation environment $e_i$ by requiring

$$\bigwedge_{j=1}^{k} \left( y_j \Leftrightarrow \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \right). \tag{9.3}$$

This expression can be translated to CNF by eliminating biconditionals $(\alpha \Leftrightarrow \beta \leftrightarrow (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$,

$$\bigwedge_{j=1}^{k} \left[ \left( y_j \Rightarrow \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \right) \wedge \left( \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \Rightarrow y_j \right) \right] \tag{9.4}$$

eliminating the implications $(\alpha \Rightarrow \beta \leftrightarrow \neg\alpha \vee \beta)$,

$$\bigwedge_{j=1}^{k} \left[ \left( \neg y_j \vee \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \right) \wedge \left( \neg \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \vee y_j \right) \right] \tag{9.5}$$

and finally distributing $\wedge$ over $\vee$ after moving negations inwards in the second term. The resulting CNF clauses are

$$\bigwedge_{j=1}^{k} \left[ \left( \neg y_j \vee \bigvee_{i=1}^{|\mathcal{M}|} x_i^j \right) \wedge \left( \bigwedge_{i=1}^{|\mathcal{M}|} \neg x_i^j \vee y_j \right) \right]. \tag{9.6}$$

The objective is to minimize the number of isolation environments to which at least one

1 A formula in CNF is a conjunction of *clauses* of the form $l_1 \vee \ldots \vee l_n$. The *literals* $l_i$ are variables or negated variables. Each clause can be translated trivially into a linear PB constraint $\sum_i l_i \geq 1$.

module has been assigned. This translates to the objective function

$$\min \sum_{i=1}^{k} y_i. \tag{9.7}$$

■

### 9.4.4 Symmetry Breaking

Our implementation employs two lightweight but effective symmetry breaking techniques to assist the Pseudo-Boolean solver in pruning the high-dimensional search space. One is equivalent to *Selective Coloring* [RMSA06] and consists in preassigning the most constrained module and its most constrained neighbor in the constraint graph. The other consists in adding a first-order logic constraint

$$\exists y_i \left[ y_i \in \left\{ y_1, \ldots y_{j-1} \right\} \wedge \neg y_i \right] \Rightarrow \neg y_j \tag{9.8}$$

for every tracking variable $y_i$. This ensures that a module can be assigned to an isolation environment $e_i$ only if at least one module has been assigned to each $e_j$ with $j < i$. Thus, the search space becomes significantly smaller. The existential quantifier $\exists$ can be trivially translated into a conjunction

$$\bigvee_{i=1}^{j-1} \neg y_i \Rightarrow \neg y_j. \tag{9.9}$$

After eliminating the implication, we get the CNF equivalent

$$y_j \vee \bigvee_{i=1}^{j-1} y_i. \tag{9.10}$$

## 9.5 Performance Evaluation

To substantiate the usefulness of i-OSGi , we conducted a performance evaluation for the key aspects of the system. This includes analyses of the time required to create and destroy isolated frameworks, the service invocation overhead for different isolation levels, and the performance of i-OSGi's isolation engine. The former two experiments are conducted on a Type I the latter on a Type II machine (see Table 9.1).

### 9.5.1 Isolated Framework Management

Table 9.2 shows the time it takes to create and the time it takes to destroy an isolated framework under a given parent location. Creation involves spawning a level-0 isolation

| | **Hardware** | | **Software** | |
|---|---|---|---|---|
| | CPU | Memory | OS | Version |
| **I** | Intel®Core 2™Duo<br>2 Cores @ 2.00GHz<br>4MB Cache / Core | 3GB | Windows | 7 |
| **II** | Intel®Xeon™X5365<br>8 Cores @ 3.00GHz<br>4MB Cache / Core | 8GB | Linux | 2.6.32 |

**Table 9.1:** Hardware and software configuration of the computers used for the performance evaluation

| **Type** | **Parent** | **$T_{Create}$ [ms]** | **$T_{Destroy}$ [ms]** |
|---|---|---|---|
| VM | / | 27097 ± 760 | 9597 ± 760 |
| Process | /1 | 2943 ± 392 | 123 ± 48 |
| | /* | 2644 ± 174 | 98 ± 25 |

**Table 9.2:** Mean time to create ($T_{Create}$) and destroy ($T_{Destroy}$) isolation environments based on 50 program runs (* denotes an arbitrary index)

environment (Process) either in an existing or in a newly created level-1 isolation environment (VM/Host). In both cases an OSGi framework and the I-OSGI runtime system are deployed and configured. In the latter case, a VM instance (*Tinycore Linux/Microcore*) is configured and launched within which the level-0 isolation environment is created as described. After launching the OSGi framework the isolation admin service blocks until the respective isolated framework service is published over the shared service registry and thus becomes available for use.

Destruction of an isolated framework consists in terminating the level-0 environment by shutting down the OSGi framework explicitly (initiating an orderly shutdown) or implicitly by shutting down the hosting VM. In both cases the shutdown is followed by a clean-up procedure that removes OSGi- and VM-related files.

As expected, the time required to create an isolated framework within a non-existing level-1 environment (/) is dominated by the costs associated with launching the VM. While creating a process within an existing VM (/i) is even slightly faster than within the primordial level-1 isolation environment (/1), the overhead is more than 9-fold in this case. Together with the fact that running a VM requires orders of magnitude more main memory than a process, this observation underlines the importance of the ability of our approach to provide least cost isolation environment setups and to share isolation environments whenever possible.

| **Isolation Level** | **Caller Location** | **Callee Location** | $\mathbf{T_{Invoke}}$ **[¯s]** |
|---|---|---|---|
| - | /1/i | /1/i | 0.452 |
| | /i/j | /i/j | 0.699 |
| 0 | /1/i | /1/j | 133 |
| | /i/j | /i/k | 492 |
| 1 | /1/* | /j/* | 464 |
| | /i/* | /k/* | 570 |

**Table 9.3:** Service invocation times for different inter-isolated framework communication scenarios (* denotes an arbitrary index and $i \neq j \neq k \neq 1$)

## 9.5.2 Inter-Isolated Framework Communication

Table 9.3 shows the service invocation overhead for all possible inter-isolated framework communication scenarios. Intra-process invocation (no isolation) gives the baseline performance with 452 ns in the primordial and 699 ns in VM-based level-1 isolation environments. The 54 % difference can be attributed to virtualization overhead. There is an obvious trend towards higher overheads for higher isolation levels: Inter-process invocation for bundles isolated on level-0 is roughly 300 times and for bundles isolated on level-1 is roughly 1000 times slower. These large differences are due to R-OSGi's socket-based communication and virtualization overhead. The latter can be seen by comparing the invocation time for the setups with modules isolated on level-1.

## 9.5.3 Isolation Engine

Solving the Minimum-MIP lies at the core of I-OSGI's isolation engine. Figure 9.3 shows a head-to-head runtime comparison of the Iterated Greedy and the Pseudo-Boolean solving strategies for 880 random MIP instances (with varying constraint densities and numbers of modules) with a 1 minute timeout. A cross indicates the time each strategy required to find the best solution with respect to the number of isolation environments. A timeout means that the respective solving strategy was not able to find a solution as good as the solution found by the other strategy. There were 117 timeouts for the IG and 366 timeouts for the PB strategy.

Figure 9.4 shows the probabilities that the PB strategy computes the better solution for the same 880 instances used above. Obviously, PB is superior for constraint densities between $0.1$ and $0.3$. For the special cases in which there are no or all possible constraints present the strategies are approximately on par.

Both figures clearly show that the performance characteristics of the strategies are significantly different and thus justify our approach to run both strategies in parallel.

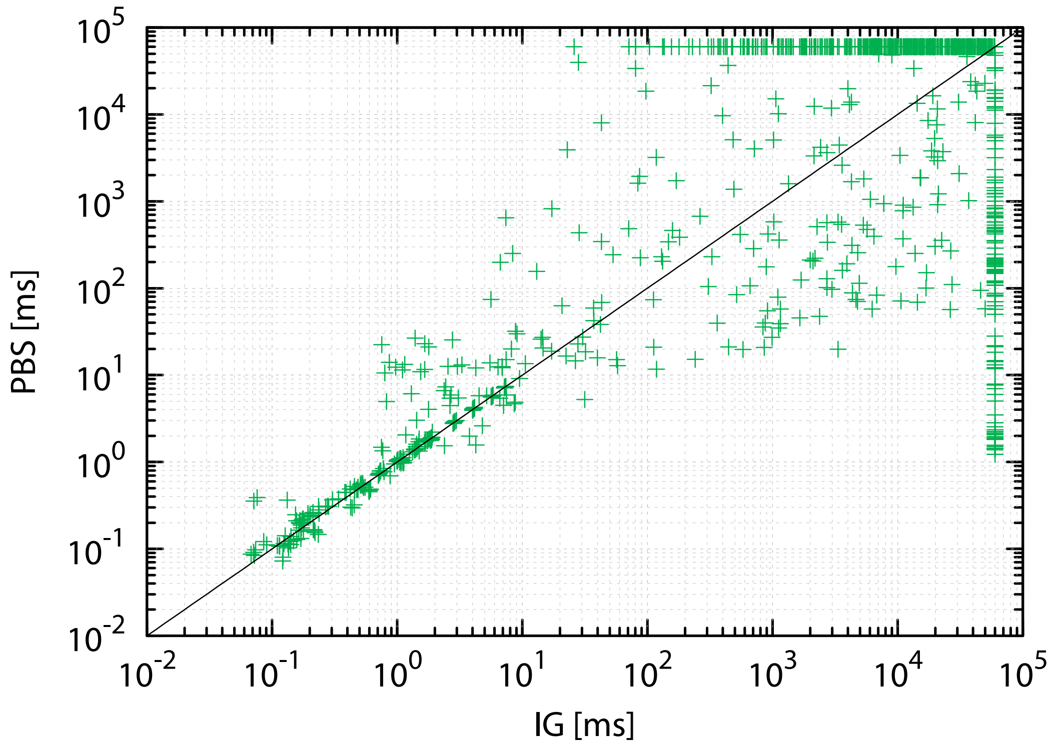


**Figure 9.3:** Head-to-head performance comparison for I-OSGI's MIP solving strategies

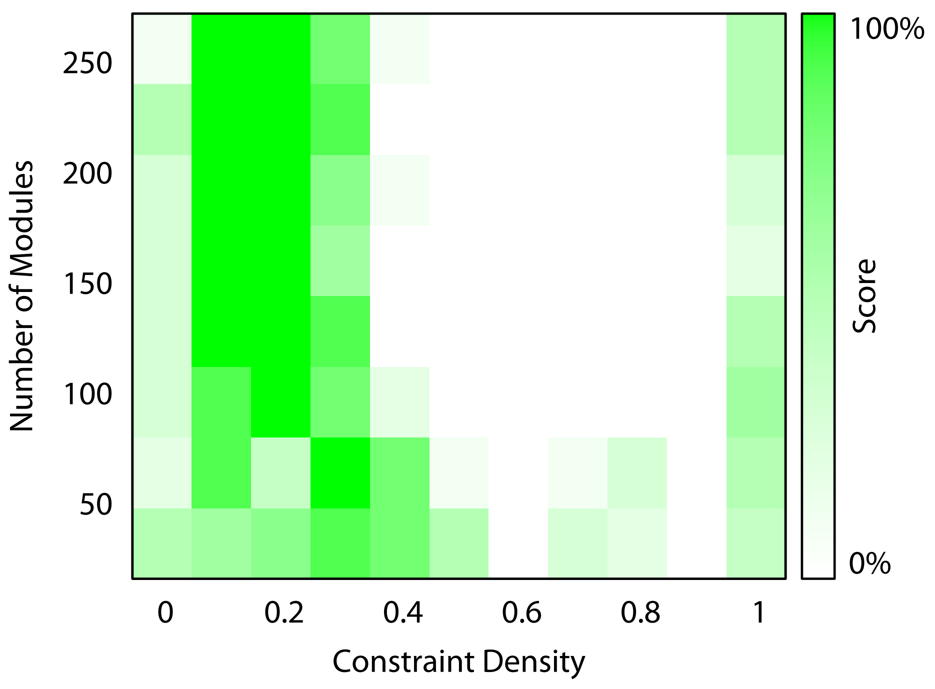


**Figure 9.4:** Probability of the Pseudo-Boolean strategy computing the better solution

# 10 Related Work

There is a large body of research concerning isolation techniques on various levels of a computer system, including programming languages, middleware, and operating systems. For a survey see [VN].

In the following, we focus on approaches targeted at component and module systems. None of the solutions discussed below is fully automated like I-OSGI or allows for adjusting the degree of isolation between modules. Furthermore, they are not able to provide a compatible runtime environment to modules comprising incompatible native code.

Gama et *al*. propose an approach [GD09, GD10] focused on enhancing the dependability of OSGi-based applications that make use of untrustworthy third-party code. Their system shields the main application by running third-party code in a fault contained security sandbox. They describe two alternative sandbox implementations: one based on isolates and another one using operating system processes. The focus of their approach lies on automatic self-recovery after the sandbox hangs or crashes. The proposed technique is orthogonal to isolation and could be applied to I-OSGI as well.

Wegner discusses a system [Weg09] that employs operating system processes to isolate groups of bundles within so-called *domains*. This approach ensures fault containment and is suitable for resource usage accounting and quota enforcement. However, in contrast to I-OSGI the system does not support multi-authority environments and requires significant manual configuration including manual mapping of bundles to domains, which is not feasible for large applications consisting of hundreds of bundles.

Geoffray et *al*. use a modified JVM called *I-JVM* [GTFC08, GTM+09] to provide isolation and resource accounting. Unfortunately, the underlying JVM implementation *J3* is by factors slower than state-of-the-art JVMs [GTL+10] and the modifications of *I-JVM* add another 20% of overhead. Depending on the bundle interaction patterns, the approach at its current state may be inferior to isolation approaches based on sandboxing with respect to performance. In contrast to I-OSGI, the approach does not provide native code security.

The work of Frenot et *al*. [RFLM06] is targeted at residential gateways. Their approach is to launch child frameworks within a master framework in a single JVM. The resulting isolation is rudimentary and limited to namespace isolation. Meanwhile, OSGi RFC 138 – used by I-OSGI to create isolation environments with the lowest degree of isolation – subsumed their approach.

Isolation concepts have also been integrated into other component frameworks. For example, Microsoft's *Component Object Model* (COM) [Low01] and its successor the .NET platform [SNS03] both provide restricted isolation mechanisms: COM components can either be loaded into the host application or deployed to a separate process. The *Managed Add-In Framework* (MAF) from .NET allows for deploying *add-ins* to an *application domain* – a concept similar to Java isolates – or to a separate process. Both approaches provide fault containment when a separate process is used.

# Part IV

# Peer-to-Peer Management*

Peer-to-Peer Desktop Grid systems are notoriously hard to manage: The large number of manageable entities hosted by volatile peers distributed over administrative domains and the absence of central control disperses system management actions in time and space. Hence, even straightforward management tasks tend to be cumbersome and error-prone. Thus, the lack of assisting management technology is a limiting factor for successfully operating even moderately sized P2P Desktop Grids. COHESION tackles this issue with an integrated Peer-to-Peer management solution that adopts and adapts existing industrial grade management standards to support scalable remote management of volatile resources within multi-authority environments.

## RELATED PUBLICATIONS

[SB07] SCHULZ, Sven and BLOCHINGER, Wolfgang: **An Integrated Approach for Managing Peer-to-Peer Desktop Grid Systems**, In: *Proc. of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, Rio de Janeiro, Brazil, S. 233–240

∗ This part contains a summary of the respective topic. For further details the reader is referred to the related publications.

# 11 Overview

Our management solution is designed as a component-based architecture that can be subdivided into six layers of functionality (see Figure 11.1): On top of the industrial grade *Java Management Extensions* (JMX) [Sunb] management standard, we provide support for *role-based management*, through a role-aware access control component that is accompanied by an infrastructure for role and policy management. Role-based management is an essential building block for supporting multi-authority environments as it allows for defining *who is allowed to do what* in a fine-grained way. A P2P JMX connector[1] and a mechanism to discover remote management agents are used to leverage an arbitrary COHESION peer, called a *management gateway*, to securely access any other peer within the system. We refer to this feature as *Peer-to-Peer Management*. Peer-to-Peer management provides a basis for managing large numbers of peers with virtualization techniques called *agent cascading* and *bean clustering*. While the former allows to map remote managed objects – called *MBeans* in JMX jargon – into the local agent analogous to mounting remote file systems in modern operating systems, the latter is a multiplexing mechanism to perform management operations on a number of managed beans as if it was a single one. To ease complex and recurring management tasks, a scripting component is provided that allows combining individual management tasks into script libraries. A service for *Disconnected Management* handles volatility by delayed execution of management scripts on temporarily absent peers. Finally, an extensible unified management tool depicted in Figure 11.2 integrates all these components into an *Integrated Management Workbench* built on top of the *Eclipse* IDE

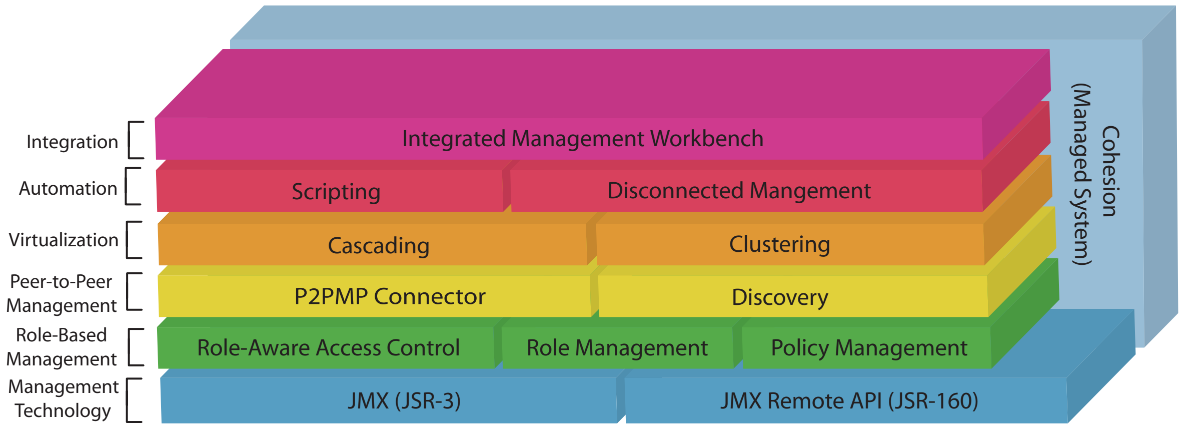


**Figure 11.1:** COHESION's layered management architecture

1 JMX *Connectors* as specified in [Suna] are used to make managed beans remotely accessible over a network.

[Ecl].

Subsequently, the individual services are described briefly. For an in-depth discussion, the reader is referred to [SB07].

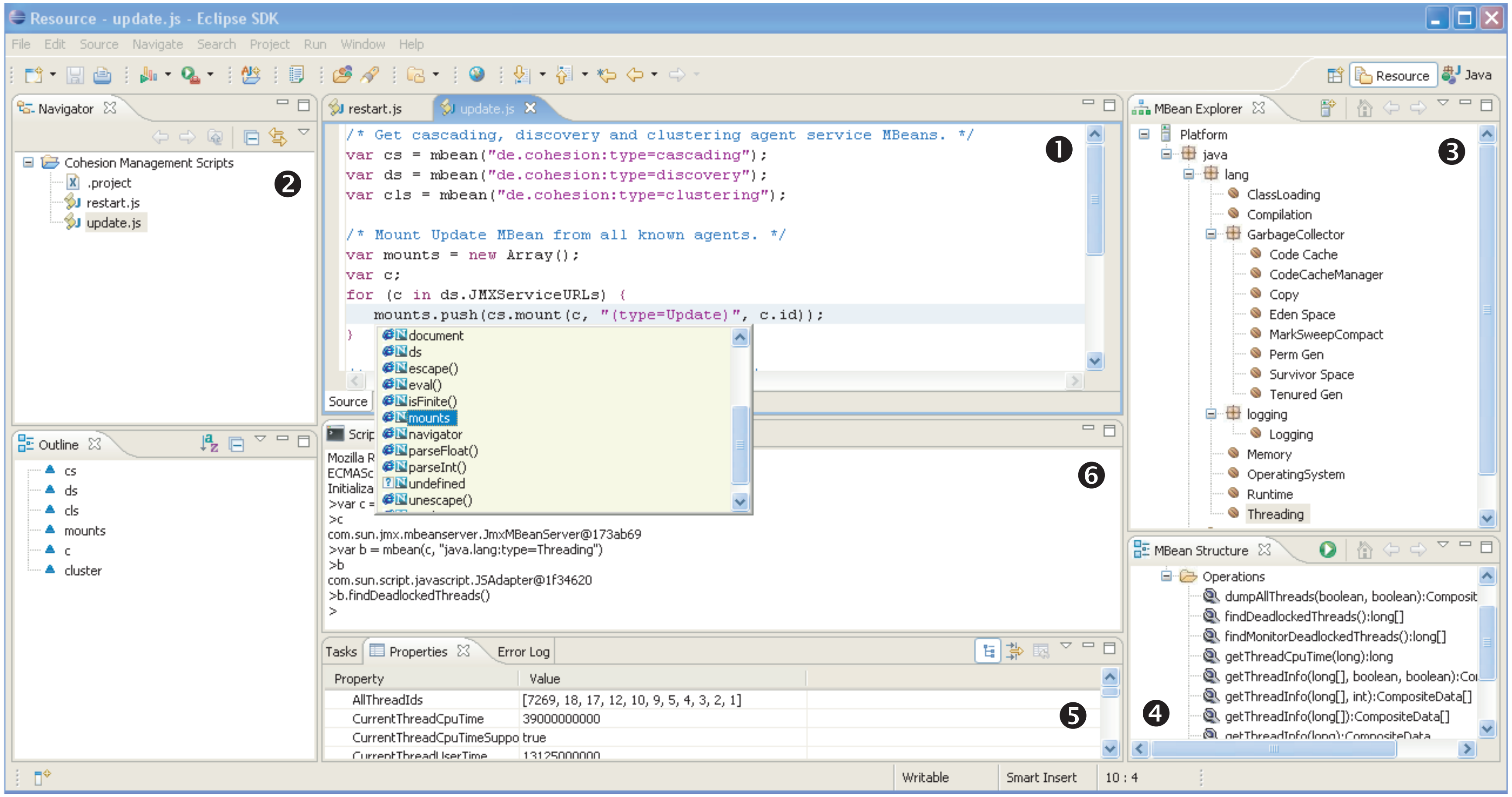
## 11.1 Role-Based Management

Our approach to manage Desktop Grids is based on splitting up authority using *management roles*. Roles are traditionally used to model authority and responsibility in distributed systems [LS97]. Their application results in reduced policy maintenance complexity, since groups of users can be treated as a single entity. Within our approach, management authority is delegated by transferring *management permissions* from one role to another. For example, by installing an application the host owner transfers permission to control application-related objects to the application issuer.

JMX provides permissions for all aspects of interaction with management agents and the MBeans deployed therein. This includes visibility of managed beans, access to their attributes, and invocation of their operations. However, Java's built-in security architecture [LG03] is insufficient for role-based access control due to a static policy implementation: Neither can permissions assigned once be later revoked nor can additional permissions be granted. Thus, we use the *OSGi Conditional Permission Admin Service* to consult a role database to dynamically determine all roles assigned to the invoking subject. To administer roles and permissions, two agent services, the *Role Management Service* and the *Policy Service*, are provided. While the former is used to assign roles to users and to modify the set of available roles, the purpose of the latter is to grant permissions to these roles or to revoke them.

## 11.2 Peer-to-Peer Management

In traditional management scenarios, management interactions are typically client/server operations, often with a central management gateway. By contrast, within our approach there is no permanent dedicated management gateway. Instead arbitrary peers can — by the operator's decision — become a gateway for temporarily managing the whole system (insofar as allowed by the operators roles/permissions). In analogy to Peer-to-Peer communication patterns, we refer to this feature as *Peer-to-Peer Management*. It provides a technical basis for decentralized management through the virtualization and automation techniques described below. To allow for P2P management, we have to provide the operator with two abilities: to discover remote management agents and to perform management operations on them. The former is implemented as a discovery agent service that advertises JMX connectors attached to COHESION peers. To allow for the latter, COHESION deploys a custom connector called the *Peer-to-Peer Management Protocol* (P2PMP) connector that uses ORBWEB for communication and thus is able to bridge network segmentations between peers that are mutually shielded from each other by restrictive firewalls.



```
/* Get cascading, discovery and clustering agent service MBeans. */
var cs = mbean("de.cohesion:type=cascading");
var ds = mbean("de.cohesion:type=discovery");
var cls = mbean("de.cohesion:type=clustering");

/* Mount Update MBean from all known agents. */
var mounts = new Array();
var c;
for (c in ds.JMXServiceURLs) {
  mounts.push(cs.mount(c, "(type=Update)", c.id));
}
```

```
Mozilla R
ECMASc
Initializa
>var c =
>c
com.sun.jmx.mbeanserver.JmxMBeanServer@173ab69
>var b = mbean(c, "java.lang:type=Threading")
>b
com.sun.script.javascript.JSAdapter@1f34620
>b.findDeadlockedThreads()
>
```

| Property | Value |
|---|---|
| AllThreadIds | [7269, 18, 17, 12, 10, 9, 5, 4, 3, 2, 1] |
| CurrentThreadCpuTime | 39000000000 |
| CurrentThreadCpuTimeSuppo | true |
| CurrentThreadUserTime | 13125000000 |

**Figure 11.2:** COHESION's *Integrated Management Workbench*: The topmost view in the middle column (❶) shows the *Eclipse* source code editor for scripting languages with code completion engaged. The script being edited is defined in a library project called `Cohesion Management Scripts`, whose structure is visualized in the *Navigator* view (❷). Scripts can be deployed easily to any connected agent. For interacting with managed beans the workbench provides specialized views: The *Agent Explorer* (❸) visualizes the tree of managed beans exposed by agents the workbench is currently connected to. The features of the currently selected managed bean are displayed in the *MBean Structure* view (❹). Integration with the *Properties* view of *Eclipse* (❺) provides the operator with a live, random access view on its attributes. Finally, there is an interactive *Scripting Shell* (❻) with full access to the managed bean trees of all connected agents.

## 11.3 Virtualization

COHESION's managed entity *virtualization* services – *Agent Cascading* and *Bean Clustering* provide a single, consolidated, logical view of the management aspects of a whole set of individual managed entities.

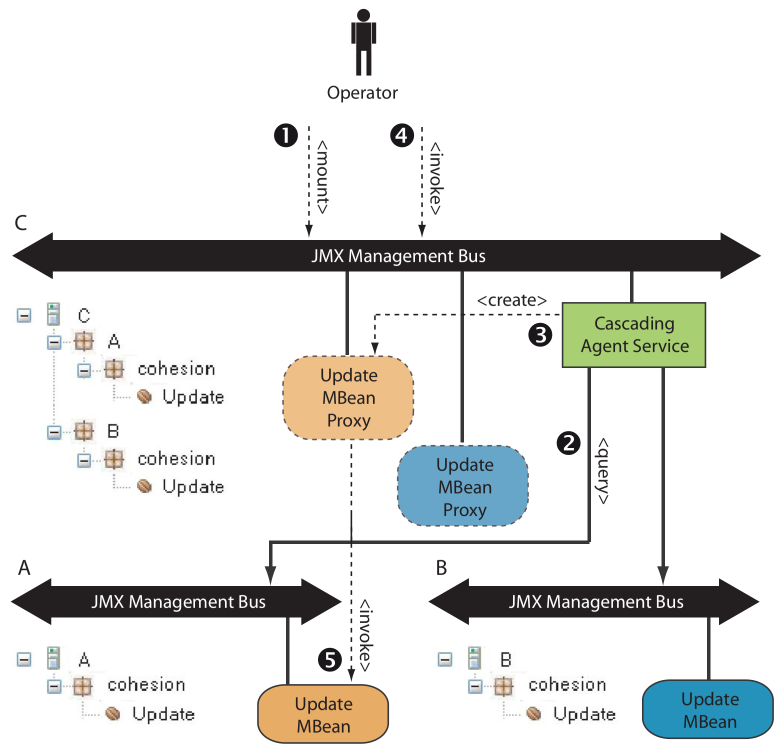
Agent cascading as proposed first in [Pac05] and depicted in Figure 11.3a is the process of integrating (parts of) the tree of managed entities of a remote agent into the local one. Cascading abstracts from the location of a managed entity by collocation. With cascading, an operator can contract the management domain of the entire distributed system into a single management agent. Thus, the whole set of managed entities becomes accessible to an operator at the gateway agent without the need to establish sessions with each and every node of the system manually.

Bean clustering, as depicted in Figure 11.3b, abstracts from differences in concrete management interfaces by automatically extracting a common unified management interface. An operation performed on a managed bean cluster is transparently dispatched to all aggregated beans. The management interfaces of bean clusters are dynamic, i.e., changes in the MBean tree throughout the clusters lifetime are automatically reflected. With bean clustering, an operator can subsume invocations of the same operation performed on a whole set of managed entities, that would otherwise have to be performed manually one by one on each managed entity separately. Apart from the increased convenience, there is also a speedup in the execution of management tasks, since batch invocations can be performed in parallel. In conjunction with cascading, it's possible to cluster managed beans from different agents. For example, an application provider is able to adjust an application parameter on all hosts running the application with a single management operation.

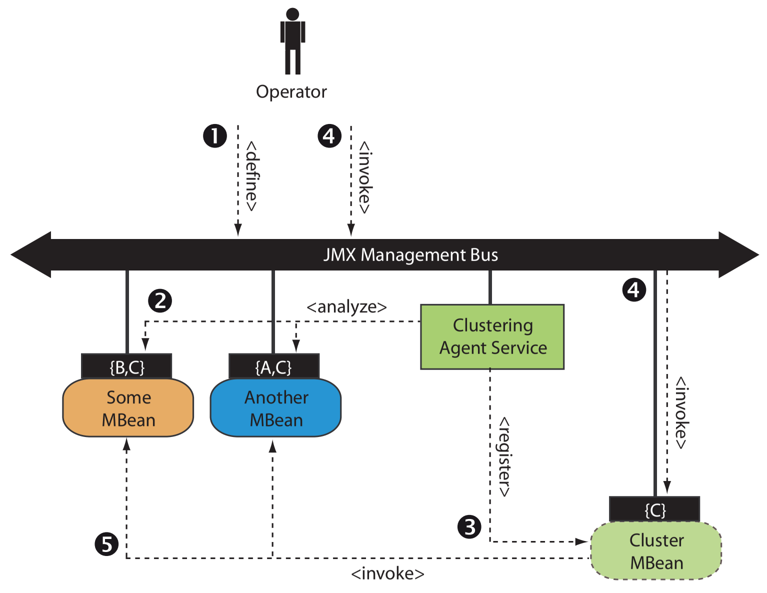
## 11.4 Automation

With possibly thousands of volatile peers within a Desktop Grid system, a management task is scattered in time and space. With *Peer-to-Peer Management* and virtualization, the operator is relieved from the burden of having to connect to each and every management agent and to perform the required management operations manually. However, the problem, that probably some peers will be offline while a management task is performed is still not addressed. With *MBean Scripting* and *Disconnected Management*, we propose two management automation techniques to tackle this problem.

The idea behind MBean scripting is to adopt a well proven automation technique from the traditional management domain of computer administration. In contrast to the simpler batch execution style of bean clusters, scripting allows for context-aware task execution by using conditional control flow constructs provided by scripting languages. The full potential of MBean scripting unfolds when it is used in conjunction with the virtualization and discovery agent services described above: By mounting parts of the managed bean trees of remote management agents and creating clusters from MBeans of interest, performing distributed management tasks is considerably simplified. Moreover, since clusters transparently adapt to changes in the set of aggregated MBeans, clustering is an effective instrument to shield from the effects of volatility during script execution. As scripts can be executed asynchronously,



**(a)** With *Agent Cascading* the operator mounts a remote object tree by specifying its network address, an object name pattern to select the subset of MBeans to be made available, and a mount path where the remote object tree should be made available (❶). The local cascading service contacts the remote agent and queries for matching MBeans (❷). For each a local proxy is created that is registered with the local MBean server (❸) after prefixing the object name with the specified mount path. Eventually, the operator invokes an operation on one of the proxies (❹). The proxy transparently forwards the invocation to the remote MBean (❺).



**(b)** With *Bean clustering* the operator defines a cluster by specifying an object name pattern (❶). The clustering service then matches the pattern against all available MBeans and extracts the common MBean features (which is a single operation named *C* in the example depicted) (❷). These common features are exposed as an *MBean Cluster* (❸). Eventually, the operator invokes *C* on the MBean cluster (❹). Finally, the MBean cluster forwards the invocation of *C* to all aggregated MBeans and returns the collected results (❺).

**Figure 11.3:** Functioning of COHESION's managed entity *virtualization* services

they can be used to extend the managed system with new scripted functionality. This resembles the approach described in [GY95].

To cope with volatility, task initiation and task execution are decoupled. Thus, an operator can manage disconnected peers by delaying script execution until the peer reconnects. Therefore, we refer to this concept as *Disconnected Management*. For the sake of scalability, group models employed for P2P Desktop Grids often provide only partial membership views. Thus, even if all peers are online, they may be not visible to the gateway agent. To cope with this fact scripts may be propagated throughout the system by gossiping.

## 11.5 Related Work

As of our knowledge, there is no other comprehensive management solution that addresses the domain-specific problems in the area of managing P2P Desktop Grids. However, there are several projects in the field of systems management that are based on the same management technology as our approach. Subsequently, we discuss the most prominent of them.

*XtremeJ* [Xtr] and the products of the IBM *Tivoli* brand [IBM] are commercial management solutions based on *Eclipse* technology. Their focus is on management of J2EE application servers. A distinguishing feature of *XtremeJ* is the proprietary *Management Query Language* (MQL) used for performing complex queries for MBeans satisfying certain conditions. *Tivoli* targets additional application domains, like mainframe, storage, and security management.

*JManage* [jma] is an open source, web and command-line based JMX client. It supports application clusters similar to COHESION's bean clusters. However, clustering is realized as a client-side service and is restricted to MBeans exposing exactly the same management interface. Furthermore, *JManage* supports fine-grained access control based on *Access Control Lists* (ACL) . Since ACLs are implemented as a client-side mechanism, the approach is not suitable to reliably protect networked systems from unauthorized access.

*JConsole* [Sunc] is a simple yet extensible management console bundled with Java SE 5. Its primal field of application is to monitor non-distributed systems. To showcase the extensibility the console is bundled with a plugin with scripting support.

# Part V

# Network Substrate

A fundamental component of the Cohesion Desktop Grid middleware is a hybrid P2P *network substrate* called Orbweb that provides support for efficient node interaction over a virtual overlay network. High Performance Desktop Grid Computing puts particularly high demands on the underlying network substrate. Existing substrates like *JXTA* [TAA+03] or *SpoVNet* [BHMW11] fall short in satisfying all requirements. Orbweb is designed to close this gap. It leverages the open industrial-strength *eXtensible Messaging and Presence Protocol* (XMPP) to tackle domain-specific challenges, including system scale, resource volatility, and network segmentation induced by middle boxes like NAT devices and firewalls. Orbweb extends XMPP with dynamically negotiated P2P communication channels, superpeer-managed peer groups with customizable virtual topologies, efficient probabilistic multicasting, and improved protocol efficiency.

## Related Publications

[SBP10] Schulz, Sven; Blochinger, Wolfgang and Poths, Matthias: Orbweb – **A Network Substrate for Peer-to-Peer Grid Computing based on Open Standards**. *Journal of Grid Computing* (2010), Bd. 8(1):S. 77–107

[SBP09] Schulz, Sven; Blochinger, Wolfgang and Poths, Mathias: **A Network Substrate for Peer-to-Peer Grid Computing beyond Embarrassingly Parallel Applications**, In: *International Conference on Communications and Mobile Computing (CMC 2009)*, IEEE Computer Society, S. 60–68

In analogy to the development of Grid Computing, which experienced a phase of consolidation through standardization in the last decade, we believe that Desktop Grid Computing has to go through a similar process by adopting existing open standards to tap its full potential. By leveraging open standards one can profit from high interoperability, improved robustness, and prolonged durability leading to lower and manageable risk and efficient use of existing resources. In particular, this allows the community to concentrate on overarching research challenges.

To this end, we propose to employ the industrial strength *eXtensible Messaging and Presence Protocol (XMPP)* to build a generic network substrate for Peer-to-Peer Desktop Grid Computing called ORBWEB. While several projects already made an ad hoc transition to XMPP (see Chapter 23), we strive to pave the way for a wider adoption of XMPP by systematic extension and optimization of the core protocols. Our key contributions are as follows:

1. We specify functional and non-functional requirements defining a network substrate suitable for High Performance Desktop Grid Computing scenarios and demonstrate that the abstractions and the existing infrastructure of XMPP are basically well-suited to satisfy these requirements.

2. We describe how a great deal of those requirements can be satisfied using the XMPP core standard and its extensions.

3. We contribute novel concepts and features amalgamated into our network substrate to further increase the applicability of XMPP for High Performance Desktop Grid Computing.

The remainder of this part is organized as follows: In Chapter 12, we identify functional and non-functional requirements for a network substrate suitable for High Performance Desktop Grid Computing with the interaction and organizational model described in Part I. After giving an overview on ORBWEB in Chapter 13, we explain how the elements of XMPP can be used and amended to satisfy the identified requirements in Chapters 14-16. In Chapters 17-20, we present our extensions designed to meet the requirements not yet satisfied by plain XMPP. Chapter 21 presents supportive tools to visualize virtual peer topologies and to analyze network traffic on superpeers. In Chapter 22, we present a detailed performance and scalability evaluation that clearly demonstrates the suitability of our approach and the improvements of our extensions over plain XMPP. Finally, Chapter 23 relates our approach to the state of the art.

## Terminology and Notation

XMPP originally has been designed as a client/server system for Instant Messaging applications. Although, it has evolved beyond these roots since then, the standard documents still make heavy use of the original terminology. Hence, the terms XMPP client/server and ORBWEB peer/superpeer are used interchangeably in the following.

The modular notation and the pseudo code used to specify the APIs and algorithms of this chapter is introduced in Appendix B.

# 12 Requirements

In this chapter, we identify *functional* and *non-functional requirements* for a P2P Desktop Grid Computing substrate. We define the term *functional requirement* as an operation or a set of operations a substrate must provide to support the P2P organizational and interaction model within a Desktop Grid environment. *Non-functional requirements* describe the qualities of these operations.

## 12.1 Functional Requirements

**(F1) Peer Groups.** *Peer Groups*[1] are sets of peers grouped together to cooperatively provide a service. They are used to structure applications and to model the environment they are executed in. As both may be dynamic and of arbitrary complexity, groups have to be lightweight first class objects such that a peer can create as many groups as required. Membership in groups should be dynamic to support volatility and evolving inter-peer relations as prevalent in ISP applications. Moreover, membership should be reified such that changes in membership can be used as an input to distributed algorithms. Pushing the concept of peer groups down the stack into the middleware relieves the application programmer from the burden of implementing groups by himself and many aspects of writing a distributed application become considerably simpler. From a technical point of view, groups provide a scope for communication and establish a security context within which applications can execute in isolation from each other. Thus, the group abstraction is an essential means to realize the P2P organizational model described in Part I.

**(F2) Virtual Topologies.** Members of a peer group are provided with a dynamically updated list of other members from the same group. These lists are called *views* and its constituents are called the *neighbors* of the local peer. The views of all peers within a peer group together induce a single distinct overlay network topology among the peers in the very same way as the union of all sets of outgoing edges of a graph defines the graph's structure (except for unconnected vertices).

As discussed in Part I, P2P Desktop Grids are characterized by a potentially large scale in combination with high volatility and by peer interaction patterns that are diverse and evolving. In existing systems with an explicit group model, views are either complete and of limited scalability or partial with limited support for application-specific interaction patterns. ORBWEB goes beyond that by enabling applications to select *virtual topologies* that best match their specific requirements. These topologies are called *virtual* as they are built

1 Peer groups are also known as *distributed process groups* throughout the literature.

from logical links as an overlay network that, in general, is different from the underlying physical network. As peer groups are used for structuring applications, they are the most natural scope for virtual topologies. Despite this relation, group membership and logical peer neighborship (defined by virtual topologies) are independent concepts. To support the dynamic nature of P2P interaction patterns, virtual topologies should be hot-swappable. To facilitate the implementation of distributed algorithms, view entries should be decorated with the role a neighbor plays with respect to the local peer, e.g., `parent` and `child` in a tree or `predecessor` and `successor` in a ring topology.

**(F3) Communication.** Communication primitives are classified according to the number of sources and sinks involved: Multicast (one-to-many) communication is the delivery of a message from a single source to a set of sinks. Unicast (one-to-one) and broadcast (one-to-all) are conceptually special cases of multicast where the message is sent to a single recipient or to all possible recipients, respectively. Many-to-one communication and all-to-one communication transfers data from many/all sources to a single sink. When data is consolidated by means of a many/all-to-one operation the respective primitive is referred to as a *reduce* operation.

Many-to-many and all-to-all communication can be decomposed into a many-/all-to-one and a one-to-many/-all communication operation. This approach of composing complex communication primitives from simple ones allows for supporting a broad spectrum of functionality without a significant increase in the complexity of the middleware. Thus, the concept has found its way into other middleware as well. A prominent example is the *Internet Indirection Infrastructure* (I3) [SAA+04].

Hence, for full coverage of these communication primitives, our substrate should support multicast, unicast, and many-to-one[1] communication. In conjunction with the group abstraction, multicast is referred to as *groupcast* in the following. The broadcast primitive can be mapped to a groupcast within a special peer group[2] that all peers of the Desktop Grid join.

**(F4) Fail-Stop Distributed System Model.** Many problems in asynchronous distributed systems become *impossible*[3] to solve when processes are prone to failure. *Consensus* is a prominent example that has been shown by Fischer, Lynch, and Paterson in their famous FLP proof to be impossible in asynchronous message passing systems, even if at most one process may fail and all communication channels are reliable. To simplify the implementation of such distributed services and applications or to make them possible in the first place, ORBWEB should implement a fail-stop distributed system model that provides strong

1 Note that scalable many-to-one communication typically incorporates the repeated execution of a reduction operation. For this reason, it is considered to be a computing service in the context of this work. It is implemented on a higher layer of the COHESION middleware stack as an information aggregation service (see Part VI).

2 As all peers are members of this peer group it must use a scalable virtual topology and a scalable multicast primitive (see Chapter 19).

3 In this context, the term *impossible* is used differently than in everyday language as it means that there is no algorithm that can *always* reach consensus in bounded time.

abstractions including processes with crash-stop behavior, perfect links, and a perfect failure detector (for a definition of these terms see Chapter 16).

## 12.2 Non-Functional Requirements

**(N1) Performance.** Achieving high parallel efficiency is the main objective in High Performance Computing (for a definition of the term *parallel efficiency* see Appendix A). Thus, an appropriate substrate must provide low-latency communication that makes economical use of the available bandwidth. To enable efficient use of idle resources, it should ideally deliver changes in group membership instantly.

**(N2) Scalability.** Scalability of a network substrate is determined by the amount of resources consumed per network node as a function of the overall network size (for a formal definition of the term *scalability* see Appendix A). The usual approach to improve scalability is to distribute state and/or responsibility among the nodes. However, this decentralization comes at the price of increased synchronization and coordination overhead and hence most likely reduced performance. This reciprocal effect can be mitigated by dropping certain qualities of the operations provided by the substrate. For example such qualities are message ordering, guaranteed message delivery, or the extent of knowledge about other peers in a peer group. Orbweb should provide mechanisms to tune this trade-off on a per-group basis depending on the requirements of the application.

**(N3) Connectivity.** As opposed to traditional parallel systems, Desktop Grids are typically operated over area networks or the Internet. Thus, connectivity is often limited due to NAT devices and restrictive firewalls. Orbweb should on the one hand ensure universal connectivity by establishing an overlay network that transparently bridges segmentations in the underlying physical network, and on the other hand expose information about the inherent communication costs associated with virtual links to allow for using the physical network most efficiently.

**(N4) Security.** Since large-scale Desktop Grids typically span more than a single administrative domain, a reasonable substrate must undertake measures to keep sensitive data private and to protect the system state from malicious participants. This includes securing communication as well as restricting access to peer groups.

# 13 Overview

ORBWEB belongs to the class of *hybrid* or *hierarchical* P2P networks [Sch01]. In contrast to *pure* P2P, the hybrid approach is characterized by the fact that part of the network functionality is delegated to a comparatively small number of distinguished peers called *superpeers*. In hybrid P2P networks built for data-centric applications, like file sharing, superpeers are typically used as caches for resource indices of connected edge peers. By concentrating knowledge on more powerful peers, query processing times can be reduced as less communication with possibly slow edge peers is necessary. ORBWEB adopts this idea by delegating group membership and virtual topology management to particularly powerful peers. This allows for rapid membership updates (N1[1]), that are essential for achieving good efficiency in P2P Desktop Grid Computing applications, although at the expense of absolute scalability[2] (N2). In this section, we give an overview, how we leveraged where possible (see Chapters 14-16) and amended where necessary (see Chapters 17-20) the XMPP protocol stack and infrastructure to realize ORBWEB around this central idea.

Figure 13.1 shows the protocol and service stack of ORBWEB. We have selected XMPP from the large number of possible communication technologies because the open XMPP standards, depicted in the lower layers of Figure 13.1, already cover two of our four functional requirements: As described in Chapter 15, a peer group abstraction (F1) can be modeled using XMPP *Multi-User Chats* (MUC). The unicast and groupcast communication primitives (F3) are also covered by the functionality provided by MUCs. While the former can be realized using private chats, the latter uses the fact that any occupant of a MUC room can send a message to the room for delivery to all room occupants. As described in Chapter

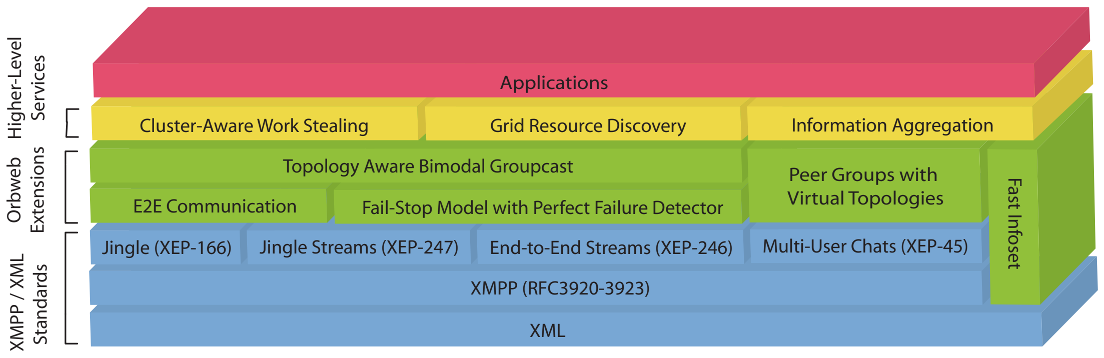


**Figure 13.1:** ORBWEB's protocol and service stack with exemplary higher-level services

1 N*x* and F*x* are references to the requirements defined in Chapter 12.
2 See Appendix A for a formal definition of term *absolute scalability*

14, XMPP already satisfies some of our non-functional requirements as well: It allows for universal connectivity (N3) through relaying by the XMPP server, which is accessible in most network scenarios as communication is client-initiated and an HTTP binding exists to tunnel restrictive server-side firewalls (XEP-124, [xsf]). With various security measures[1] at the protocol level, in particular signing and/or encryption of XMPP stanzas, XMPP also fulfills the security-related requirements (N4).

Despite the fact that XMPP is a good match for our requirements in some areas, the fact remains that XMPP was not explicitly designed for P2P Grid Computing. Hence, there is no clearly defined distributed system model available for XMPP (F4) and virtual topologies (F2) are not supported. Moreover, several of our non-functional requirements are not satisfied: First, XMPP implements no P2P interaction model: Even those messages that could be exchanged directly between two clients are relayed by the XMPP server. Second, the MUC protocol maintains complete membership information at all nodes, resulting in view maintenance costs that grow quadratically with group size. Third, groupcast messages are delivered using replicated unicast, i.e. by having the server send the message explicitly to each group member, resulting in costs that grow linearly with group size. Fourth, XMPP is an XML protocol that is verbose and highly redundant. Thus, XMPP using conventionally encoded XML wastes bandwidth and processing power. These shortcomings result in the server quickly becoming a performance bottleneck, when groups grow large and/or a large number of messages have to be relayed. Thus, ORBWEB would fail to satisfy the non-functional requirements for performance (N1) and scalability (N2) without adaptations.

We addressed these issues by providing a set of extensions to the XMPP protocol itself and the *Openfire*/*Smack* XMPP software stack from Jive Software [jiv] (see the middle layers of Figure 13.1): First, ORBWEB implements a fail-stop distributed system model (F4) on top of XMPP. The definition of the fail-stop model and a discussion of its implementation within ORBWEB can be found in Chapter 16. Furthermore, we modified the XMPP communication subsystem to create direct inter-client connections that can be used for P2P message exchange. As described in Chapter 17, these connections are created based on traffic pattern analysis in a way that respects the limitations of typically resource -constrained peers within Desktop Grids. Thanks to this modification, ORBWEB takes considerable parts of the relay load off the XMPP server and becomes a P2P system. In Chapter 18, we describe how the MUC protocol can be extended to support custom virtual topologies (F2). This allows for example the establishment of tree-structured topologies among the peers of a group, where each peer maintains only a partial and small membership view of configurable size. The resulting maintenance costs are logarithmic with respect to the size of the group. Chapter 19 discusses a probabilistic topology-aware decentralized groupcast implementation with superpeer-side costs that are constant with respect to the size of the group. Finally, we describe in Chapter 20, how *Fast Infoset* (FI) [fi], a binary encoding of XML, can be integrated into the XMPP software stack without sacrificing scalability. As will be substantiated in Chapter 22, these optimizations together significantly improve the performance (N1) and the scalability (N2) of ORBWEB.

1 For a detailed discussion of the security features of XMPP the reader is referred to the standard documents (RFC 3920 [SA04b], RFC 3921 [SA04c], and RFC 3923 [SA04a]).

# 14 XMPP Basics

The *Extensible Messaging and Presence Protocol* (XMPP) is an open, XML-based protocol for real-time communication that has been standardized by the *Internet Engineering Task Force* (IETF). As XMPP is designed to be modular, it can be easily adapted to use cases not covered by the XMPP core specifications published as RFC 3920 [SA04b] and RFC 3921 [SA04c]. Extensions are managed by the *XMPP Software Foundation* [xsf] as publicly available *XMPP Extension Protocols* (XEP). Historically, XMPP – formerly known as *Jabber* – has been used for instant messaging. However, due to its extensibility, the scope of XMPP has grown significantly since then. All kinds of applications based on real-time message exchange including signaling for video conferencing, whiteboarding, collaboration, content syndication, and generalized XML routing have been built on top of XMPP. Today, XMPP-based software is deployed on thousands of servers across the Internet. A large number of client and server implementations written in a variety of languages exist, making XMPP available for almost every platform.

The following section describes the architecture of XMPP. Subsequent sections provide abstract specifications and, where required for reasons of comprehensibility, simplified implementations for the core components of this architecture including the underlying transport, XMPP session management, and Multi-User Chats (MUC). Specifications are given as module definitions as introduced by Cachin *et al.* in [CRG11]. For a condensed summary of their notation and our extensions to it, the reader is referred to Appendix B. Note that error handling code has been omitted from the algorithms for the sake of simplicity.

In the following, *mappings* are denoted using calligraphic letters, e.g. $\mathcal{M}$. $\mathcal{M}(a)$ denotes the value $a$ is mapped to by $\mathcal{M}$. If and only if there is no such mapped value, $\mathcal{M}(a)$ evaluates to $\perp$. The inverse of a mapping is denoted as $\mathcal{M}^{-1}$. $\pi_i(t)$ denotes the projection from the $n$-tuple $t = (t_1, t_2, \ldots, t_{n-1}, t_n)$ to its $i$-th element $t_i$ for $i \in \{1, 2, \ldots, n-1, n\}$.

## 14.1 Network Architecture

As depicted in Figure 14.1, the XMPP network has a decentralized client/server architecture that resembles the email network. Clients and servers are referred to as *entities* in XMPP jargon. Entities are addressed using unique *Jabber Identities* (*JID*). A JID is a triple consisting of an optional *node* name, a DNS server name or IP address called a *domain*, and an optional *resource* identifier. The typical representation of a JID is a string `node@domain/resource`. Server JIDs consist of the domain part only. Every client is connected to a single XMPP server through which it can exchange messages with other clients. When a client establishes a session with a server, a unique JID is negotiated and assigned to the client. These client JIDs are fully qualified, i.e., contain both the optional node and resource parts.

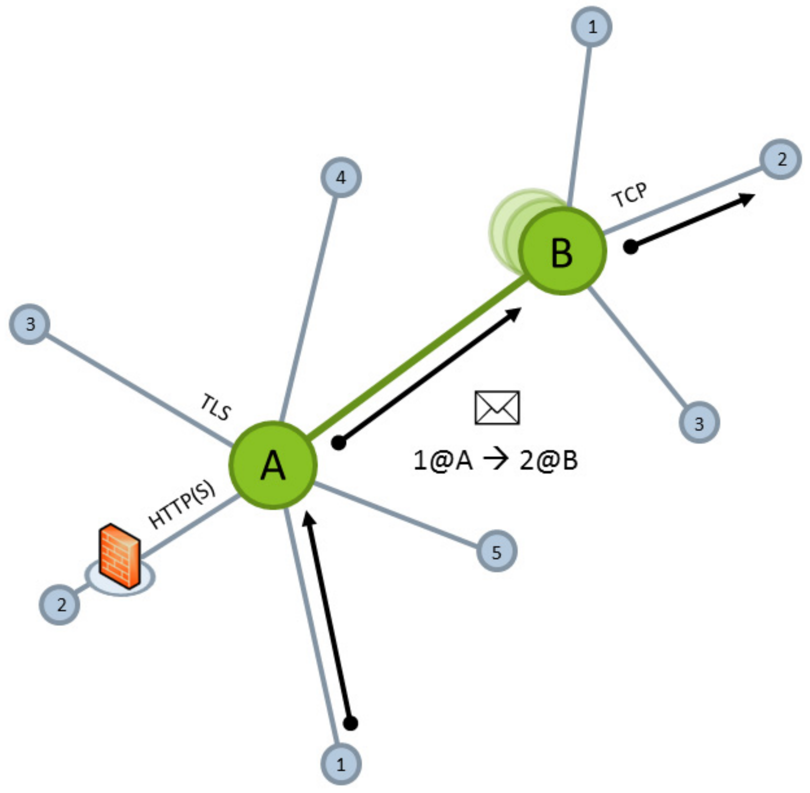


**Figure 14.1:** The XMPP network consists of federated servers (A,B) that route stanzas between clients (1-5).

Entities communicate by exchanging (usually small) XML documents called *stanzas*. There are three main types of stanzas: message, presence, and info/query (IQ). They share a common set of attributes. These are the JID of the sender (`from`), the JID of the recipient (`to`), and the type of the stanza (`type`). A message stanza is used to push data to the remote entity specified by the `to` attribute. Presence stanzas are a basic broadcasting mechanism used to publish availability information to subscribed recipients. IQ messages are used for request/response interaction between entities.

XMPP *servers* are responsible for routing stanzas through the XMPP network. For that purpose a server maintains a routing table that contains a mapping from an entities JID to an associated session. For clients connected to the same server, routing of a message requires only a single lookup in the routing table. If no entry for the target JID exists, the stanza is discarded. Otherwise, the stanza is sent to the target client using the associated session. If an XMPP server receives a message addressed to a recipient connected to another XMPP server, i.e. to a JID with a different domain part, the message is forwarded. If no session for the target server exists, one is established dynamically and stored in the routing table. Servers that cooperate by routing stanzas to each other are called *federated* in XMPP jargon. There is no concept of multi-hop routing between federated XMPP servers, i.e., messages from one client to another are routed in at most three hops. Federation is beyond the scope of this thesis and hence will not be considered in the following.

**Module 14.1** Interface of the XMPP stanza transport module

**Module:**

**Name:** Stanza Transport, **instance** *st*.

**Requests:**

⟨ *st*, Connect | *endpoint* ⟩: Establish a connection to *endpoint*.

⟨ *st*, Send | *endpoint*, *stanza* ⟩: Send the given *stanza* to *endpoint*.

**Indications:**

⟨ *st*, Established | *endpoint* ⟩: A connection has been established to *endpoint*.

⟨ *st*, Closed | *endpoint* ⟩: The connection to *endpoint* has been closed.

⟨ *st*, Deliver | *endpoint*, *stanza* ⟩: A *stanza* has been received from *endpoint*.

**Properties:**

**ST1:** *FIFO delivery:* If some *endpoint* sends a stanza $s_1$ before it sends a stanza $s_2$, then the receiving endpoint delivers $s_1$ before $s_2$.

**ST2:** *Atomic delivery:* A stanza is delivered either completely or not at all.

## 14.2 Stanza Transport

An XMPP client can connect to a server using different protocols. The most widely protocol used for this purpose today is TCP. However, when using TCP is impossible due to restrictive firewalls or proxies, which is often the case in enterprise setups, the client can fall back to an alternative connection method that emulates a long-lived TCP connection using a series of requests and responses that are exchanged over short-lived HTTP(S) connections. XMPP uses a protocol called BOSH defined in XEP-0124 for this purpose.

Module 14.1 defines the *Stanza Transport* interface that abstracts both the ability to connect to a remote endpoint as well as sending and receiving stanzas over the established connection. Implementations emit notifications when a connection is established or closed, and when a stanza is received over a connection. The serialization of stanzas to and the deserialization of stanzas from the transport's payload representation is part of the module implementation. Furthermore, implementations are required to guarantee per-endpoint FIFO delivery (ST1) and that stanzas are delivered atomically, i.e., a stanza is delivered to higher layers of the network stack, if and only if the whole stanza has been received (ST2). In the following, we omit *endpoint* as a parameter to the requests and indications of the Stanza Transport module if its value is unambiguously determined by the context.

## 14.3 Session Management

After an XMPP client has established a stanza transport with the server, an XMPP session is established by negotiating a unique client JID in a process called *resource binding*. The implementations for the respective client- and server-side Modules 14.2 and 14.4 are the Protocols 14.3 and 14.5, respectively. Basically, the negotiation is accomplished using a pair of specialized IQ messages. First, the client sends a request IQ stanza containing a *bind* request (C-SM-1). On receipt of this request, the server creates a unique JID and updates

**Module 14.2** Client-side interface of the XMPP Session Management module

**Module:**

**Name:** Client-side Session Management, **instance** *c-sm* **with** server endpoint *sp*.

**Requests:**

$\langle$ *c-sm*, Init | *sp* $\rangle$: Initiate session with *sp*.

**Protocol 14.3** Client-side XMPP session management protocol

**Implements:**

Client-side Session Management, **instance** *c-sm* **with** server endpoint *sp*.

**Uses:**

Stanza Transport, **instance** *st*.

```
upon event ⟨ c-sm, Init | $sp$ ⟩ do                                          ▷ C-SM-0
    $localJid := \perp$;
    trigger ⟨ st, Connect | $sp$ ⟩;

upon event ⟨ st, Established | $endpoint$ ⟩ do                               ▷ C-SM-1
    trigger ⟨ st, Send | $endpoint$, IQ[type='set'](Bind()) ⟩;

upon event ⟨ st, Deliver | $endpoint$, IQ[type='result'](Bind($jid$)) ⟩ do   ▷ C-SM-2
    $localJid := jid$;
```

**Module 14.4** Server-side interface of the XMPP Session Management module

**Module:**

**Name:** Server-side Session Management, **instance** *s-sm*.

**Indications:**

$\langle$ *s-sm*, Closed | *jid* $\rangle$: The session with the client with the bound *jid* has been closed.

its routing table $\mathcal{R}$[1] with a tuple consisting of that JID and the respective endpoint. The tuple is used to lookup the associated stanza transport for the purpose of routing whenever an incoming stanza with a matching `to` attribute comes in subsequently (S-SM-2). To acknowledge the successful binding process, the server responds with a response IQ stanza containing the bound JID (S-SM-1). The client stores the JID as it has to be included as the `from` attribute in any stanza that is sent to the server subsequently. Although not reflected in the protocol descriptions herein, clients can also propose a JID to the server that is refused, however, in case a binding for that JID already exists.

When a session is closed, either deliberately by the client or due to a broken transport connection, the server removes the associated entry from the routing table (S-SM-3).

1 According to the definitions given in the introduction to this chapter, $\mathcal{R}(id)$ gives the *endpoint* for the client with identifier $id$ or $\perp$ if and only if no such entry exists. $\mathcal{R}^{-1}$ is the inverse of $\mathcal{R}$, i.e. $\mathcal{R}^{-1}(endpoint)$ gives the client identifier associated with the given $endpoint$ or $\perp$ if and only if no matching entry exists.

**Protocol 14.5** Server-side XMPP session management protocol.

```
Implements:
    Server-side Session Management, instance s-sm.

Uses:
    Stanza Transport, instance st.

upon event ⟨ s-sm, Init ⟩ do                                                    ▷ S-SM-0
    $\mathcal{R} := \varnothing$;

upon event ⟨ st, Deliver | endpoint, IQ[type='set'](Bind()) ⟩ do               ▷ S-SM-1
    $boundJid := \textsc{CreateUniqueJid}()$;
    $\mathcal{R} := \mathcal{R} \cup \{(boundJid, endpoint)\}$;
    trigger ⟨ st, Send | endpoint, IQ[type='result'](Bind(boundJid)) ⟩;

upon event ⟨ st, Deliver | endpoint, s :=Stanza[to](data) ⟩ do                 ▷ S-SM-2
    $target := \mathcal{R}(to)$;
    if $target \neq \bot$ then
        trigger ⟨ st, Send | target, s ⟩;

upon event ⟨ st, Closed | endpoint ⟩ do                                         ▷ S-SM-3
    $jid := \mathcal{R}^{-1}(endpoint)$;
    if $jid \neq \bot$ then
        $\mathcal{R} := \mathcal{R} \setminus \{r \in \mathcal{R} | \pi_2(r) = endpoint\}$;
        trigger ⟨ s-sm, Closed | jid ⟩;
```

## 14.4 Multi-User Chats

*Multi-User Chats* (MUC), defined in XEP-0045 [SA13], extend the core XMPP protocol to enable clients to discover each other and to communicate in a one-to-many fashion. The scope of a MUC is a *room*. A *room* is a named dynamic set of JIDs, called its *occupants*, maintained by the XMPP server. The (simplified[1]) client- and server-side MUC interface specifications are given in Module 14.6 and 14.8, the associated implementations are Protocol 14.7 and 14.9.

An XMPP client can enter (C-MUC-1) and leave (C-MUC-2) any number of MUC rooms by sending an appropriate presence stanza to the room. Changes in room membership are tracked by the server using an occupant map $\mathcal{O}_S$. An entry in $\mathcal{O}_S$ is a tuple $(boundJid, occupantJid)$, where *boundJid* is the identifier bound to the client as a result of the resource binding process described in Section 14.3 and *occupantJid* is the occupant JID triple generated and stored as $c\text{-}muc.self$ by the client (C-MUC-1). After $\mathcal{O}_S$ has been updated, the server propagates the update to all occupants (including the new occupant) by sending presence stanzas using replicated unicast (S-MUC-1). Additionally, a presence

1 Note, that XEP-0045 defines a bunch of additional features like different roles and privileges within a room, the ability to ban users, and dynamically changing the nickname. These features are not relevant here and are not exposed by the ORBWEB API.

**Module 14.6** Interface of the Client-side Multi-User Chat module

**Module:**

**Name:** Client-side Multi-User Chat, **instance** *c-muc* **with** MUC *roomname*.

**Requests:**

⟨ *c-muc*, Enter | *nickname* ⟩: Enter the room using the given *nickname*.

⟨ *c-muc*, Leave ⟩: Leave the room.

⟨ *c-muc*, Send | *target*, *message* ⟩: Send the given *message* to the occupant identified by the *target* JID or to all occupants of the room when *target* is the room's JID.

**Indications:**

⟨ *c-muc*, OccupantStateChanged | *occupantJid*, *state* ⟩: An occupant identified by *occupantJid* has entered ($state = \textsc{Joined}$) or left ($state = \textsc{Left}$) the room.

⟨ *c-muc*, Deliver | *source*, *message* ⟩: Deliver the given *message* received from *source* (occupant or room JID).

stanza is sent to the new occupant for each occupant already in the room. Although not explicitly stated in the algorithm pseudo code and not required by the XMPP specification, $\mathcal{O}_S$ is traversed such that the presence update for the triggering occupant is emitted last in our implementation. On receipt of each of these presence stanzas, occupants update their occupants set $\mathcal{O}_C$ accordingly and signal the event to modules on higher layers (C-MUC-4). The process of a client entering or leaving a room is terminated by the reception of a *presence* stanza that signals the state change for the client itself. When the server-side Session Management module indicates that a session has been closed, the server handles this event as if the respective client had left the room explicitly (S-MUC-1).

Occupants of a MUC room can send message stanzas addressed to a single occupant (private chat) or to the whole room (C-MUC-3). The former are just relayed by the XMPP server to the recipient, the latter are send to all occupants (including the sender) by the hosting XMPP server using a replicated unicast (S-MUC-2). The two cases can be discriminated by looking at the `to` attribute of the incoming message: if the resource part is missing, i.e., $\pi_3(to) = \perp$, a message is targeted to the whole room. In case one of the parties is not an occupant of the room (any more), the message is dropped. When the message arrives on the client, it is delivered to modules on higher layers (C-MUC-5).

While each MUC room is hosted by a single XMPP server, clients connected to federated servers may participate as well. However, MUCs in federated setups have not been considered in this thesis for the sake of simplicity.

**Protocol 14.7** Client-side XMPP Multi-User Chat protocol.

```
Implements:
    Client-side Multi-User Chat, instance c-muc with roomname.

Uses:
    Stanza Transport, instance st.
    Client-side Session Management, instance c-sm.

upon event ⟨ c-muc, Init | $roomname$ ⟩ do                                   ▷ C-MUC-0
    $state$ := NOTJOINED;
    $self$ := $\bot$;
    $\mathcal{O}_C = \varnothing$;

upon event ⟨ c-muc, Enter | $nickname$ ⟩ such that $state \neq$ JOINED do    ▷ C-MUC-1
    $self := \{roomname, \pi_2(c\text{-}sm.localJid), nickname\}$;
    trigger ⟨ st, Send | Presence[from=$c$-$sm.localJid$, to=$self$, type='available'] ⟩;

upon event ⟨ c-muc, Leave ⟩ such that $state$ = JOINED do                   ▷ C-MUC-2
    trigger ⟨ st, Send | Presence[from=$c$-$sm.localJid$, to=$self$, type='unavailable'] ⟩;

upon event ⟨ c-muc, Send | $target$, $data$ ⟩ such that $state$ = JOINED do  ▷ C-MUC-3
    trigger ⟨ st, Send | Message[from=$c$-$sm.localJid$, to=$target$]($data$) ⟩;

upon event ⟨ st, Deliver | Presence[$from$, to=$c$-$sm.localJid$, $type$] ⟩ do   ▷ C-MUC-4
    if $type$ ='available' then
        $\mathcal{O}_C = \mathcal{O}_C \cup \{from\}$;
        if $from = self$ then
            $state$ := JOINED;
        trigger ⟨ c-muc, OccupantStateChanged | $from$, JOINED ⟩;
    else
        $\mathcal{O}_C = \mathcal{O}_C \setminus \{clientJid\}$;
        if $from = self$ then
            $state$ := LEFT;
        trigger ⟨ c-muc, OccupantStateChanged | $from$, LEFT ⟩;

upon event ⟨ st, Deliver | $m$ := Message[$from$, to=$c$-$sm.localJid$] ⟩ such that $state$ = JOINED
do                                                                           ▷ C-MUC-5
    if $from \in \mathcal{O}_C$ then
        trigger ⟨ c-muc, Deliver | $from$, $m$ ⟩;
```

**Module 14.8** Interface of the Server-side Multi-User Chat module

```
Module:
    Name: Server-side Multi-User Chat, instance s-muc with MUC roomname.
```

**Protocol 14.9** Server-side XMPP Multi-User Chat protocol.

```
Implements:
    Server-side Multi-User Chat, instance s-muc with roomname.

Uses:
    Stanza Transport, instance st.
    Server-side Session Management, instance s-sm.

upon event ⟨ muc, Init | $roomname$ ⟩ do                                    ▷ S-MUC-0
    $\mathcal{O}_S := \varnothing$;

upon event ⟨ st, Deliver | Presence[$from$, $to$, $type$] ⟩ such that $\pi_1(to) = roomname$ or upon
event ⟨ st, Closed | $from$ ⟩ do                                           ▷ S-MUC-1
    $occupantJid := \{roomname, \pi_2(from), nickname\}$;
    if $type$ = 'available' then
        $\mathcal{O}_S := \mathcal{O}_S \cup \{from, occupantJid\}$;
    else
        $\mathcal{O}_S := \mathcal{O}_S \setminus \{from, occupantJid\}$;
    forall $o \in \mathcal{O}_S$ do
        trigger ⟨ st, Send | s-sm.$\mathcal{R}(\pi_1(o))$, Presence[from=$occupantJid$, to=$\pi_1(o)$, $type$] ⟩;
        trigger ⟨ st, Send | s-sm.$\mathcal{R}(from)$, Presence[from=$\pi_2(o)$, to=$from$, $type$] ⟩;

upon event ⟨ st, Deliver | endpoint, Message[$from$, $to$]($data$) ⟩ such that $\pi_1(to) = roomname$
do                                                                         ▷ S-MUC-2
    if $\mathcal{O}_S(from) \neq \perp$ then
        if $\pi_3(to) \neq \perp$ then
            $recipientJid := \mathcal{O}_S^{-1}(to)$
            if $recipientJid \neq \perp$ then
                trigger ⟨ st, Send | s-sm.$\mathcal{R}(recipientJid)$, Message[from=$\mathcal{O}_S(from)$,
to=$recipientJid$]($data$) ⟩;
        else
            forall $o \in \mathcal{O}_S$ do
                trigger ⟨ st, Send | s-sm.$\mathcal{R}(\pi_1(o))$, Message[from=$\mathcal{O}_S(from)$, to=$\pi_1(o)$]($data$) ⟩;
```

# 15 Peer Groups

This chapter describes the concept of *Peer Groups* and how they are realized on top of the XMPP Client-side Multi-User Chat module introduced in the previous chapter.

The idea of peer groups is not new but is known for decades under the original name *Process Groups* [CZ85, BJ87]. Process groups are sets of local or distributed processes grouped to cooperatively provide a service. In the latter case one speaks of *Distributed Process Groups*. The most important reasons for grouping processes are according to Liang *et al.* [LCN90]:

1. To abstract the common characteristics of processes.
2. To encapsulate internal state and hiding the interactions between group members from the external world.
3. To be able to compose higher-level system services by using groups as building blocks.

As process groups are considered very useful, they (or similar concepts) are also provided by some of the existing substrates for Peer-to-Peer applications discussed in Chapter 23.

A *Process Group Model* describes the characteristics of a process group. These characteristics are manifold: Birman *et al.* [Bir93] differentiate *anonymous* and *explicit* group models. While for the latter the members of the group are known to each other, this is not true for the former. Liang *et al.* [LCN90] distinguish between *deterministic* group models with built-in strong consistency and *non-deterministic* group models with relaxed consistency constraints but lower maintenance overhead. Another differentiator proposed by the same authors is whether process groups are *open* or *closed*, i.e., whether requests to the group may be issued from outside the group or not.

The ORBWEB group model has been designed to be very flexible and can be specified on a per group basis. ORBWEB groups can be anonymous or explicit and open or closed. They provide consistent membership lists and are thus deterministic according to the definition of Liang *et al.* All these properties are strongly related to the aspect of visibility between peers, which is covered by the concept of virtual topologies in ORBWEB. Virtual topologies and how they are used to model visibility within peer groups is described in detail in Chapter 18. For the rest of this chapter, we assume the special case of an explicit process group model in which each peer is visible to all other peers.

## 15.1 Peer Group Management

Module 15.1 shows the interface of ORBWEB's *Peer Group Management* (PGM) module. An ORBWEB peer can create any number of peer groups. Once created, a peer group can

**Module 15.1** Interface of the Peer Group Management module

**Module:**

**Name:** Peer Group Management, **instance** *pgm* with group *name*.

**Requests:**

⟨ *pgm*, Init | *name* ⟩: Create a group with the given *name*.

⟨ *pgm*, Join ⟩: Join the group.

⟨ *pgm*, Leave ⟩: Leave the group.

⟨ *pgm*, View | *members* ⟩: Returns all group *members* that are in the local peer's view.

⟨ *pgm*, Unicast | *target*, *message* ⟩: Send the given *message* to the group member identified by *target*.

⟨ *pgm*, Groupcast | *message* ⟩: Send the given *message* to all members of the group.

**Indications:**

⟨ *pgm*, Joined | *peerId* ⟩: The local peer has joined the group successfully and has been assigned the peer identifier *peerId*.

⟨ *pgm*, Left ⟩: The local peer has left the group.

⟨ *pgm*, ViewUpdate | *peerId*, *state* ⟩: A peer identified by *peerId* has joined ($state = \text{JOINED}$) or left ($state = \text{LEFT}$) the group.

⟨ *pgm*, Deliver | *source*, *message* ⟩: Deliver the given *message* received from the peer with identifier *source*.

**Properties:**

**PGM1:** *Unique Identities:* The peer identifier assigned to a peer is unique for every invocation of the join request.

**PGM2:** *Fail-Stop Behavior:* The abstractions provided by the module support the fail-stop distributed system model.

**PGM3:** *Atomic Delivery:* Messages sent using the ⟨ Unicast ⟩ and ⟨ Groupcast ⟩ requests are delivered either completely or not at all.

be joined and once joined left again at any time using the ⟨ Join ⟩ and ⟨ Leave ⟩ requests. When the process of joining a group is completed, modules on higher layers are notified via a ⟨ Joined ⟩ indication. Analogously, a ⟨ Left ⟩ indication is emitted when a peer has left a group. Whenever a remote peer joins or leaves the peer group, this event is reported to all other peers by means of a ⟨ ViewUpdate ⟩ indication. The local membership view can be requested using the ⟨ View ⟩ request. Joined peers are able to send messages to individual co-members using the ⟨ Unicast ⟩ or to the entire group using the ⟨ Groupcast ⟩ request. The exact semantics of the latter is discussed in detail in Chapter 19. The ⟨ Deliver ⟩ indication is triggered whenever a message is to be delivered to higher layers.

Implementations of the Peer Group Management module are required to ensure that each invocation of the ⟨ Join ⟩ request – even for those issued by the same peer – produces a unique peer identifier (PGM1). Moreover, implementations must make sure that the abstractions implicitly provided by the requests and indications of this module, i.e. communication channels, processes, and failure detector, support the Fail-Stop distributed system model (PGM2). Due to the high complexity, the exact definition of the underlying requirements and the proof that the implementation described in the following section satisfies these requirements are given in Chapter 16.

## 15.2 XMPP MUC-based Implementation

Orbweb's implementation of the PGM module is based on XMPP Multi-User Chats. The algorithm sketch is shown in Protocol 15.2. Basically, it is a thin wrapper around the Client-Side MUC module (see Module 14.6) that adds logic to satisfy the requirement for unique identities (PGM1) and to translate between occupant JIDs and peer identifiers. The superpeer is only indirectly involved by means of the Server-Side MUC module in this PGM implementation.

The uniqueness of peer identifiers (PGM1) is ensured by creating a unique identifier prefix that is combined with a local *incarnation* counter. While the latter is incremented during the processing of ⟨ Leave ⟩ requests by PGM-2, the former is generated by the GenerateUnique function in PGM-0 from the *media access control* (MAC) address of the host and the current value of the local hardware clock[1]. To see that the resulting identifiers are unique, we look at all possible ways a peer can leave a group:

1. Intentionally by explicitly issuing a ⟨ Leave ⟩ request, which immediately increases the incarnation number (PGM-2).
2. Due to the underlying XMPP session being closed, which triggers a ⟨ Leave ⟩ request (PGM-7) and subsequently an increase of the incarnation number (see 1).
3. Resulting from a crash of the hosting process, which leads to a reinitialization of the unique identifier prefix when the same group is rejoined later (PGM-0).

In all three cases the peer identifier assigned to the rejoining peer is different from all identifiers that have been created before (with respect to the local hardware clock). Hence, the *Unique Identities* property (PGM1) is satisfied.

Note that the implementation inherits the *Atomic Delivery* guarantee (PGM3) from the *Session Transport* module.

1 We assume that the MAC addresses of all participating hosts are universally administered and hence are universally unique and that the local hardware clock is never set to an earlier time.

**Protocol 15.2** XMPP MUC-based Peer Group Management protocol

```
Implements:
    Peer Group Management, instance pgm with group name.

Uses:
    Client-side Multi-User Chat, instance c-muc.

upon event ⟨ pgm, Init | $name$ ⟩ do                                                    ▷ PGM-0
    $unique := \textsc{GenerateUnique}()$;
    $incarnation := 0$;
    $localPeerId := \perp$;
    $view := \emptyset$;

upon event ⟨ pgm, Join ⟩ such that c-muc.$state \neq \textsc{Joined}$ do                ▷ PGM-1
    $localPeerId := unique$+'-'+$incarnation$;
    trigger ⟨ c-muc, Enter | $localPeerId$ ⟩;
    wait for event ⟨ pgm, Joined | $localPeerId$ ⟩;

upon event ⟨ pgm, Leave ⟩ such that c-muc.$state = \textsc{Joined}$ do                 ▷ PGM-2
    $incarnation := incarnation + 1$;
    trigger ⟨ c-muc, Leave ⟩;
    wait for event ⟨ pgm, Left ⟩;

upon event ⟨ c-muc, OccupantStateChanged | $from$, $state$ ⟩ do                         ▷ PGM-3
    $peerId := \pi_2(from)$;
    trigger ⟨ pgm, ViewUpdate | $peerId$, $state$ ⟩;
    if $state = \textsc{Joined}$ then
        $view := view \cup peerId$;
        if $peerId = localPeerId$ then
            trigger ⟨ pgm, Joined | $peerId$ ⟩;
    else
        $view := view \setminus peerId$;
        if $peerId = localPeerId$ then
            trigger ⟨ pgm, Left ⟩;

upon event ⟨ pgm, Unicast | $target$, $msg$ ⟩ such that c-muc.$state = \textsc{Joined}$ do   ▷ PGM-4
    $recipientJid := (name, \pi_2(\text{c-muc}.self), target)$;
    trigger ⟨ c-muc, Send | $recipientJid$, $msg$ ⟩;

upon event ⟨ pgm, Groupcast | $msg$ ⟩ such that c-muc.$state = \textsc{Joined}$ do      ▷ PGM-5
    $groupJid := (name, \pi_2(\text{c-muc}.self), \perp)$;
    trigger ⟨ c-muc, Send | $groupJid$, $msg$ ⟩;

upon event ⟨ c-muc, Deliver | $source$, $msg$ ⟩ such that c-muc.$state = \textsc{Joined}$ do  ▷ PGM-6
    $peerId := \pi_3(source)$;
    trigger ⟨ pgm, Deliver | $peerId$, $msg$ ⟩;

upon event ⟨ c-muc.st, Closed | $endpoint$ ⟩ do                                          ▷ PGM-7
    trigger ⟨ pgm, Leave ⟩;

upon event ⟨ pgm, View | $members$ ⟩ do                                                  ▷ PGM-8
    $members := view$;
```

# 16 Distributed System Model

A *distributed system model* (DSM) describes the behavior of a distributed system in terms of three basic abstractions: processes, communication links, and time. A DSM determines which algorithms can be used to implement a distributed programming abstraction like a totally ordered multicast channel, an agreement primitive, or group membership. In the following, we describe the DSM that is exposed by ORBWEB to the higher layers of the COHESION middleware stack and to applications.

## 16.1 Basic Abstractions

A *process* is an entity that is able to perform computations. A process abstraction describes what faults may interrupt the flow of computation and communication steps that make up a process and how the process may behave after a fault occurred. The most relevant process abstractions are *crash-stop*, *crash-recovery*, and *arbitrary-fault* (also known as *byzantine*). As the name implies crash-stop processes simply stop executing steps on failure and never resume. Crash-recovery processes may crash and stop communicating but might recover at a later point in time. Typically, algorithms designed for crash-recovery processes use persistent storage to compensate for the loss of state when they crash. Arbitrary-fault processes may behave arbitrarily. Typical reasons for this kind of behavior are malicious attackers and implementation bugs.

Processes may interact by exchanging messages over *communication links* or *links* for short. Link abstractions are typically characterized using three aspects: How often a message is delivered[1] that has been sent multiple times? How often a message may be duplicated? And whether the network may corrupt or create messages? While there are numerous possible link abstractions, many are of theoretical interest only. The most important of practical importance are *fair-loss* and *perfect links*. The former essentially guarantees that a message that is sent infinitely often is delivered infinitely often[2], messages are duplicated only a finite number of times, and messages are neither corrupted nor created by the network. The properties of perfect links are detailed below. The practical importance of these link abstractions stems from the fact that UDP is an implementation of the fair-loss and TCP is an approximation[3] of the perfect link abstraction [CGR11].

1 The term *delivered* means (1) received by the target host and (2) handed over to the target application.

2 In other words fair-loss links deliver messages with non-zero probability.

3 In asynchronous systems, TCP is just an approximation of the perfect link abstraction. While TCP includes acknowledgments and retransmission mechanisms to recover from omissions, TCP breaks the connection and fails to deliver messages, if the other endpoint is unresponsive for an extended period of time, erroneously assuming that the corresponding node has crashed [CGR11].

Both processes and links have properties related to time: Processes proceed at a specific (relative) speed and links have message delivery delays referred to as *latencies*. Time-related models describe whether bounds can be given for these properties. One distinguishes *synchronous* and *asynchronous* systems. An asynchronous system does not allow making timing assumptions at all: There are no known bounds on processing and communication delays. For synchronous systems the opposite is true: There *are* known upper bounds on both types of delays. In between synchronous and asynchronous systems are *partially synchronous* systems in which one can expect periods of synchronous behavior of arbitrary length. Because modeling algorithms based on explicit timing assumptions is cumbersome, *failure detectors* have been introduced to abstract from concrete timing assumptions. A failure detector classifies each processor as either *failed* or *correct*. The corresponding failure detectors for synchronous and partially synchronous systems are the *perfect* and the *eventually perfect failure detector*, respectively. Informally spoken, the latter is allowed to change its mind about the status of a process for a finite number of times, while the former is not.

A combination of these three basic abstractions results in a specific distributed system model. For example the *fail-recovery* distributed system model is based on crash-recovery processes, stubborn links[1], and an eventually perfect failure detector. In general, combinations consisting of abstractions giving stronger guarantees create more powerful distributed system models in which algorithms may be implemented more easily. Thus, fail-stop as the most powerful distributed system model substantially simplifies the implementation of distributed programming abstractions.

Due to reasons of space, we were able to give a brief overview on the subject only. For a comprehensive treatment of the subject, the reader is referred to the excellent text books on distributed systems and algorithms by Lynch [Lyn96] and Cachin *et al.* [CRG11].

## 16.2 Orbweb's Fail-Stop DSM

Orbweb operates on top of a distributed system. We assume that this system is partially synchronous. Orbweb hides this fact and provides a more powerful distributed system model shielding the Cohesion application programmer from handling the complexity by themselves. As required by (F4), Orbweb provides a *fail-stop* distributed system model. This simplifies the implementation of many distributed algorithms [CRG11]. The fail-stop model is characterized by three properties:

1. Processes execute correctly but may crash at some point in time. After a process has crashed it never recovers. This property is called *crash-stop* behavior.

2. Processes communicate using *perfect links*. A perfect link reliably delivers every message sent by a correct process to a correct process exactly once. This breaks down to three properties: First, every message sent by a correct process $p$ to a correct process $q$ is eventually delivered by $q$ (*reliable delivery*). Second, no message is

1 *Stubborn links use retransmission to compensate for message losses.*

delivered more than once (*no duplication*). Third, only those messages are delivered on $q$ that have been sent by some process $p$ to $q$ (*no creation*).

3. Every process has access to a *perfect failure detector* [CT96]. Such a failure detector eventually detects all crashed processes and never outputs false positives, i.e., never reports a correct process to have failed. These properties are called *strong completeness* and (perpetual) *strong accuracy*, respectively.

In the following, we give an informal proof that the DSM exposed by the Orbweb PGM described in Chapter 15 satisfies these requirements.

## 16.3 Perfect Failure Detection

Orbweb provides a *Perfect Failure Detector* (PFD) despite the assumption of a partially synchronous system. This seemingly contradiction to the well-known fact that only eventually perfect failure detectors can be implemented in such systems [CRG11] is resolved by the fact that our definition of a *crashed process* is different from the canonical one, which says that a crashed process in the crash-stop model does not execute any local computation and does not exchange messages with other processes. In contrast, our definition allows further execution of local computations but does not allow a peer to communicate with any other process once it has been detected as crashed. The rationale behind this definition is that if the results of a local computation step never become visible to the rest of the system, there execution is actually meaningless. As processes communicate only indirectly over the superpeer our definition of *crashed* is as follows:

**Definition 16.3.1.** *A peer is called* crashed[1] *if and only if the XMPP session to the* Orbweb *superpeer has been closed. Otherwise the peer is called* correct*.*

As mentioned before, a perfect failure detector must satisfy two properties:

1. Eventually every crashed process is permanently detected as crashed by every correct process (*Strong Completeness*).

2. If a process $p$ is detected by any other process $q$, then $p$ is crashed (*Strong Accuracy*).

Using definition 16.3.1, we have to prove the following three lemmas:

**Lemma 16.3.1.** *A crashed* Orbweb *peer group member* $p$ *is eventually detected by all correct* Orbweb *peers within that peer group* $g$*.*

1 Note that we restrict ourselves to crashes and ignore deliberate departures triggered by means of a ⟨ *pgm*, Leave ⟩ request. Since messages sent from or to a peer that is not a member of a group are silently discarded in S-MUC-2 and C-MUC-5, a peer that has deliberately left a group is from the perspective of other peers by all means equivalent to a crashed one.

*Proof.* We assume that a correct peer $q$ that occupies $g$ does not detect $p$. By definition $q$ must have an open XMPP session with the superpeer. Hence, it is part of the occupant map $\mathcal{O}_S$. As $p$ has crashed the XMPP session of $p$ with the superpeer has been closed by definition. As of S-MUC-1 (see Protocol 14.9) the XMPP server has eventually sent a presence stanza to all occupants of the MUC room including $q$. This triggers an ⟨ *c-muc*, OccupantStateChanged ⟩ indication that in turn triggers a ⟨ *pgm*, ViewUpdate | p.c-muc.$localPeerId$, LEFT ⟩ indication on every client in particular on $q$ which contradicts the assumption. ■

**Lemma 16.3.2.** *Once detected, a crashed* ORBWEB *peer $p$ is detected permanently by a correct peer $q$.*

*Proof.* $p$ has been detected by $q$. Hence, it has been removed from the view of $q$. $p$ can reappear in the view of $q$ for two reasons only: Either the same or another peer joins with the same peer identifier or a presence message indicating that $p$ has joined the group arrives out of order. The former can not happen since peer identifiers are unique as explained in Section 15.2. The latter is impossible, since messages from and to the same client are delivered in FIFO order (guaranteed by the ST1 property of the Stanza Transport module) and are processed by the protocols in FIFO order (see Appendix B).

■

**Lemma 16.3.3.** *If a peer $p$ is detected by any other peer $q$, then $p$ has crashed.*

*Proof.* If $p$ is detected by $q$, then $q$ has triggered a ⟨ *pgm*, ViewUpdate | p.c-muc.$localPeerId$, LEFT ⟩ indication. This must have been triggered by a presence stanza sent during the execution of S-MUC-1. S-MUC-1, however, is executed if and only if the XMPP session between the server and $p$ is closed. In this case, $p$ has crashed by definition. ■

**Theorem 16.3.1.** ORBWEB *process groups realize a perfect failure detector.*

*Proof.* Follows directly from Lemmas 16.3.1 through 16.3.3. ■

### 16.3.1 TCP and the XMPP Session Stanza Module

Definition 16.3.1 equates a peer's correctness with the existence of an XMPP session between the underlying XMPP client and the XMPP server hosted on the ORBWEB superpeer. Until now the exact conditions under which a ⟨ *st*, Close ⟩ indicator is emitted by the *Session Stanza* module has not been discussed. Obviously, these conditions are related to the underlying transport protocol. ORBWEB uses TCP (*Transmission Control Protocol*) [Pos81].

As TCP was designed for resiliency and efficiency, intermittent problems (including process, host, and router crashes, unplugged network cables, etc.) on the network path in between and at the endpoints are not detected until an endpoint tries to send data. If the TCP packets are not acknowledged in a timely fashion, the connection is considered broken and closed on the sender side. However, as no more data can be transmitted, the receiving side cannot be notified of this event and will block on the receive operation forever. This scenario is called *half-open connections*. The problem is that when an ORBWEB peer crashes in the classical sense, the superpeer is not notified of this fact and the server-side *Session Stanza* module does not immediately trigger a $\langle$ *st*, Close $\rangle$ indication. Consequently, the crashed peer would stay in all previously joined rooms and would never be detected by the PFD.

TCP has an optional feature called *keep-alives* that in theory can be used to detect half-open connections without modifying higher-level protocols. Unfortunately, this approach has drawbacks that render its use impractical: First, the feature is optional and thus is not necessarily implemented in all network stacks. Second, the system-wide parameters are usually set to values much too high (e.g., two hours for Microsoft operating systems) for most applications. Setting them to lower values is not only considered bad practice but also requires administrative privileges which cannot be assumed in a Desktop Grid scenario. Hence, ORBWEB implements the keep-alive approach on the XMPP layer by forcibly sending a single whitespace character whenever no XMPP stanza has been sent for a configurable time period. Whitespace between XML tags is discarded when stanzas are parsed from the incoming data stream. If the corresponding TCP packet is not acknowledged for some reason, the connection is closed becoming half-open. The approach is applied symmetrically on both the peer and the superpeer, such that a half-open connection becomes fully-closed eventually. A fully-closed connection, however, triggers automatic cancellation of stale membership in all groups and delivery of resultant view updates to the remaining members.

## 16.4 Perfect Links

The PGM module allows for unicasting messages between peers within the same peer group. The virtual link between two such peers $p$, $q$ is a two hop link on the transport layer ($p \rightarrow s \rightarrow q$, where $s$ is the superpeer). These virtual links are implementations of the *Perfect Link* abstraction. A perfect link satisfies the following properties:

1. Every message sent by a correct process p to a correct process $q$ is eventually delivered by $q$ (*reliable delivery*).

2. No message is delivered more than once (*no duplication*).

3. Only those messages are delivered on $q$ that were sent by some process $p$ to $q$ (*no creation*).

Using the nomenclature of ORBWEB , this translates to the following lemmas (note that we talk about messages at the PGM level, not at the XMPP stanza level).

**Lemma 16.4.1.** *Every message sent (unicast) by a correct peer $p$ to a correct peer $q$, where $p$ and $q$ are members of the same peer group $g$, is eventually delivered by $q$.*

*Proof.* As $p$ is correct an XMPP session exists. Thus, the message eventually arrives at the superpeer in S-MUC-2. Since $q$ is correct it also has an XMPP session with the server. As both $p$ and $q$ are members of $g$, they are represented in $\mathcal{O}_S$. Hence, the message is not dropped by the validity checks in S-MUC-2 and is sent to $q$ and delivered to the PGM module in C-MUC-5 and finally in PGM-6 to the application. If the XMPP session of $p$ or $q$ is closed while the message is in transit than $p$ or $q$, respectively, are not correct and hence the message has not to be delivered by $q$. ■

**Lemma 16.4.2.** *No message is delivered more than once on the same peer $p$.*

*Proof.* This follows from the fact that neither TCP as the underlying transport protocol nor the ORBWEB superpeer duplicates messages. For TCP this is guaranteed by the use of sequence numbers. For the superpeer this is obvious from the protocol. ∎

**Lemma 16.4.3.** *Only those messages are delivered on $q$ that have been sent to $q$ by some member $p$ of $g$.*

*Proof.* It is obvious from the ORBWEB protocol definitions that no messages are created neither by the superpeer nor by any peer outside PGM-4. TCP itself satisfies the *no creation* property. Furthermore, messages from XMPP clients that are not occupants of the underlying MUC room are dropped in C-MUC-5. Hence, every delivered message must originate from some member of $g$ executing PGM-4 as a result of a ⟨ *pgm*, Unicast ⟩ request. The proof that the message is actually delivered at the peer it was targeted to is straightforward but lengthy and is hence omitted for reasons of brevity. ∎

While not strictly required for the fail-stop property, ORBWEB links actually are *FIFO-Order Perfect Links.*

**Lemma 16.4.4.** *If some peer sends a message $m_1$ before it sends $m_2$, then no correct peer delivers $m_2$ unless it has already delivered $m_1$.*

*Proof.* This follows immediately from the FIFO guarantee of TCP and the FIFO processing order of the ORBWEB peers and superpeers. ∎

**Theorem 16.4.1.** *PGM unicast messaging implements* FIFO-Order Perfect Links*.*

*Proof.* Follows immediately from Lemmas 16.4.1 through 16.4.4 ∎

## 16.5 Crash-Stop Behavior

While we gave a definition of the term *crashed* in Section 16.3, the above definition of crash-stop behavior is incomplete with respect to the meaning of the term *recover*. Recovery of a crashed process in general means that the process becomes operational again and restarts to execute the algorithm it is expected to using state persisted on stable storage before the crash occurred. ORBWEB allows physical peers to recover. However, XMPP resource binding (see Section 14.3) that takes place during session establishment ensures that a returning physical peer gets assigned a new unique identity. This mechanism ensures that a peer logically never recovers[1] and that messages targeted to the crashed logical peer are not received and hence not delivered by the new incarnation of the physical peer. A consequence of our definition of a *crashed* peer is that the lifecycle of a peer is bound to the lifecycle of the XMPP session. Hence, an ORBWEB peer is forced to reinitialize in case the XMPP session is closed.

1 ORBWEB peers must adhere to the convention to not use any persisted state that may interfere with the semantics of the provided DSM.

## 16.6 Impact of Network Partitioning

A *network partition* is the state of a network in which all paths between any two non-empty groups of network nodes fail simultaneously. Dealing with network partitions in distributed systems with shared state is non trivial as either consistency or availability has to be sacrificed temporarily until the cause of the network fault has been eliminated. This fact has been formalized by Brewer in his *CAP Theorem* as a conjecture, which has been proven later by Gilbert and Lynch [GL02].

In ORBWEB , network partitions translate to peer crashes. If a network partition occurs, the ORBWEB superpeer is located in exactly one partition. All XMPP sessions to peers outside this *primary partition* are closed as soon as the underlying TCP connections are closed. As a result, the peers are detected as crashed and removed from all occupied peer groups eventually. They may of course rejoin the ORBWEB network with a new logical identity as soon as the network fault has gone.

# 17 Efficient P2P Interaction

Orbweb superpeers act as relays for exchanging messages between peers. This guarantees universal connectivity even for hosts that are behind restrictive firewalls or NAT devices. However, the indirection over an intermediary server limits scalability and performance unnecessarily, when peers *are* able to communicate directly. Hence, a modification of the XMPP message delivery subsystem that enables P2P communication promises to increase the overall system-wide message throughput by eliminating the performance bottlenecks induced by XMPP servers and to decrease message latency by reducing the number of necessary hops from two to one. We call this feature *End-to-End (E2E) communication*, as peers at the ends of the mediated XMPP connection interact directly. Note that E2E communication is an optimization and complements superpeer-relayed communication that is always available as a fallback when E2E session establishment is not possible due to, for example, NAT devices on the network path between two peers that do not allow to establish a transport link. Furthermore, as will be discussed in Section 17.2, E2E communication can only be used at the expense of losing the strong guarantees of the fail-stop model provided by Orbweb.

Establishing and maintaining an E2E connection consumes host resources. This includes, among other things, memory for the XMPP/XML parsing and serialization infrastructure, network ports, and network bandwidth, as well as CPU cycles for session establishment and the half-open connection detection mechanism described in Section 16.3.1. While the resource consumption is essentially the same regardless whether an XMPP session is established with another peer or with the superpeer, there is only one connection to the superpeer but typically many to other peers. Furthermore, E2E sessions are much more volatile, as their lifetime is coupled to the availability of the volatile target peers. Hence, the overall resource consumption for E2E session management is significantly higher as that associated with superpeer-relayed communication. In order to prevent resource exhaustion, Orbweb's E2E facility enforces a limit on the number of concurrent sessions. Since heterogeneity is a characteristic property of Desktop Grids, this limit can be tuned to the capabilities of peers on a per-peer basis.

Limiting the amount of concurrent E2E sessions creates the need for selecting peers for which the establishment of an E2E session is most beneficial. In Orbweb, this is done by sampling outgoing XMPP traffic on the peers. *E2E Sessions* are established with those partners with which most stanzas have been exchanged recently. As virtual topologies are intended to be chosen according to the interaction pattern of applications, messages are likely exchanged more frequently between peers that are neighbors within the virtual topology than between peers that are not. Hence, the E2E communication facility ensures that virtual links are mapped to links on the transport layer when possible.

Besides selecting which peers are promising session partners, the system must also handle the actual session establishment process. This is done by leveraging multiple existing

| **Extension** | **Name** | **Description** |
|---|---|---|
| XEP-0166 | *Jingle* | Enables client-to-client sessions between XMPP entities. Jingle is a pure signaling protocol. Thus it controls the connection negotiation process over the XMPP channel, while P2P interaction is accomplished *out-of-band* using custom communication technologies like the *Real-time Transport Protocol* (RTP), the *User Datagram Protocol* (UDP), or the *Interactive Connectivity Establishment* (ICE) protocol. Jingle is primarily targeted to support media exchange applications like voice or video chats. However, due to its modular design Jingle can be easily extended to support other session types and transport mechanisms. |
| XEP-0247 | *Jingle XML Streams* | Defines a Jingle application type for establishing a direct XML stream between two XMPP entities over a reliable transport. |
| XEP-0246 | *End-to-End XML Streams* | Defines how two peers interact, particularly which XMPP stanzas they exchange, after the session has been negotiated. This includes the exchange of stream headers, the use of the *Transport Layer Security* (TLS) protocol and the *Simple Authentication and Security Layer* (SASL) for establishing the security context and the closing of the XML stream. |

**Table 17.1:** XMPP extensions used by ORBWEB's E2E communication facility

XMPP extensions (see Table 17.1): We use *Jingle* (XEP-0166) and *Jingle XML Streams* (XEP-0247) for session negotiation and *End-to-End XML Streams* (XEP-0246) to establish an XMPP connection between peers according to the results of the negotiation process.

## 17.1 Session Management

ORBWEB's E2E communication facility as depicted in Figure 17.1 samples the outgoing traffic of all local XMPP sessions. This includes the standard client-to-server session (C2S) as well as all E2E sessions. Based on the number of outgoing XMPP stanzas, a priority is computed for each target JID at regular intervals, called *rounds* for the sake of brevity. To lessen the impact of past traffic patterns the priority value is multiplied by an *aging factor* $f_{age} \in ]0,1[$ after each round. Based on these priorities the facility computes a list of JIDs, called the *nominal session list*, for which it is expected to be most beneficial to have an E2E session with. By comparing the nominal session list with the list of active sessions, called the *actual session list*, a series of session establishment and session termination tasks is computed. For each establishment attempt one of the following conditions hold:

1. The attempt succeeds and the new session is added to the actual session list.

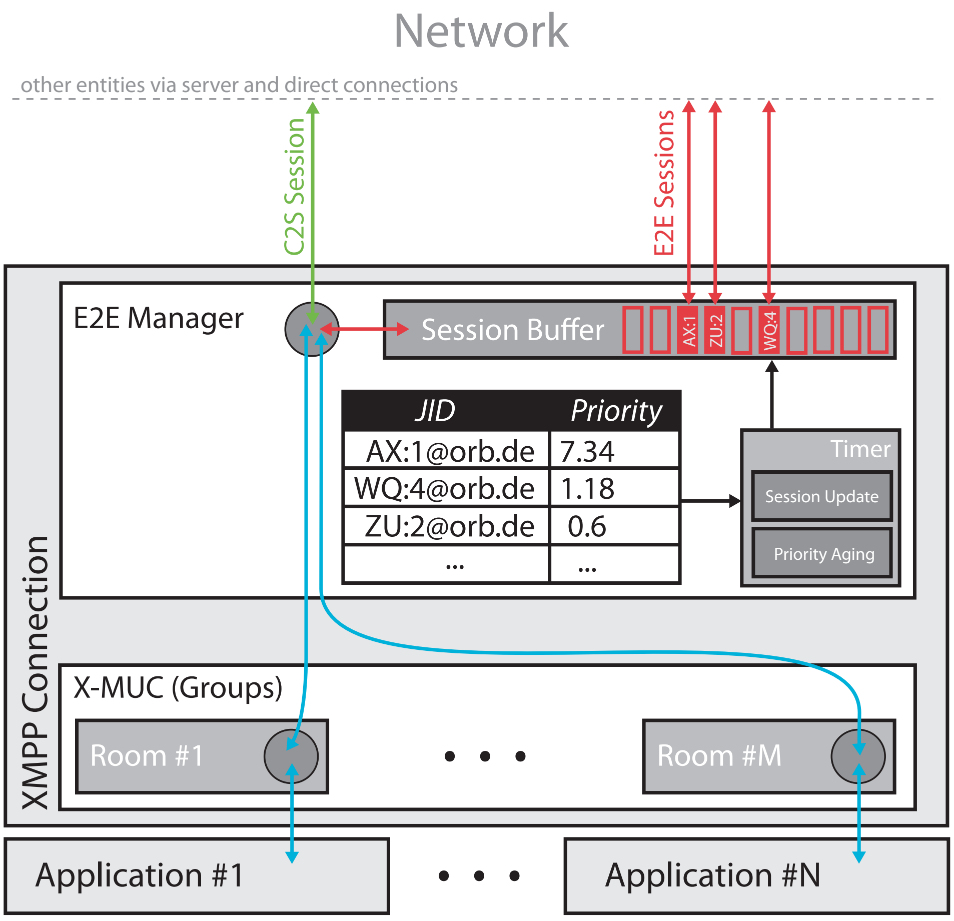


**Figure 17.1:** ORBWEB's E2E communication facility creates E2E sessions to peers with which many messages have been exchanged recently. These sessions are used to deliver XMPP stanzas directly. If no E2E session exists for a given target peer, the facility falls back to superpeer relayed message delivery.

2. The attempt succeeds with respect to TCP connection establishment but during handshake it turns out that the remote endpoint is not the expected peer. This scenario can occur in networks with NAT devices. In this case the session partner is excluded from further connection attempts (*blacklisted*).

3. The attempt fails because the session partner does not respond in a timely fashion or does respond with an error stanza, which happens for example when the session limit on the remote peer is exceeded. In both cases the session partner is *greylisted* according to a *greylisting strategy*, that determines for how much time $T_G$ a potential partner is excluded from session establishment after the $n$-th attempt to establish an E2E session has failed. In our experiments the *logistic function*

$$T_G \propto 1/(1+e^{-n}) \qquad (17.1)$$

turned out to be a good choice.

*Session trashing* is the condition in which a system is spending most of its execution time closing and establishing sessions caused by rapid priority order alteration near the priority limit where a session becomes qualified for E2E session establishment or is displaced by another one. To avoid session thrashing the priority value of a recently closed session is reduced by multiplication with a *penalty factor* $f_{penalty} \ll 1$.

To be able to handle large numbers of connections ORBWEB's E2E communication facility implements the *reactor pattern* based on the *Apache Mina* [min] high performance protocol construction framework. While systems based on the traditional approach of using a thread-per-connection model waste significant parts of the available CPU time with context switching leaving less time for doing actual I/O processing, *Mina* allows for having a single thread serve a large number of sockets through I/O multiplexing enabled by asynchronous I/O introduced in Java v1.4. This results in significantly improved scalability and performance.

## 17.2 Impact on the System Model

When messages are delivered using E2E communication, ORBWEB can not provide the strong guarantees of the fail-stop distributed system model, as the requirement for perfect links is no longer satisfied (cf. Chapter 16). In particular, the *reliable delivery* property of the perfect link abstraction is violated, as E2E sessions may be closed by both endpoints at any time. When this happens while a message is in transit, the message is lost although both endpoints are correct. Note that losing messages is also possible in the fail-stop model, but only when one of the endpoints is not correct. In case an algorithm relies on the fail-stop model, ORBWEB's communication system can be forced to fall back to superpeer-relayed delivery on a per-message basis.

## 17.3 XMPP Network Distance Service

Depending on the application, a peer may exchange messages with a large number of other peers. As described in the introduction to this chapter, it is usually not possible to maintain E2E sessions to all of them. While the E2E facility ensures that E2E sessions are established for the most actively used communication paths, there is no application-level knowledge about the costs of communicating with a remote peer. Provided that the application allows for selecting with which peers to collaborate at what intensity, a peer should prefer interacting with peers over E2E sessions for efficiency reasons. To enable this, ORBWEB's E2E communication facility supports querying for the distance $d_{XMPP}$ in terms of hops to other peers $v$ within the ORBWEB network on the XMPP layer. For a given target peer $v$ possible distances are $d_{XMPP}(v) = 0$ for the peer itself, $d_{XMPP}(v) = 1$ for peers with which an E2E session has been established, and $d_{XMPP}(v) = 2$ for peers between which messages are relayed by the ORBWEB superpeer. Note that this distance metric only describes the number of hops within the XMPP network but is *not* suitable for reliably characterizing the network path to another peer in terms of latency or path length in terms of hops in the IP network. Nevertheless, the utilization of the ORBWEB infrastructure can be optimized by reengineering network- and application-level protocols to be distance-aware as then less XMPP stanzas have to be relayed by the superpeer.

# 18 Virtual Topologies

The interaction patterns between peers can be fundamentally different across applications: While the random stealing algorithm described in Part VII exhibits random interaction patterns, the nodes of a distributed hierarchical information aggregation system as described in Part VI interact along the edges of a tree. For ISPs these interaction patterns are even dynamic and evolving. Hence, maintaining interaction relationships is, especially within volatile environments like Desktop Grids, a challenging task.

To simplify distributed algorithm design and implementation, ORBWEB offers, in accordance with requirements (F1),(F2), and (N2), fine-grained control over the mutual peer visibility[1] within a group. As already known from Chapter 15, ORBWEB peer groups emit ⟨ *pgm*, ViewUpdate ⟩ indications whenever the group membership changes. By tracking these indications a so called *membership view* [GKM03] can be bulit and maintained on the peers. It is defined as follows:

**Definition 18.0.1.** *A* membership view *(or* view *for short) of a peer $p$ is the subset of comembers within a group about whose availability status $p$ is informed.*

Based on this, we can define the abovementioned *mutual peer visibility* more formally as:

**Definition 18.0.2.** *A* virtual topology *of a group $g$ is the graph that consists of the union of the view graphs of all members of $g$. A* view graph *for peer $p$ is the graph consisting of a vertex for $p$ and all peers in the view of $p$ and one edge between $p$ and each peer in the view of $p$.*

ORBWEB exposes *virtual topologies* as an abstraction closely related to the peer group abstraction. To each group a single virtual topology is assigned. This topology is maintained as a data structure centrally on the superpeer and enforced by sending tailored presence stanzas in handler S-MUC-1 of the server-side XMPP MUC protocol (see Protocol 14.9) to customize the membership views on the peers accordingly. Applications can create any number of groups and for each of them select the virtual topology most suitable for the distributed algorithm to be realized.

## 18.1 Implementation

To be able to implement new virtual topologies with minimal effort within the ORBWEB framework, we decouple group management from view management logic. While the

1 Note, that ORBWEB peers can unicast messages to all other peers within the group, even if they are not in their view, as long as they know their peer identifier.

former provides all the functionality defined by the *Peer Group Management* module (see Chapter 15), the latter determines which peers are visible to a peer within a group by means of its membership view. Due to this separation of concerns, only the view management logic has to be implemented, either by starting from scratch or by composition of existing topologies.

The group management part is implemented by a generic replacement for the MUC implementation of *Openfire* called *eXtensible MUC* or *X-MUC*. X-MUC delegates view management to dedicated *view managers*. The architecture is based on the *composite* design pattern [GHJV95]: Each presence stanza received by the X-MUC component, indicating that a peer has joined or left the group, is forwarded to a *view manager* instance. The view manager translates the incoming stanza into a set of outgoing stanzas based on its internal model of the group's topology. Outgoing stanzas carry role information as required by (F2) and are delivered by the X-MUC component to the respective recipients. These in turn update their membership views accordingly.

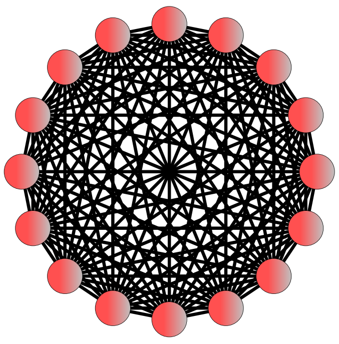
## 18.2 Elementary View Managers

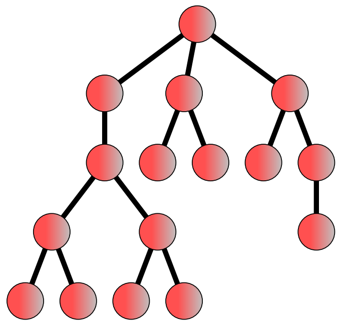
ORBWEB provides a set of elementary view manager implementations satisfying the requirements for a broad range of use cases. Additionally, view manager composition can be used to create more complex topologies by combining two or more elementary view managers. Figures 18.1 a)-e) show sample topologies generated by the elementary view managers. Their complexity characteristics are summarized in Table 18.1. In the following discussion, let $G$ denote the managed group and $|G|$ the number of peers in the group. ORBWEB's elementary view managers are:

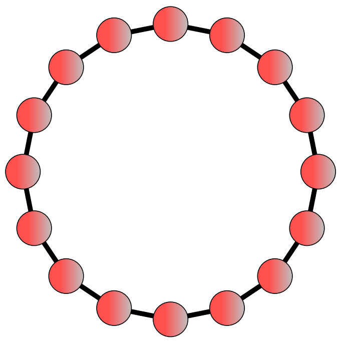
**Complete.** This manager creates the same membership views as the standard XMPP MUC. It realizes a single fixed explicit group model with complete membership views resulting in each group member being aware of the status of each other group member. However, groups with an explicit group model do not scale to thousands of hosts neither client- nor server-side. The server has to transmit a quadratic number of messages to inform each

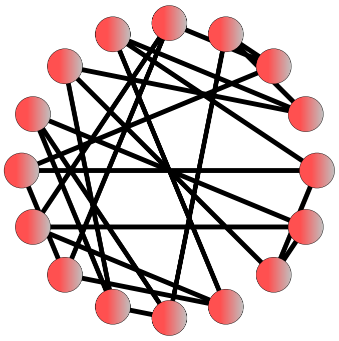
| **Manager** | **Space Complexity** | | **Message Complexity** | **Diameter** |
|---|---|---|---|---|
| | Client | Server | Update | |
| **Complete** | $O(\|G\|)$ | $O(\|G\|)$ | $O(\|G\|)$ | 1 |
| **Blind** | $O(1)$ | $O(\|G\|)$ | $O(1)$ | $\infty$ |
| **RBT** | $O(s)$ | $O(s\|G\|)$ | $O(s)$ | $O(\log(s\|G\|))$ |
| **Ring** | $O(1)$ | $O(\|G\|)$ | $O(1)$ | $\left\lfloor \frac{\|G\|}{2} \right\rfloor$ |
| **Random** | $O(\log(\|G\|))$ | $O(\|G\|\log(\|G\|))$ | $O(\log(\|G\|))$ | - |

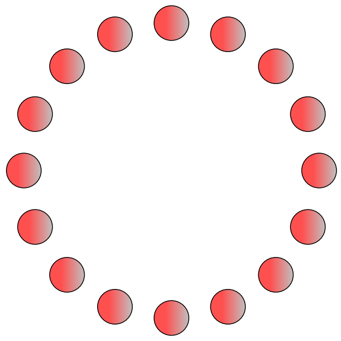
**Table 18.1:** View manager complexity characteristics for a group $G$ of size $|G|$. Update complexities are the costs for handling a single join event. For the RBT view manager, the *view size* $s$ determines how often a node is inserted into the red-black tree.


**(a)** Complete


**(b)** Red-Black Tree


**(c)** Ring


**(d)** Random


**(e)** Blind

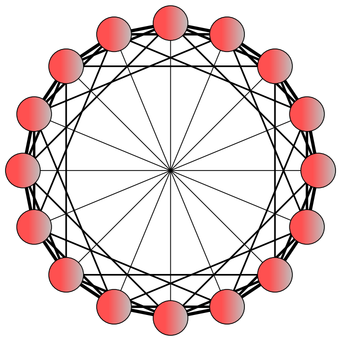

**(f)** Chord

**Figure 18.1:** ORBWEB topologies for a group with 16 members. An edge between two nodes means that they are part of each other's view.

group member of each other's membership status, i.e., join and leave operations are both of $O(|G|)$ time and message complexity. Additionally, the client keeps an account of all other group members, which leads to memory usage linear with respect to the group size on each client.

**Red-Black Tree (RBT).** This manager organizes group members in a tree and notifies peers only about the group presence of its adjoining peers in the tree. Hence, if a peer is not the single member of a group, it will see a minimum of 1 (as a leaf) and a maximum of 3 (as an inner node) other members – we say its *view size* is 3. Each member can also configure its view size to other values, say $s$, which will cause the view manager to insert it $\min(1, \lfloor s/3 \rfloor)$ times with random keys into the tree. The member then has a maximum of $s$ visible neighbors. To be able to perform tree updates in an efficient way, our implementation is based on *Red-Black Trees* [Bay72]. Updates in the tree (a joining or leaving member) can thus be computed in $O(s \log(|G|))$ time, where $s$ is the maximum view size. Configurability of view sizes allows for implementing distributed algorithms that are aware of the capabilities of participating peers and exploit this knowledge to best utilize a heterogeneous resource set [SBH09]. Another use case for configurable view sizes is to provide more detailed information about a group's composition to peers that take special responsibilities within a group. A prominent distributed algorithm that makes use of such a distinguished peer is the three-phase commit protocol [SS87].

The properties of red-black trees allow for providing an alternative implementation of our communication primitives where the server is not involved: As the red-black tree is a spanning tree, we can easily realize a groupcast by propagating messages along its edges. This approach is used in Chapter 19 to implement a scalable probabilistic groupcast primitive. Additionally, as a red-black tree is a binary-search tree, we can use host-to-host message routing for unicasts known from DHTs. The choice of using a red-black tree assures that two members are always connected by at most $4s\log(|G|)$ other members. Thus, the path length is $O(s\log(|G|))$.

**Ring.** This manager arranges the members of a group into a bidirectional ring. The order in which members appear in the ring can be customized by providing a custom comparator for peer identifiers. A prominent example of ring-based distributed algorithms is termination detection [DFvG86].

**Random.** Random networks are used in many distributed algorithms, especially in protocols using gossiping strategies. The random view manager creates random networks where each peer $v$ has a given outdegree $deg^{+}(v)$. While the default value of $deg^{+}(v)$ is $\log(|G|)$, each peer can configure the number of random contacts based on its capabilities, a concept similar to the configurable view size used by the RBT view manager.

**Blind.** This manager implements the anonymous group model providing no information about other group members, which leads to optimal time and space usage, but minimum information. Note, that the group still enforces security restrictions and thus is appropriate to drive the root group realized in most communication frameworks with a hierarchical group model, e.g., the *world peer group* in *JXTA* [TAA+03]. Although, no membership information is communicated, groups using the blind view manager still allow for unicast and groupcast communication, and hence may be used in scenarios where contact information is exchanged by external protocols or through other groups managed by one of the other view managers.

## 18.3 Composite View Managers

A key strength of ORBWEB's view management is the ability to compose complex topologies from simpler ones. The composition mechanism is based on a special view manager that aggregates a set of subordinate view managers. To create such a *composite view manager* the implementor simply specifies what happens on a peer join or leave event: to which subordinate view manager(s) to add a joining peer with which parameters and from which to remove a leaving peer. The composite view manager translates presence updates from subordinate view managers into a single consistent view by interception and merging of individual presence stanzas. Note, that view managers may attach attributes to presence stanzas, which can be used by peers to differentiate between view members contributed by different subordinate view managers.

### 18.3.1 Chord

We illustrate the concept of composite view managers using the *Chord* P2P document routing protocol [SMK+01]. *Chord* arranges peers in a ring topology based on a unique identifier created by applying a hash function on local information that is unique for a given peer. Keys are mapped to peers using *consistent hashing* [KLL+97] which ensures that only a small number of key reassignments are necessary in the face of peer arrivals or departures: Identifiers are ordered in an identifier circle modulo $2^m$, where key $k$ is assigned to the first peer whose identifier is equal to or follows $k$ in the identifier space. In addition to the predecessor and successor within the ring topology, each peer maintains a set of links called *fingers*, where the $i^{th}$ finger references the $2^{i-1}$-th successor in the ring. Fingers ensure efficient lookup as they allow implementing a distributed binary search. Key lookup happens iteratively: a peer $q$ receiving a lookup request from peer $p$ for key $k$ searches its finger table for the peer $r$ whose identifier most immediately precedes $k$ and sends the result back to $p$. Peer $p$ then sends a lookup request to peer $r$. Hence, $p$ iteratively learns about peers with identifiers closer and closer to $k$.

To implement *Chord* on top of ORBWEB, we encode *Chord* identifiers into the identifier of ORBWEB group members and use a composite view manager consisting of a ring view manager that sorts peers according to this identifier and a custom view manager that is responsible for maintaining the fingers. While the ring is updated on every join/leave event to guarantee routability within the *Chord* P2P network, applying the same strategy for maintaining the finger topology would be very inefficient as a large number of fingers would have to be updated every time a node joins or leaves. To avoid such massive reconfigurations, we restrict immediate updates to the addition of the fingers for the newly arrived peer. All other updates are performed by a background thread that periodically selects a small set of peers at random and updates their fingers. Hence, we trade off routing efficiency against topology maintenance costs and decouple the latter from the degree of resource volatility. The topology created by the ORBWEB *Chord* view manager is depicted in Figure 18.1 f) for a 16-node group. The topology includes the basic ring and fingers up to the $4^{th}$ order.

Although ORBWEB's superpeer-assisted *Chord* implementation does not scale as well as the original fully distributed protocol, it still supports network sizes of thousands of hosts typical for P2P Grid applications. Complete knowledge of the membership list results in superior convergence of the *Chord* network, which is of particular importance for high performance applications executing on highly volatile networks typical for P2P Desktop Grid Computing systems. Both claims are substantiated by experimental results discussed in Chapter 23.

# 19 Group Communication

As described in chapters 14-16, ORBWEB provides the application developer with a *fail-stop* distributed system model that includes *perfect* point-to-point links. However, interaction in distributed systems is typically not restricted to bilateral relationships. Often multiple parties have to interact in a coordinated manner. This point-to-manypoint type of communication is covered by a concept called group communication. Group communication primitives can come in various types and flavors. Unfortunately, the terminology is not consistent in the literature. In the context of this work, we speak of a *broadcast* when all and of a *multicast* in case only a subset of the nodes of a system are the recipients of a point-to-manypoint communication operation. As group communication in ORBWEB is closely related to the peer group abstraction, we usually refer to a multicast as a *groupcast* operation.

Group communication primitives can come in various flavors exhibiting different qualities. In accordance with requirement (N2), ORBWEB allows applications to select between a groupcast protocol that makes deterministic and one that makes probabilistic reliability guarantees. The former is provided by the groupcast scheme specified by the XMPP MUC standard as described in Chapter 14. The latter is a modification of the *Bimodal Multicast* [BHO+99] protocol adapted to the specifics of the Desktop Grid environment. While the deterministic groupcast scheme is, as confirmed by our performance measurements in Chapter 22, not scalable – neither with respect to message size and rate nor with respect to group size – it is *(regular) reliable* and *totally ordered*. As will be discussed in Section 19.2, the guarantees given by the probabilistic groupcast are much weaker. But in exchange its scalability is significantly better.

## 19.1 Totally Ordered Regular Reliable Groupcast

As described in Chapter 14, the XMPP network clients are arranged in a star topology with the XMPP server at its center. The *totally ordered regular reliable groupcast* (*torrg* for short and occasionally referred to as *servercast* in the following) provided by ORBWEB uses this star topology to perform a *replicated unicast*: A message that is groupcast by some member of peer group $G$ is sent to the XMPP server first. The XMPP server then forwards the message to all group members involving a total of $|G|$ unicasts. The formal definition of the associated module is shown in Module 19.1. The implementation has already been presented in C-MUC-5 of Protocol 14.7 and S-MUC-2 of Protocol 14.9. We now give a proof that our implementation guarantees the properties TORRG1-TORRG5. We assume without loss of generality that all correct peers are members of some group $g$. The proofs for the *no duplication* and *no-creation* property are analogous to the respective proofs for perfect links in Section 16.4 and are thus omitted.

**Module 19.1** Interface of the Totally Ordered Regular Reliable Groupcast module

**Module:**

**Name:** Totally Ordered Regular Reliable Groupcast, **instance** *torrg* within group *g*.

**Requests:**

⟨ *torrg*, Groupcast | *message* ⟩: Send the given *message* to all members of the group *g*.

**Indications:**

⟨ *torrg*, Deliver | *source*, *message* ⟩: Deliver the given *message* received from the peer with identifier *source*.

**Properties:**

**TORRG1:** *Validity:* If a correct peer $p$ groupcasts a message $m$, then $p$ eventually delivers $m$.

**TORRG2:** *Agreement:* If a message $m$ is delivered by some correct peer, then $m$ is eventually delivered by every correct peer.

**TORRG3:** *No duplication:* No message is delivered more than once.

**TORRG4:** *No creation:* If a peer delivers a message $m$ with sender $q$, then $m$ was previously groupcast by peer $q$.

**TORRG5:** *Totally Ordered:* Let $m_1$ and $m_2$ be two messages and suppose $p$ and $q$ are any two correct peers that deliver $m_1$ and $m_2$. If $p$ delivers $m_1$ before $m_2$, then $q$ delivers $m_1$ before $m_2$.

**Lemma 19.1.1.** *If a correct peer $p$ groupcasts a message $m$, then $p$ eventually delivers $m$ (TORRG1).*

*Proof.* $p$ is correct and thus has by definition 16.3.1 an open connection to the superpeer. Hence, $m$ is received by the superpeer which triggers the event handler S-MUC-2. As $m$ is sent to all members of $g$ in S-MUC-2 and $p$ is a group member of $g$ (by assumption and due to the fact that it is correct), $m$ is sent to $p$. As $p$ is correct, $m$ is received by and delivered on $p$ by C-MUC-5 and PGM-6. ■

**Lemma 19.1.2.** *If a message $m$ is delivered by some correct peer, then $m$ is eventually delivered by every correct peer (TORRG2).*

*Proof.* As $m$ is delivered by some correct peer, it must have been sent to that peer by the superpeer as part of the event handler S-MUC-2 due to TORRG4. As in S-MUC-2 all correct peers are traversed, $m$ is sent to all correct processes. As correct processes have an open session with the superpeer, $m$ is delivered eventually by all of them. ■

**Lemma 19.1.3.** *Let $m_1$ and $m_2$ be two messages and suppose $p$ and $q$ are any two correct peers that deliver $m_1$ and $m_2$. If $p$ delivers $m_1$ before $m_2$, then $q$ delivers $m_1$ before $m_2$ (TORRG5).*

*Proof.* As $p$ delivered $m_1$ before $m_2$, peers process events in FIFO order, and because of the FIFO delivery property (ST1) of the *Session Transport* module, $m_1$ has been sent to $p$ by the superpeer before $m_2$. As the superpeer event handlers are executed atomically, i.e., two executions of the loop in S-MUC-2 never interleave, $m_1$ has been sent to $q$ before $m_2$. As of ST1 and the FIFO order handling of events, $m_1$ is delivered before $m_2$ on $q$. ■

## 19.2 Probabilistic Groupcast

To ensure the reliability property of multicast schemes despite faulty processes, the sender has to collect acknowledgments from the receiving processes. Due to finite sender resources (bandwidth, memory, CPU cycles) this problem known as the *ack implosion problem* limits the scalability of reliable multicast schemes. Although these limits can be pushed by using hierarchical approaches that distribute the onus of acknowledgment processing by aggregating incoming acknowledgments on multiple levels, the scalability limitation by itself is intrinsic to these *sender-initiated* schemes. Furthermore, they exhibit the possibility of unpredictable performance under stress or in the face of slow or stalled participants [BHO+99, OOB00]. Even with a stable network of equally powerful participants, these protocols can hardly scale beyond several hundred participants [PS97].

One way to circumvent the scalability limitation of sender-initiated multicast schemes is to put the burden of failure detection (omission or corruption of messages) to the receivers. The resulting scheme is called *receiver-initiated* as the receiver initiates a retransmission in case of failure using a *negative acknowledgment* (NACK). Due to their decentralized mode of operation, maintaining delivery state for messages becomes practically infeasible. Hence, receiver-initiated multicast schemes give up the deterministic validity and the agreement property guaranteed by reliable multicast protocols. Instead they offer what is called *probabilistic validity* [CRG11]. The interface of a probabilistic groupcast is given in Module 19.2.

A prominent and well understood example for this kind of multicast scheme is the *Scalable Reliable Multicast* (SRM) [FJL+97]. However, it has been shown that SRM behaves pathologically under certain conditions resulting in *retransmission storms* [Liu97, Luc98]. As the problematic behavior is triggered by transient high rates of message loss, SRM can be expected to be a poor choice for P2P Desktop Grids, where message losses caused by unexpected host departures or perturbations due to slow or stalled hosts are likely. Hence, ORBWEB implements an adapted version of another probabilistic multicast scheme called *Bimodal Multicast* [BHO+99] that has been designed to overcome the limited robustness of SRM. The next four sections summarize the essentials of the original algorithm and introduce our variant with the adaptions required to cope with the specifics of the Desktop Grid execution environment.

### 19.2.1 Bimodal Multicast

*Bimodal Multicast* – abbreviated as *pbcast* by its inventors – is composed of two phases: The first is an unreliable groupcast that makes a best-effort attempt to efficiently deliver a message to all group members. The second is a round-based two-phase anti-entropy protocol that detects and corrects inconsistencies in message history by continuously gossiping summaries of the local message history. The first subphase of the anti-entropy protocol detects message loss and if required triggers subphase two, where such losses are compensated.

Figure 19.1 illustrates the execution of the *Bimodal Multicast* protocol in a group of 4 peers. After a period of unreliable groupcasts (participants $P_1$, $P_2$, and $P_4$ each send a groupcast message, where participant $P_2$ fails to receive message $M_0$, $P_4$ lacks $M_1$, and $P_3$ misses $M_2$) constituting the first phase, the two-phase anti-entropy protocol is executed.

**Module 19.2** Interface of the Probabilistic Groupcast module

**Module:**

**Name:** Probabilistic Groupcast, **instance** *pg* within group *g*.

**Requests:**

⟨ *pg*, Groupcast | *message* ⟩: Send the given *message* to all members of the group *g*.

**Indications:**

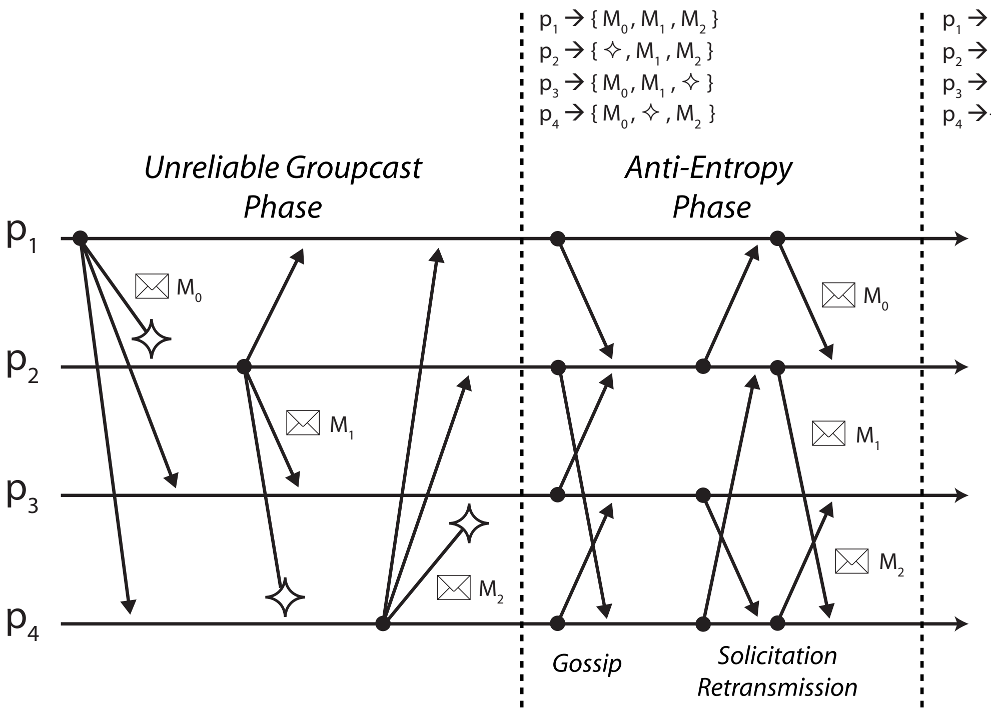
⟨ *pg*, Deliver | *source*, *message* ⟩: Deliver the given *message* received from the peer with identifier *source*.

**Properties:**

**PG1:** *Probabilistic Validity:* There is a $\varepsilon > 0$ such that when a correct peer groupcasts a message $m$, the probability that every correct peer eventually delivers $m$ is at least $1 - \varepsilon$.

**PG2:** *No duplication:* No message is delivered more than once.

**PG3:** *No creation:* If a peer delivers a message $m$ with sender $q$, then $m$ was previously groupcast by peer $q$.



$p_1 \rightarrow \{ M_0, M_1, M_2 \}$
$p_2 \rightarrow \{ ✧, M_1, M_2 \}$
$p_3 \rightarrow \{ M_0, M_1, ✧ \}$
$p_4 \rightarrow \{ M_0, ✧, M_2 \}$
$p_1 \rightarrow \{ M_0, M_1, M_2 \}$
$p_2 \rightarrow \{ M_0, M_1, M_2 \}$
$p_3 \rightarrow \{ M_0, M_1, M_2 \}$
$p_4 \rightarrow \{ M_0, M_1, M_2 \}$
Unreliable Groupcast Phase
Anti-Entropy Phase
$p_1$
$p_2$
$p_3$
$p_4$
$M_0$
$M_1$
$M_2$
$M_0$
$M_1$
$M_2$
Gossip
Solicitation Retransmission

**Figure 19.1:** Phases of the *Bimodal Multicast*

Note, that the illustration simplifies the actual process, as the groupcast and the anti-entropy phases are executed concurrently and participants advance independently. In the gossip subphase of the anti-entropy protocol each participant randomly selects another participant to which it sends a *digest* of its message buffer. All incoming payload messages are put into the message buffer and are removed after a configurable number of rounds. A digest contains the sequence numbers of all messages within the buffer. In the second subphase of the anti-entropy protocol those participants that have received a digest perform a comparison with their own buffer entries. For each missing message they send a retransmission request, called a *solicitation*, to the sender of the digest. Upon receipt of a solicitation participants respond with the retransmission of the requested message. At the end of the second subphase of the anti-entropy protocol the message buffers of all participants have been populated with messages that were not reliably transmitted by the groupcast phase. The inventors of the *Bimodal Multicast* propose a number of optimizations. For an in-depth discussion see [BHO+99].

### 19.2.2 Network Segment Detection

As discussed in the introduction to Part I, a distinguishing feature of Desktop Grids is the fact that they are usually operated over wide area networks (WAN) or the Internet. Due to the universal deployment of network address (and port) translating (NA(P)T) devices and restrictive firewalls, together often referred to as *middleboxes*, this results in restricted connectivity between the geographically and administratively distributed hosts of a Desktop Grid. The testbed used for our experimental analysis in Chapter 22 serves as a good example here. Although, it spans hosts from one organization only, the network infrastructure includes three firewalls and two NAPT devices (see Figure 22.1).

Middleboxes create what is called *network segments*. Network segments are used to isolate parts of the network from each other for various reasons and on various layers of the ISO/OSI stack. For example routers are used to separate broadcast domains. Firewalls and NAPT devices are primarily used for security reasons, although NAPT devices also help to mitigate the *IPv4 address exhaustion* problem. In the context of this work, we do not take any other middleboxes than firewalls and NAPT devices into account. Future work may also include other device types. In particular, routers may be oversubscribed, i.e., the overall available bandwidth may be exceeded, which leads to packet drops at segment boundaries and hence limited cross-segment communication performance. Network segments created by routers could be detected by exploiting the fact that they define IP broadcast domains. Hence, such a segment could be detected by instructing a peer to emit a UDP broadcast and then having every receiving node sending an acknowledgment to the broadcasting peer. However, already mentioned, this is future work.

If ORBWEB operates over a segmented network, the restricted connectivity between the segments prevents the establishment of E2E sessions between hosts located in different segments. If distributed algorithms are not designed to take this fact into account, the load on the ORBWEB superpeer will be high due to a large number of relayed messages. In addition, there is most often no or limited knowledge about the network segmentation available publicly because of security concerns. Even if this was not true, the costs of manually (re-)configuring the substrate to consider network segments are prohibitive in a

P2P Desktop Grid environment, where no or limited administrative manpower is available. Hence, ORBWEB implements an adaptive algorithm for network segment detection that allows for self-management and eliminates the need for manual intervention.

ORBWEB models network segments as so called *network components* (or just *components* for short). We introduce the notion of a component because a component only eventually becomes identical to a network segment. Each component is identified by a unique component identifier that is made available to both peers and the superpeer. When a peer joins the ORBWEB network by connecting to the superpeer, it is assigned to its own new component. This is done as at this time no information is available whether the peer belongs to an existing component and if yes to which one. The superpeer periodically issues *probe* requests to pairs of peers $(p_i, p_j)$ selected randomly from two randomly selected components. On receipt of such a probe request, the E2E manager at the recipient peer tries to establish an E2E session with the other peer and sends the result of this attempt back to the superpeer. If for *both* peers the attempt succeeds, the peers are considered to be within the same network segment and hence are assigned to the same component. To avoid false negatives, E2E session limits are not enforced for probing requests.

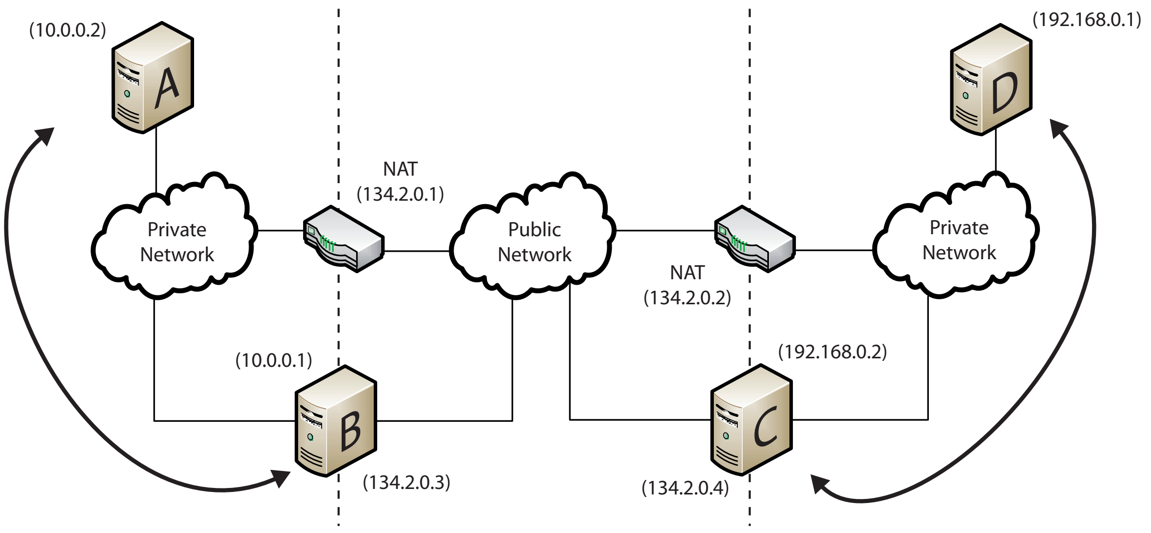
A prerequisite to the correctness of this algorithm is that connectivity is transitive: For three peers $p_i$, $p_j$, and $p_k$ this requirement means that if E2E sessions can be mutually established between $p_i$ and $p_j$ as well as between $p_j$ and $p_k$, then sessions can be mutually established between $p_i$ and $p_k$. As illustrated by Figure 19.2a and 19.2b, assuming transitivity is justified for typical WAN setups as long as firewall policies are reasonable and the IP address within the private network is used for E2E session establishment on multi-homed machines.

## 19.2.3 Topology-Aware Bimodal Multicast

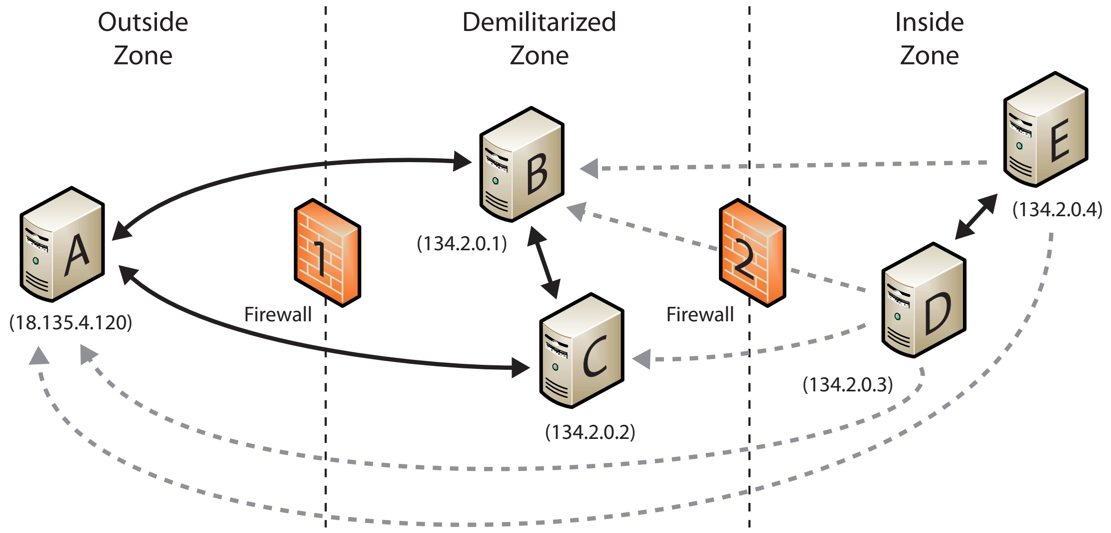
The anti-entropy protocol of *Bimodal Multicast* randomly selects peers as receivers for gossip messages without considering the underlying physical network topology. While this strategy is suitable for LAN settings with full connectivity, it is, as described above, *not* well-suited for scenarios with network segmentations, as a large number of gossip messages would have to be relayed by the superpeer to peers in other segments. To remedy the resulting scalability limitation, we extend the *Bimodal Multicast* protocol to take the segmentation of the underlying network into account.

Therefore each peer is provided with a membership view for execution of the anti-entropy protocol that is composed of peers selected randomly from the set of peers located within the same network segment only. We refer to this extended protocol as the *Topology-Aware Bimodal Multicast*[1] or *ta-pbcast*. A *ta-pbcast* operation first sends the message to the superpeer, which forwards the message to a single peer in each component that is the root of a spanning tree used to efficiently distribute the message within that segment. Subsequently, the anti-entropy protocol is performed within each component.

1 Network topology detection in general also includes the detection of other aspects of the physical network infrastructure, like network switches. However, when we speak of *topology awareness*, we mean the awareness regarding network segmentation as described above.



**(a)** Using the IP address within the private network ensures compliance with the transitivity requirement in scenarios with multiple private networks and multi-homed machines, e.g., cluster heads. The components are correctly detected as $\{A,B\}$ and $\{C,D\}$, which would not be the case if B or C used its public IP address for E2E session negotiation.



**(b)** Connectivity is transitive for common firewall setups incorporating multiple security zones, e.g., a DMZ for publicly exposed server machines $\{B,C\}$, as long as firewall policies are reasonable. Allowing incoming connections to $E$ from $B$ or $C$ for example, would violate transitivity but would also compromise security as a successfully hijacked node B would result in exposition of the inside zone. The correctly detected components are $\{A,B,C\}$ and $\{D,E\}$.

**Figure 19.2:** Transitivity of connectivity in typical WAN setups. Dashed lines with one arrowhead indicate unidirectional, solid ones with two arrowheads indicate bidirectional connectivity.

While performing the anti-entropy protocol only within components saves resources on the superpeer, it reduces the probability that a groupcast message is eventually delivered to all correct peers in that group. This is due to the fact that for the *Bimodal Multicast* $\varepsilon$ in the probabilistic validity property of the Probabilistic Groupcast interface definition in Module 19.2 drops exponentially with the number of participants for a fixed share of infected participants after the unreliable groupcast phase. However, as pointed out by Birman *et al.* [BHO+99], achieving a high share of infected peers by means of the unreliable groupcast is decisive for the success of a *pbcast* operation. Hence, we increase this share by having the superpeer select a configurable number of additional *injection points* for messages to be groupcast, i.e., peers within the same component. With respect to message propagation, each injection point behaves like it was the root of the propagation tree: the injected message is propagated to its children as well as to the parent peer. As we give priority to stable peers, we refer to this scheme as *stability-aware multi-injection*. The share of infected peers is increased by stability-aware multi-injection for two reasons: First, by using $k$ injection points, we have at least $k$ infected peers. By selecting the most stable ones, the probability that they are available during the anti-entropy phase is maximized. Second, the probability that a propagation path is truncated due to a departing peer and the resulting reconfigurations of the spanning tree is minimized, as every propagation path in the spanning tree is likely covered by multiple propagation paths within the trees rooted at the injection points.

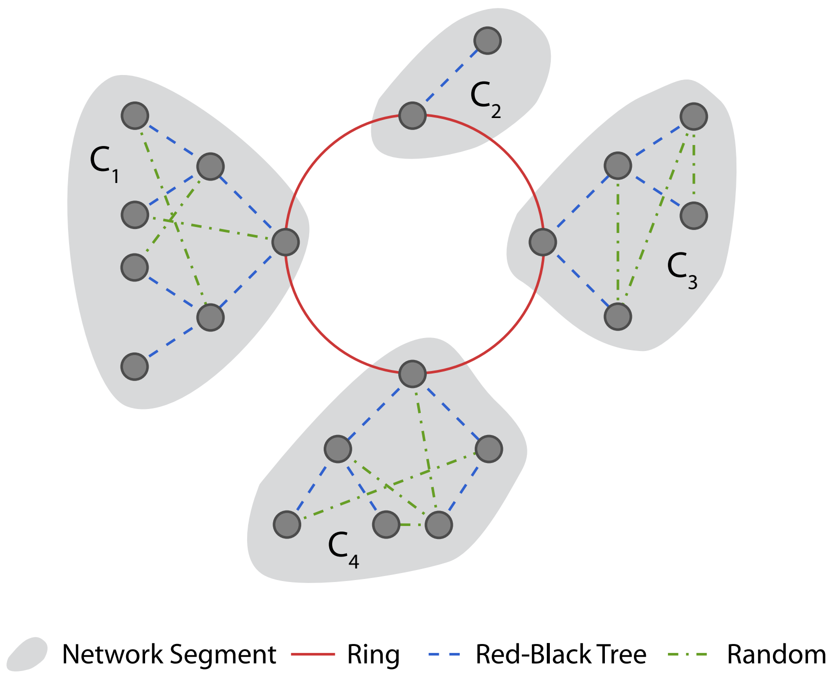


**Figure 19.3:** Composite virtual topology maintained by the *ta-pbcast* view manager

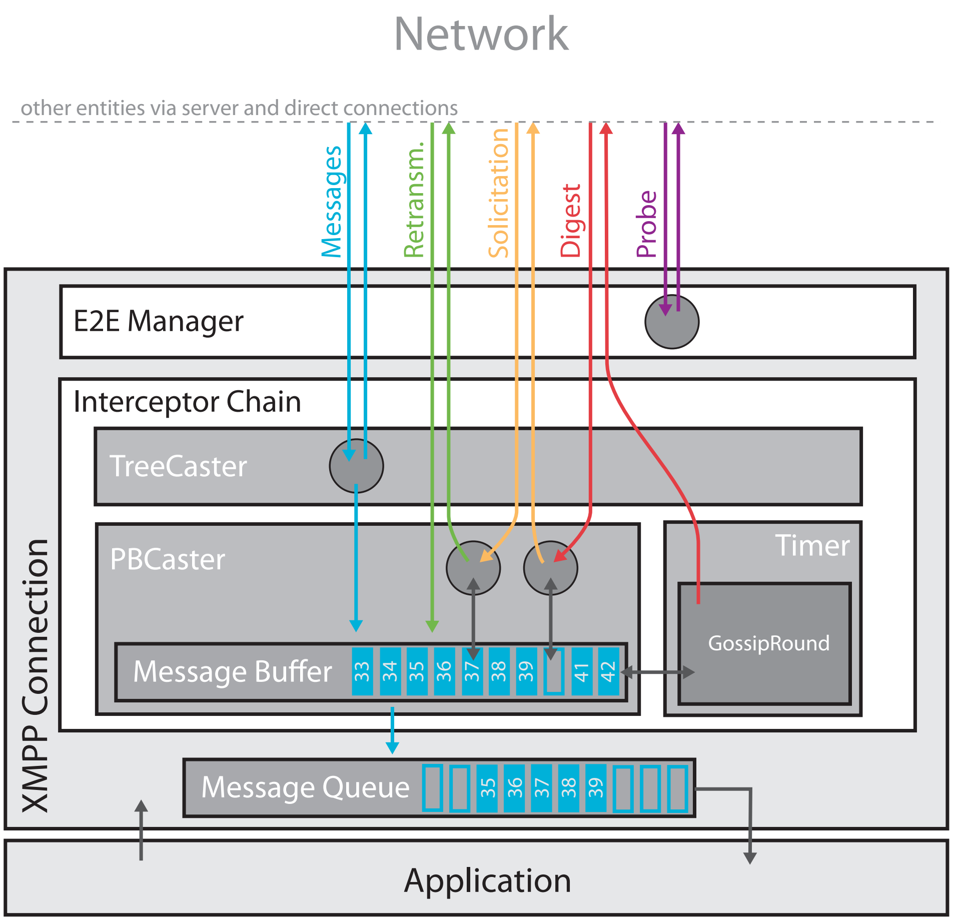


**Figure 19.4:** Components of the client-side *ta-pbcast* logic

### 19.2.4 Implementation

The *ta-pbcast* implementation of ORBWEB is based on a composite view manager combining a ring view manager and $|C|$ superimposed red-black tree (RBT) and random view managers, where $C$ is the set of components detected by the network segment detection mechanism described above. The resulting overall virtual topology is illustrated in Figure 19.3.

The *ta-pbcast* virtual topology is populated as follows: When a peer joins the group, it is first integrated into the ring topology. Thus, it becomes the root of a new red-black tree and is served with groupcast messages by the superpeer directly. In case two components are detected to belong to the same network segment, the red-black trees of both are merged into a single one.

Figure 19.4 shows the schematic for the client-side groupcast logic. The view members contributed by the RBT view manager are used by the `TreeCaster` for the first phase of the *Bimodal Multicast* protocol. To avoid unnecessary message propagations – either caused by rotations within the RBT tree or due to multi-injection – the `TreeCaster` uses a unique sequence number that is attached to incoming messages by the superpeer to decide over which virtual links an incoming message still has to be propagated. The `PBCaster` performs the anti-entropy phase of the *Bimodal Multicast* protocol with the view members managed by the random view manager.

As all tree and random edges exist between peers within the same network segment, no more messages have to be relayed by the superpeer as soon as E2E session establishment has been completed. A *ta-pbcast* groupcast operation thus has a superpeer-side message complexity of $O(k|C|))$, which results in costs that are in general significantly smaller than the costs of the superpeer-based *torrg* groupcast.

# 20 Efficient XML Processing

XMPP clients and servers spend a significant amount of time on processing XMPP stanzas. Thus, an attempt to increase the overall performance of the communication subsystem should primarily address optimizing XML processing. In this section, we describe how we optimized the XML processing stack of *Openfire*/*Smack* by incorporating a binary XML encoding called *Fast Infoset* (FI) [fi] to yield substantial latency and throughput improvements.

Note that if performance was the only required quality for a network substrate, another technology than XMPP would have been probably the better choice. However, as XMPP provides many concepts that can be used to realize the functional requirements described in Chapter 12, we think that XMPP is – despite the impact of using XML on the performance, superior to the available alternative substrate technologies discussed in Chapter 23.

## 20.1 Fast Infoset

A major drawback of XML is that it is verbose. Since document size affects all stages in the XML processing chain (serialization, transmission, and parsing), techniques to reduce document size promise to increase processing performance. However, there is a trade-off between document size and pre- and postprocessing effort. Simple stream compression methods like GZIP significantly reduce document size, but at the same time cause considerable pre-/postprocessing overhead. *Fast Infoset* overcomes this limitation by interweaving serialization and compression or decompression and parsing, respectively.

FI specifies a binary encoding format for the XML Information Set. It aims to provide more efficient serialization and parsing than the character-based standard XML format (hereinafter referred to as *W3C*). FI is used to optimize both document size ($\approx 50\%$ on average compared to standard XML 1.0 serialization using *Apache Xerces* v2.7.1 [fi]) and processing performance ($\approx 25\%$ faster serialization and between 5 and 8 times faster parsing compared to *Xerces* v2.7.1 SAX) and thus is more advanced than simple stream compression based on GZIP used in contemporary XMPP servers. These improvements are achieved through exploiting redundancy by avoiding end-tags, applying string indexing and Huffman encoding, aligning information for faster access and by directly embedding binary data into the stream, bypassing the usually necessary conversion to Base-64 representation.

## 20.2 Implementation

*Openfire* and ORBWEB's E2E facility both leverage the *reactor* design pattern [Sch95] to achieve high scalability. Pending I/O-operations are handled sequentially by a single dispatcher thread (per CPU core). This processing model is very efficient, since fewer threads means less resource consumption and fewer context switches. However, since XMPP stanzas are delivered as elements embedded in a single large XML stream, this non-contiguous I/O processing style is problematic, as to our best knowledge, there are no non-blocking Java XML parsers available. A non-blocking XML parser returns immediately when no more data is available from the input stream, leaving the parser in a continuable state and allowing the dispatcher thread to continue with processing pending operations from other connections.

In the original *Openfire* implementation this problem is solved by prepending a lightweight parser to the actual parser, that is non-blocking and that buffers incomplete XMPP stanzas as long as they become complete. However, this approach is no longer applicable when FI is used: First, implementing a lightweight parser doing the same job is a very complex task, since decoding FI is considerably more involved than decoding a standard XML Infoset document or fragment. Second, FI relies on indexing tables that are prepended to the actual octet stream. Invoking the actual parser on the extracted XMPP stanza would imply updating the tables, which would add significant overhead.

We circumvent this problem by dissecting the incoming byte stream containing the XMPP stanzas (see Figure 20.1). For that purpose, we modified the XML serializers to prepend a header to each stanza specifying its length in bytes. Receiver-side logic reads the header and waits until the specified number of bytes have been received. As soon as the whole sequence of bytes has been received, the whole stanza is forwarded at once to the XML parser, which immediately returns after the end tag concluding the XMPP stanza has been read. The length header is encoded as an octet-packed integer[1] for space-efficiency reasons.

```
                <stream:stream to="theta"
                               xmlns="jabber:client"
 TCP                           xmlns:stream="http://etherx.jabber.org/streams"
Packet                         version="1.0">

  I     0x0     E1                                                  ...á
---------------------------------------------------------------------------------
  II    0x0000  E0 00 00 01 00 78 CD 0C-6A 61 62 62 65 72 3A 63     à....xÍ.jabber:c
        0x0010  6C 69 65 6E 74 CF 05 73-74 72 65 61 6D 1F 68 74     lientÏ.stream.ht
        0x0020  74 70 3A 2F 2F 65 74 68-65 72 78 2E 6A 61 62 62     tp://etherx.jabb
        0x0030  65 72 2E 6F 72 67 2F 73-74 72 65 61 6D 73 F0 3F     er.org/streamsð?
        0x0040  81 82 05 73 74 72 65 61-6D 78 01 74 6F 44 74 68     □,.streamx.toDth
        0x0050  65 74 61 78 06 76 65 72-73 69 6F 6E 42 31 2E 30     etax.versionB1.0
        0x0060  F0                                                  ð

        <iq id="F9P2Vp-0" to="cohesion" type="get">
            <query xmlns="jabber:iq:register"></query>
        </iq>

 III    0x0     C7                                                  ...Ç
---------------------------------------------------------------------------------
  IV    0x0000  E1 00 74 FF 7C 01 69 71-78 01 69 64 46 39 50 32     á.tÿ|.iqx.idF9P2
        0x0010  56 70 2D 30 00 47 63 6F-68 65 73 69 6F 6E 78 03     Vp-0.Gcohesionx.
        0x0020  74 79 70 65 42 67 65 74-F0 38 CD 11 6A 61 62 62     typeBgetð8Í.jabb
        0x0030  65 72 3A 69 71 3A 72 65-67 69 73 74 65 72 F0 3C     er:iq:registerð<
        0x0040  04 71 75 65 72 79 FF                                .queryÿ
```

**Figure 20.1:** FI encoded XMPP stream with headers (TCP packets I and III) specifying the length of the following XMPP stanza.

1 The *octet-packed integer encoding* represents arbitrary large integer values as sequences of octets. The highest bit is used to mark the last octet of a sequence.

# 21 Tooling

Understanding what happens within large distributed systems is complicated by the large number of interacting nodes, lack of centralized access to their execution context, and ubiquitous concurrency. For this reason, tooling is of particular importance on all levels of P2P Desktop Grid Computing systems. This is particularly true for the network substrate that forms the basis for the upper layers of the system. ORBWEB meets this demand by providing a rich set of supportive tools including traffic analysis and online visualization of virtual topologies and network segmentation. We describe these tools subsequently.

## 21.1 Traffic Analysis

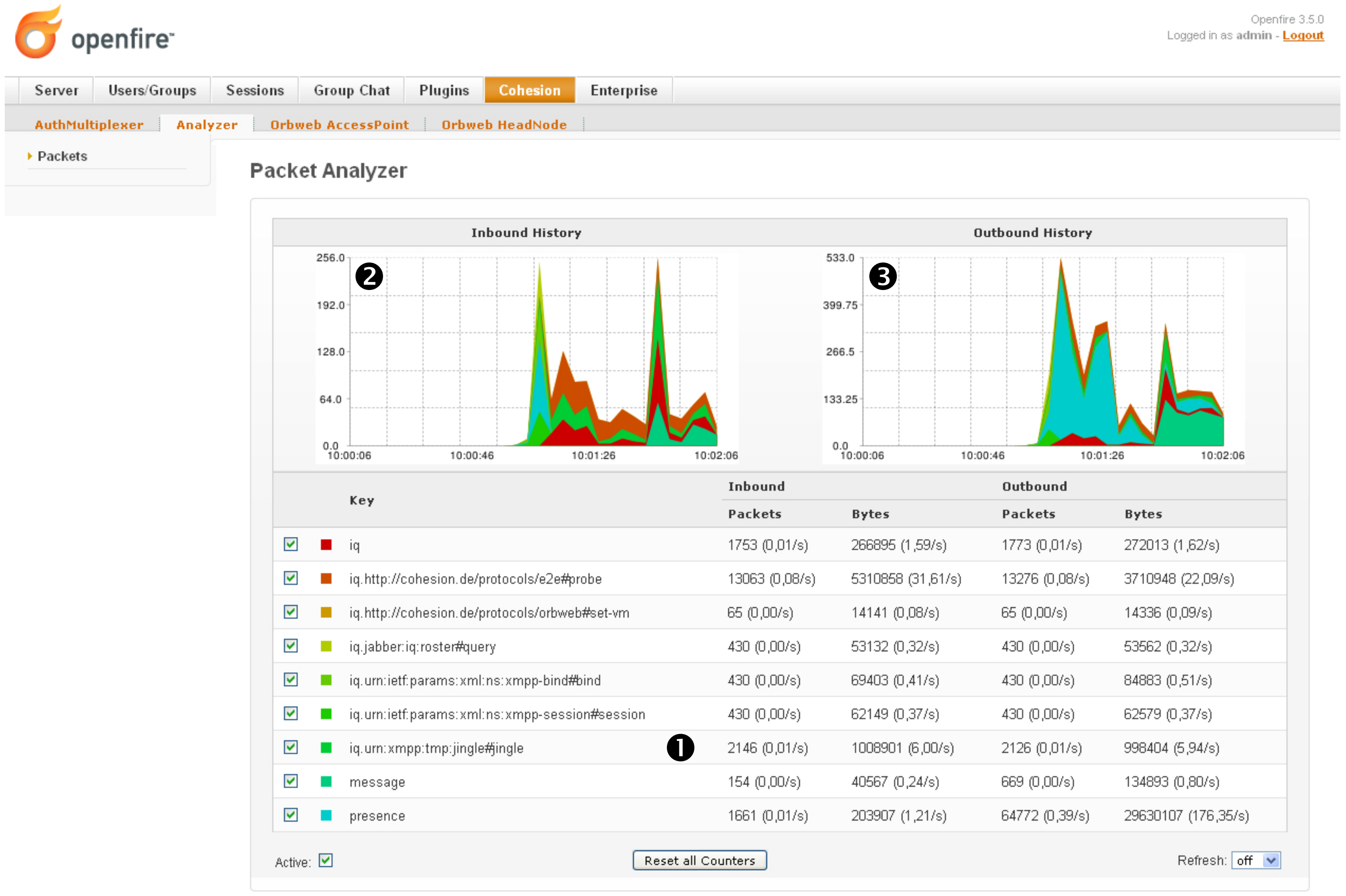
XMPP and its extensions specify a large number of different stanza types. When developing complex protocols, like *ta-pbcast* (see Chapter 19), network traffic data provided by domain independent network analysis tools like WireShark [ORBP07] is insufficient to gain a precise understanding of the actual packet flow produced by the protocol implementation. Hence, ORBWEB provides a packet analysis tool that is implemented as a plugin for and thus tightly integrates with the *Openfire* XMPP server. Figure 21.1 shows a screenshot of the tool: It provides counters for incoming, outgoing, the overall number and the transmission rate for presence, IQ (including namespace and action), and message stanzas (❶). To get a quick overview of the share a stanza type contributes to the overall traffic, a stacked chart visualization is provided (❷, ❸).

## 21.2 Component Visualization

As described in Chapter 19, the *ta-pbcast* view manager automatically identifies network segments among the set of participating peers. As the message complexity of a groupcast operation directly depends on the number of network segments, knowledge about the network topology is of prime importance for debugging and evaluation purposes. Hence, ORBWEB provides a tree map visualization of the components currently identified by the *ta-pbcast* view manager (see Figure 21.2). In conjunction with the traffic analyzer the component visualization can provide valuable insight into the reasons of pathological traffic patterns.

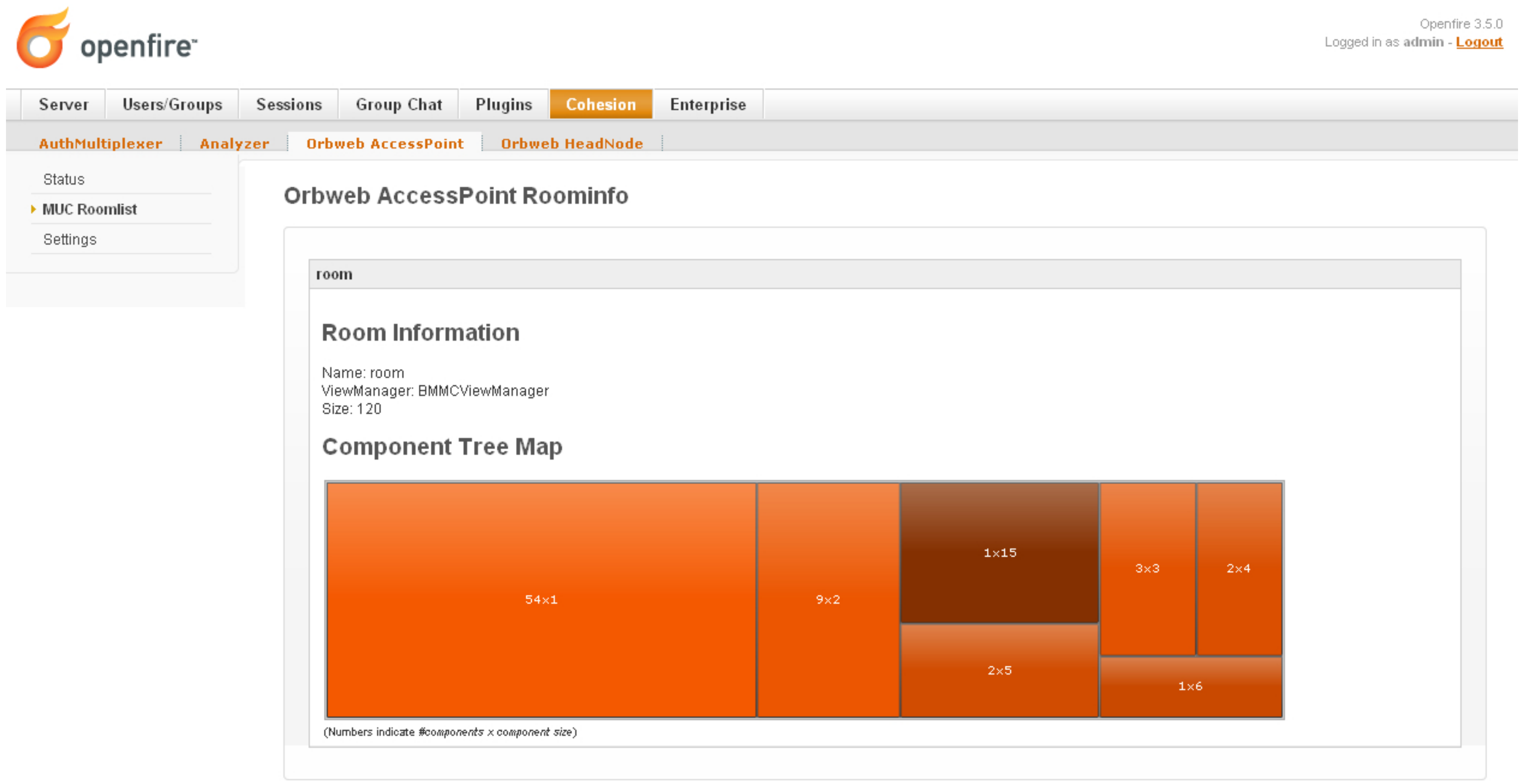
## 21.3 Topology Visualization

ORBWEB's superpeer-driven group management approach allows for easy debugging and testing of group membership models by running a number of test peers collocated on a single host. Figure 21.3 shows our topology visualization tool showing a group of 40 locally running peers managed by the *ta-pbcast* view manager as described in Chapter 19. The tool allows for dynamically adding (❶) and removing peers (❷), unicasting and groupcasting of messages (❸), the selection of view managers (❹), and the online visualization of the group's topology (❺). The latter leverages the graph visualization library *yfiles* [WEK01]. The view is dynamically updated on every change in the topology, i.e., addition and removal of peers and links. Light gray links indicate that the link partners are part of each other's local membership list (❻). Black links indicate that an E2E session exists between the link partners.



| | | Key | Inbound | | Outbound | |
|---|---|---|---|---|---|---|
| | | | Packets | Bytes | Packets | Bytes |
| ✓ | 🟥 | iq | 1753 (0,01/s) | 266895 (1,59/s) | 1773 (0,01/s) | 272013 (1,62/s) |
| ✓ | 🟧 | iq.http://cohesion.de/protocols/e2e#probe | 13063 (0,08/s) | 5310858 (31,61/s) | 13276 (0,08/s) | 3710948 (22,09/s) |
| ✓ | 🟨 | iq.http://cohesion.de/protocols/orbweb#set-vm | 65 (0,00/s) | 14141 (0,08/s) | 65 (0,00/s) | 14336 (0,09/s) |
| ✓ | 🟩 | iq.jabber:iq:roster#query | 430 (0,00/s) | 53132 (0,32/s) | 430 (0,00/s) | 53562 (0,32/s) |
| ✓ | 🟩 | iq.urn:ietf:params:xml:ns:xmpp-bind#bind | 430 (0,00/s) | 69403 (0,41/s) | 430 (0,00/s) | 84883 (0,51/s) |
| ✓ | 🟩 | iq.urn:ietf:params:xml:ns:xmpp-session#session | 430 (0,00/s) | 62149 (0,37/s) | 430 (0,00/s) | 62579 (0,37/s) |
| ✓ | 🟩 | iq.urn:xmpp:tmp:jingle#jingle | 2146 (0,01/s) | 1008901 (6,00/s) | 2126 (0,01/s) | 998404 (5,94/s) |
| ✓ | 🟩 | message | 154 (0,00/s) | 40567 (0,24/s) | 669 (0,00/s) | 134893 (0,80/s) |
| ✓ | 🟦 | presence | 1661 (0,01/s) | 203907 (1,21/s) | 64772 (0,39/s) | 29630107 (176,35/s) |

**Figure 21.1:** The ORBWEB traffic analyzer records and visualizes the overall number and the throughput rates for incoming and outgoing XMPP stanzas broken down by stanza type. The visualization is based on the *Google Chart API* and embedded into the *Openfire* administration console.



Orbweb AccessPoint Roominfo

room

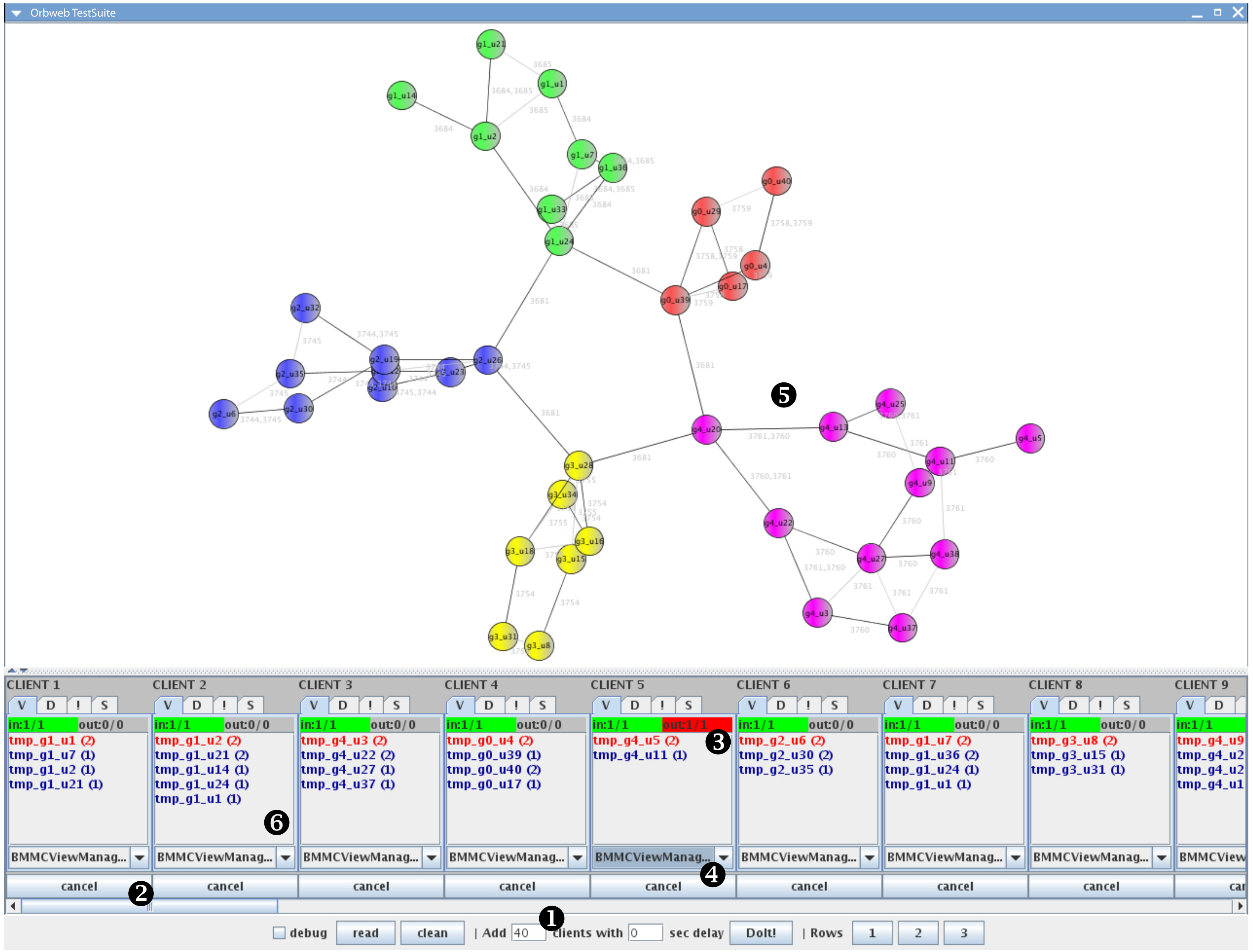
Room Information

Name: room
ViewManager: BMMCViewManager
Size: 120

Component Tree Map

(Numbers indicate *#components x component size*)

**Figure 21.2:** A snapshot of the components within a 120-peer group managed by the *ta-pbcast* view manager visualized as a tree map just after creation. As indicated by the labels, there are 54 components of size 1, 9 of size 2, 3 of size 3, 2 of size 4 and 5, and 1 of size 6 and 15. The visualization is based on *Adobe Flex* and embedded into the *Openfire* administration console.



```
CLIENT 1            CLIENT 2            CLIENT 3            CLIENT 4            CLIENT 5
in:1/1  out:0/0     in:1/1  out:0/0     in:1/1  out:0/0     in:1/1  out:0/0     in:1/1  out:1/1
tmp_g1_u1 (2)       tmp_g1_u2 (2)       tmp_g4_u3 (2)       tmp_g0_u4 (2)       tmp_g4_u5 (2)
tmp_g1_u7 (1)       tmp_g1_u21 (2)      tmp_g4_u22 (2)      tmp_g0_u39 (1)      tmp_g4_u11 (1)
tmp_g1_u2 (1)       tmp_g1_u14 (1)      tmp_g4_u27 (1)      tmp_g0_u40 (2)
tmp_g1_u21 (1)      tmp_g1_u24 (1)      tmp_g4_u37 (1)      tmp_g0_u17 (1)
                    tmp_g1_u1 (1)
BMMCViewManag...    BMMCViewManag...    BMMCViewManag...    BMMCViewManag...    BMMCViewManag...
cancel              cancel              cancel              cancel              cancel

CLIENT 6            CLIENT 7            CLIENT 8            CLIENT 9
in:1/1  out:0/0     in:1/1  out:0/0     in:1/1  out:0/0     in:1/1
tmp_g2_u6 (2)       tmp_g1_u7 (2)       tmp_g3_u8 (2)       tmp_g4_u9
tmp_g2_u30 (2)      tmp_g1_u36 (2)      tmp_g3_u15 (1)      tmp_g4_u2
tmp_g2_u35 (1)      tmp_g1_u24 (1)      tmp_g3_u31 (1)      tmp_g4_u2
                    tmp_g1_u1 (1)                           tmp_g4_u1
BMMCViewManag...    BMMCViewManag...    BMMCViewManag...    BMMCView
cancel              cancel              cancel              ca
```

**Figure 21.3:** ORBWEB test client running 40 logical ORBWEB peers within a group with a *ta-pbcast* view manager (see Chapter 19)

# 22 Performance Evaluation

This chapter presents an analysis of the results obtained from running performance tests for both the original *Smack*/*Openfire* XMPP stack and ORBWEB.

## 22.1 Evaluation Method

The testbed used for our performance evaluation consists of 41 hosts located in three different Fast Ethernet Local Area Networks connected by a campus network as depicted in Figure 22.1. 40 machines are used to host ORBWEB peers based on *Smack* v3.0.0 and the single Type IV machine hosts an ORBWEB superpeer based on *Openfire* v3.5.0. The hard- and software setup of the hosts is summarized in Table 22.1. Note that, although all hosts of the testbed run *Linux*, ORBWEB is written in Java and is thus independent from the underlying operating system. To be able to push the superpeer to its limits, we have each physical host run up to 256 peers in parallel leading to a maximum overall number of ≈ 10K peers. For test scenarios involving E2E sessions, we limit the number of collocated peers to 24 to avoid the risk of host overload. When XML encodings are compared, we use random message payloads. As random data is in general incompressible for lossless compression

| **Type** | **Hardware** | | **Software** | |
|---|---|---|---|---|
| | CPU | Memory | OS | Kernel |
| **I** | AMD®Athlon™64 X2 4600+<br>2 Cores<br>512KB Cache / Core | 3GB | Linux | 2.6.22-14-generic |
| **II** | Intel®Xeon™2.67GHz<br>2 Processors<br>512KB Cache / Processor | 2GB | Linux | 2.6.22-9 |
| **III** | Intel®Pentium™D 3.40GHz<br>2 Cores<br>2048KB Cache / Core | 2GB | Linux | 2.6.23-gentoo-r8 |
| **IV** | Intel®Core™2 Q6600 2.40GHz<br>4 Cores<br>2048KB Cache / Core | 8GB | Linux | 2.6.22-14-server |

**Table 22.1:** Hard- and software setup of the ORBWEB testbed hosts

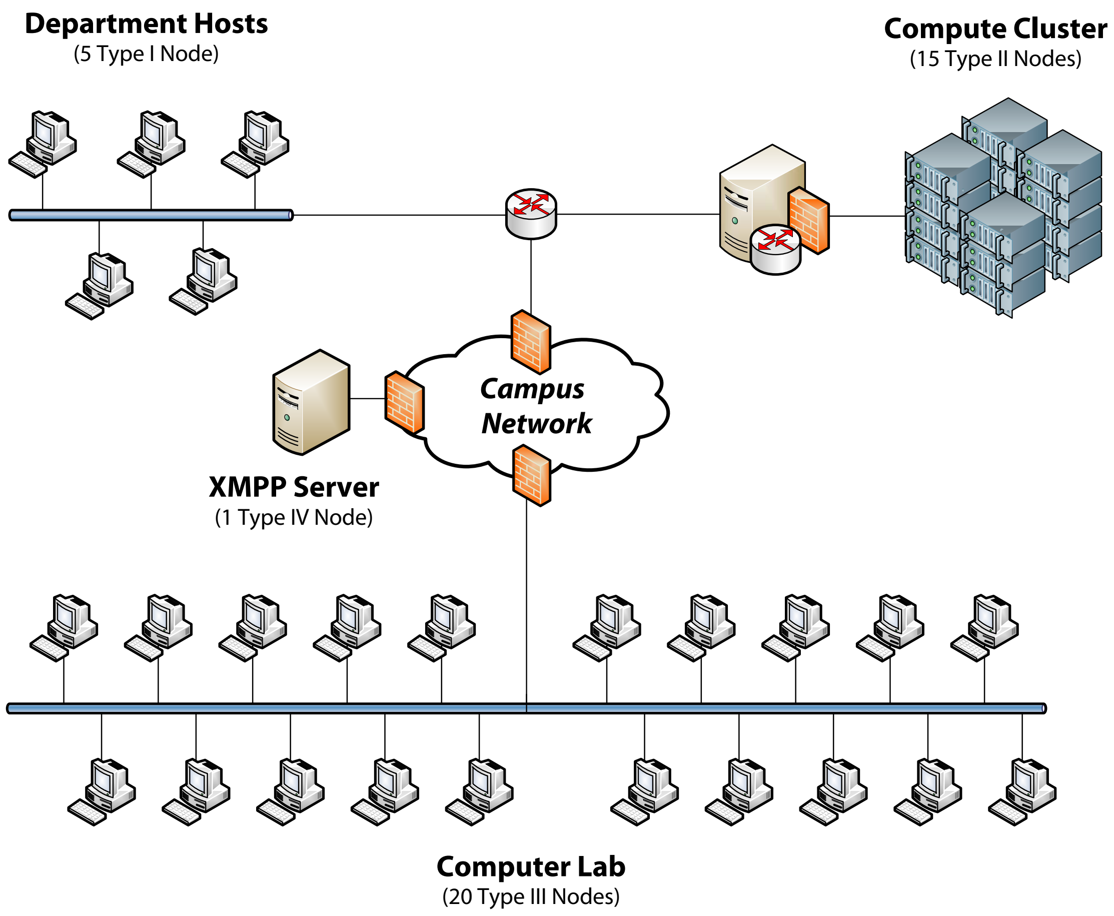


**Figure 22.1:** Topology of the testbed used for ORBWEB performance tests

algorithms[1], related results are worst-case approximations[2].

We use *The Grinder* [GA], an open source distributed load testing framework written in Java, for test deployment and orchestration. To minimize context switching overhead collocated peers are driven by threads within a single process controlled by the load injection agent of *The Grinder*. Furthermore, the online collection of statistics and results of *The Grinder* was replaced by a post-mortem approach to eliminate any interference with the actual test execution. A warm-up phase preceded each test run to eliminate the impact of *Just-In-Time* (JIT) compilation. Results are average values of 5 independent runs.

1 For random input sequences, each symbol appears with the same probability. Hence, there is no redundancy that can be exploited by a lossless compression algorithm [Sol07].

2 The resulting numbers are approximations only, because the input sequence is finite which results in a slightly non-uniform distribution for symbol occurrence.

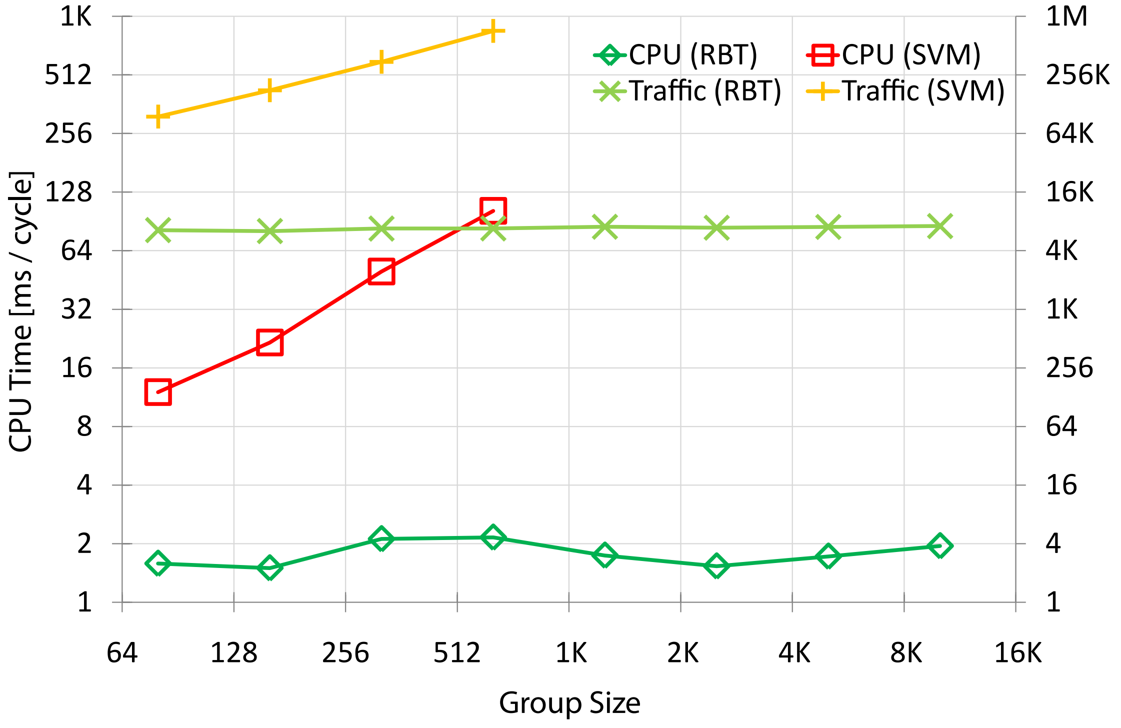
## 22.2 Membership Management

In this section, we report on the performance of the membership management operations *join* and *leave* for various group sizes. The first test consists in having one of the group members join and leave periodically (once per second). To assess the suitability of our X-MUC extension for driving large-scale groups, we compare the superpeer-side network traffic and CPU usage of both the simple original view-complete (SVM) and our RBT-based view manager (RBT) implementation with partial membership lists and a view size of 3. In order to obtain values related to a single join/leave-cycle, we calculate the quotient of the overall resource consumption within the test period and the number of join/leave cycles actually performed. The W3C encoding has been used for this test case. Figure 22.2a shows the results of our experiments. The theoretical linear message complexity of the SVM is perfectly confirmed by our measurements. Both CPU time and network traffic increase linearly reaching a maximum of 102 ms/cycle and 743 KB/cycle respectively for 640 peers. Tests with peer counts beyond 640 peers failed due to excessive memory consumption on the superpeer. From these results, we conclude that view completeness becomes infeasible for ORBWEB groups growing larger than ≈ 512 peers, where roughly 10 join/leave-cycles can be processed each second. However, this heavily depends on the actual degree of volatility. The corresponding values for the RBT view manager are 2.15 ms/cycle and 6.9 KB/cycle respectively, resulting in improvements (growing with group size) by a factor of ≈ 48 compared to the SVM view manager. Note that the manager scales up to groups consisting of 10K peers showing constant per-cycle resource usage.

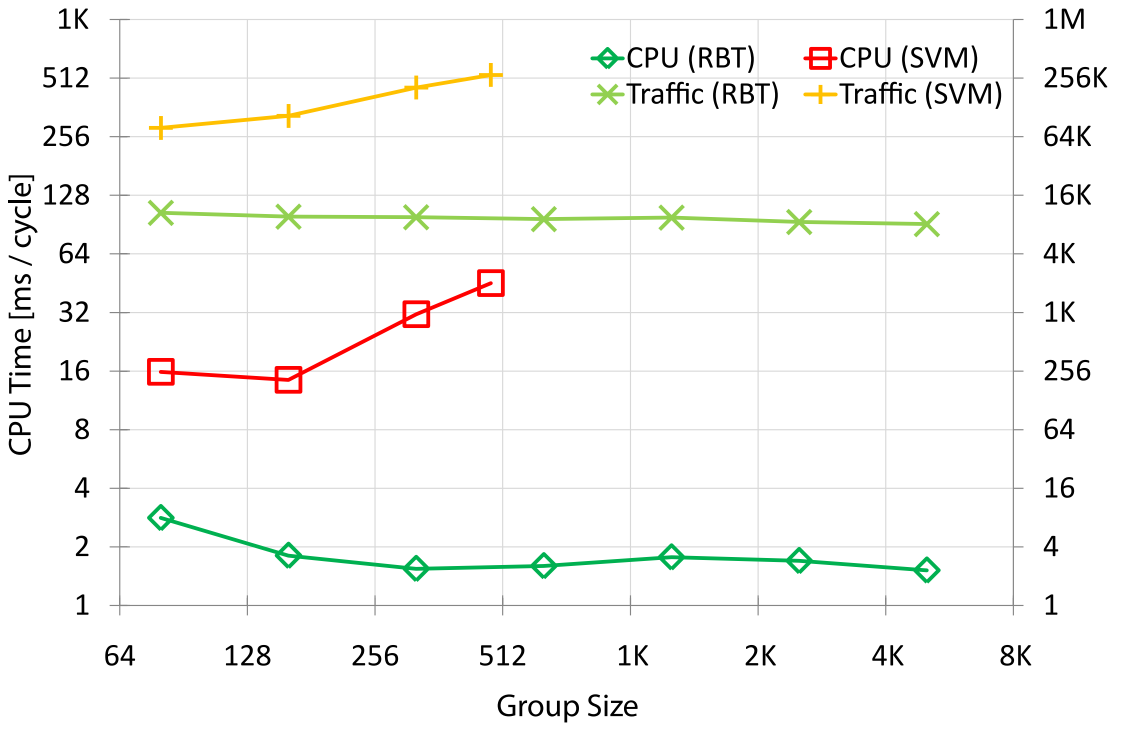
Figure 22.2b shows the results of our second test, where all group members leave and rejoin immediately every 15 seconds. Despite the extremely high churn rate of roughly 330 joins/leaves per second, the tree-based view manager exhibits constant resource usage for up-to 5K peers[1].

Figure 22.3 shows the resource usage on peers and the superpeer for the RBT view manager with respect to peer session time $T_{Session}$ within a group of 960 peers with and without *Fast Infoset* (FI) encoding enabled. Like in the previous test case, peers immediately rejoin after having left the group. Resource usage linearly decreases with increasing session time demonstrating ORBWEB's ability to operate efficiently both under extreme and real-world conditions. Resource consumption on peers is comparatively low and thus will be negligible for applications from our target application class. With average bandwidth savings of 74% on the peers and 75% on the superpeer the superior space efficiency of FI is confirmed and turns out to be even better than what could be expected from the results for general XML documents discussed in Chapter 20. While CPU usage numbers on peers are on average 14% better with FI enabled, there is a non-negligible performance penalty of 106% on average when using FI on the superpeer. This penalty can be attributed to inefficiencies related to contention in highly concurrent setups of the prototypical FI implementation used in our experiments. Nevertheless, as substantiated below, using FI significantly improves message latency and throughput despite the higher CPU usage observed in this test scenario.

1 Note that the load on the superpeer for the same group size is over 300 times higher in this scenario than in the previous scenario. Hence, the maximum group size has been limited to 5K to avoid overload.

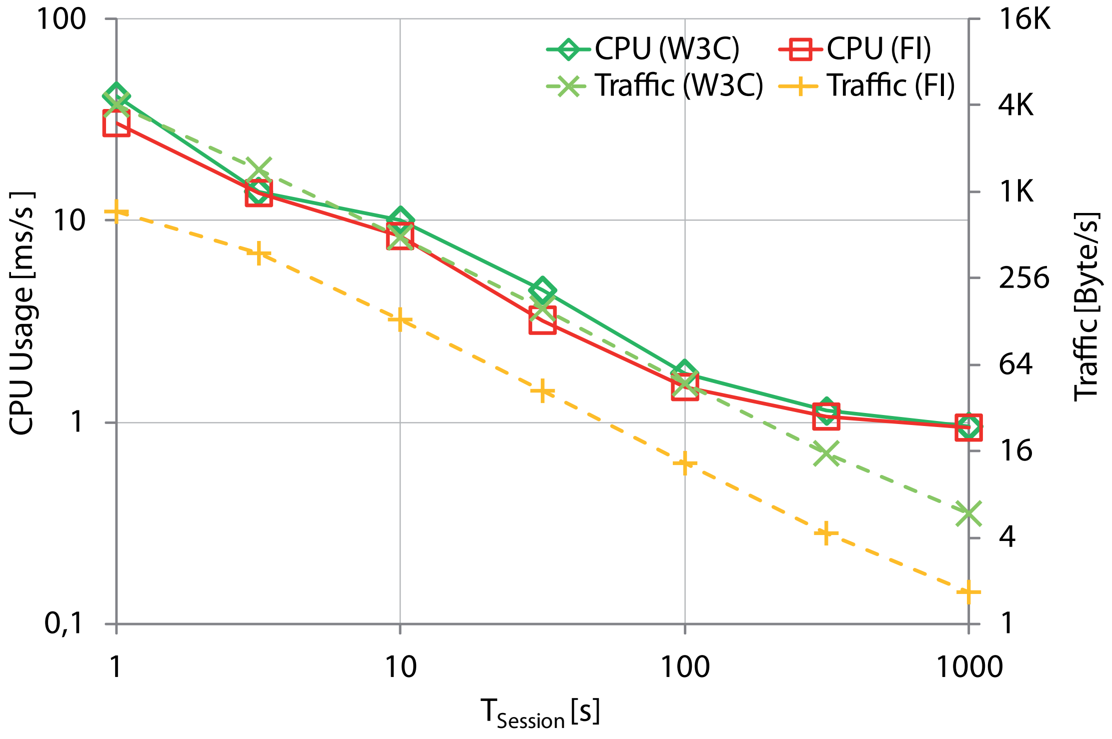


**(a)** Single volatile scenario

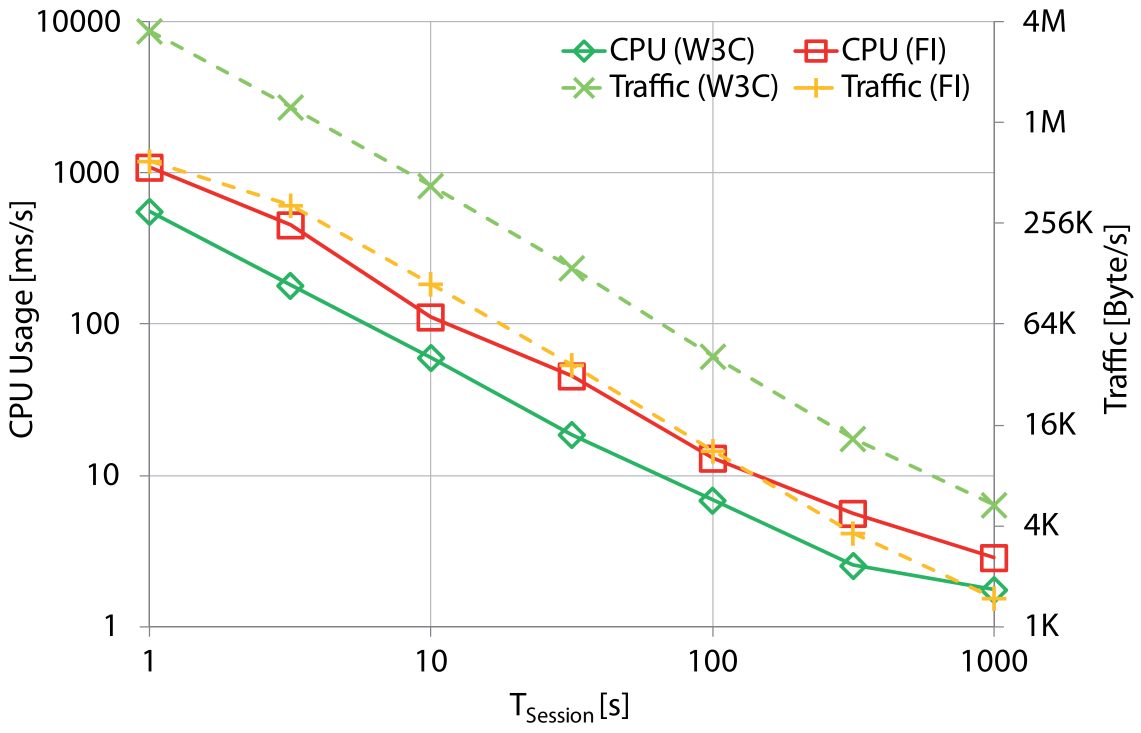


**(b)** All volatile scenario

**Figure 22.2:** Superpeer resource usage per join/leave-cycle with respect to group size



**(a)** Peer



**(b)** Superpeer

**Figure 22.3:** Resource usage for the RBT view manager with respect to session time $T_{Session}$

Whether such high degrees of volatility as covered by our experiments actually arise in real-world scenarios heavily depends on environmental conditions – like host failure rates and host usage patterns –, the application, and the aggressiveness with which idle resources are exploited. A study on resource availability in Desktop Grids [KTB+04] shows that the mean host session time in a real-world Desktop Grid is 2.8 hours for weekdays and 5.9 hours for weekends. Obtaining meaningful experimental data for such long session times is virtually impossible as perturbations dominate the resource usage actually caused by membership management operations. Examples for such perturbations are garbage collection and keep-alive messages. Their influence is visible in Figure 22.3 for session times above 100 seconds. By extrapolation from the results of our second test case, we can nevertheless estimate that the CPU utilization for a 5K group (1.51 ms/cycle) managed by ORBWEB's RBT view manager would be approximately 0.19‰ for weekdays and 0.09‰ for weekends without FI enabled and 0.39‰ for weekdays and 0.19‰ for weekends with FI enabled on our Type IV quad-core server machine. These results substantiate that ORBWEB satisfies the requirement to support a significant number of concurrent groups of considerable size for real-world churn rates and thus is well suited for realizing P2P Desktop Grids implementing the organizational and interaction models described in Part I.

## 22.3 Communication

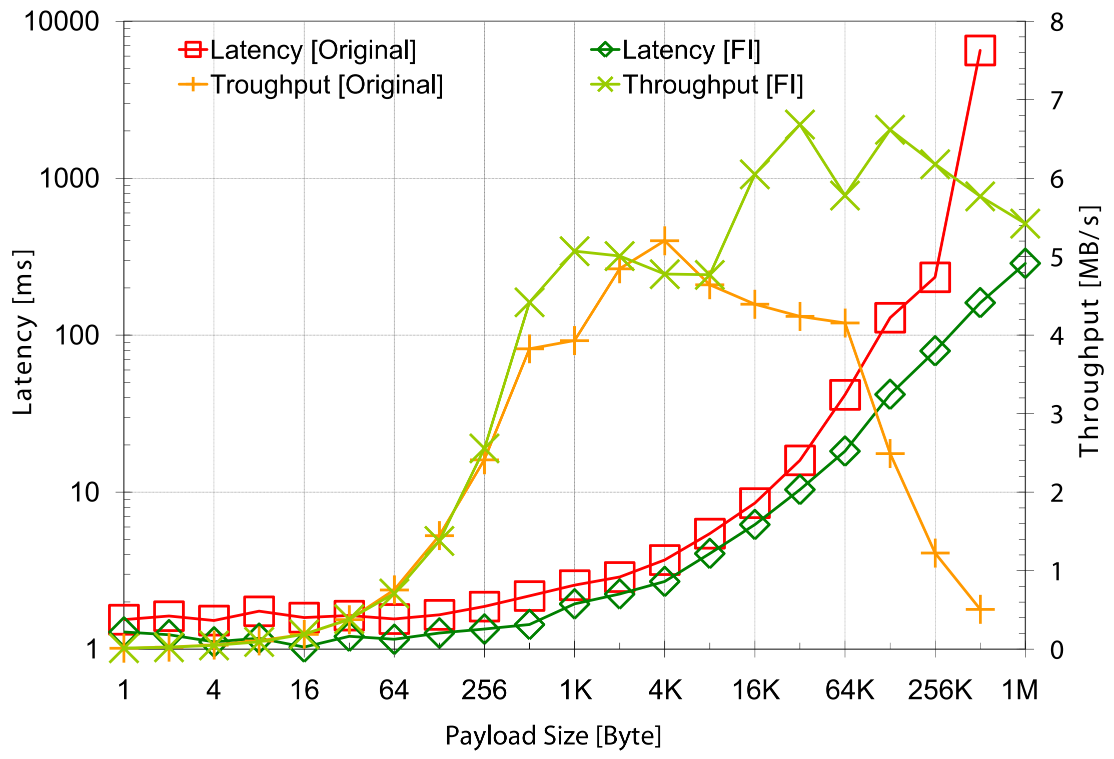
In this section, we report on the results of performance tests for the I/O subsystem performance with regard to the basic communication primitives.

### 22.3.1 I/O Subsystem Performance

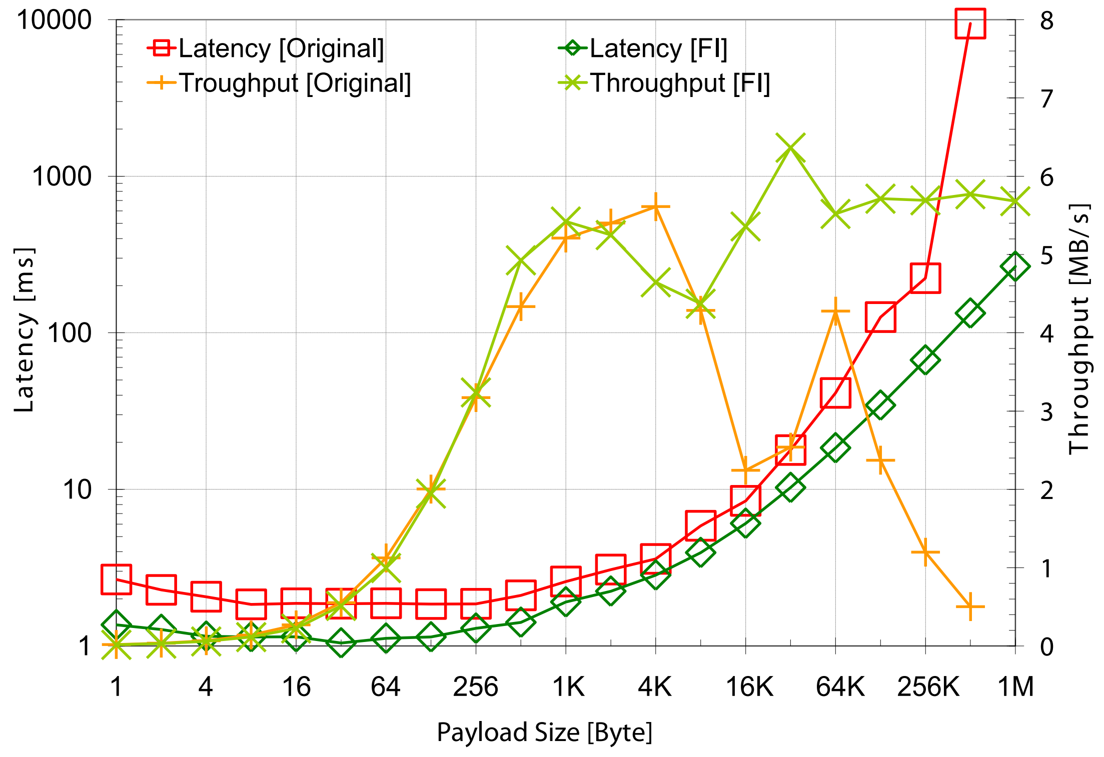
With our first set of tests we assess the performance of ORBWEB's XMPP processing pipeline by measuring throughput and latency for unicasts and groupcasts with and without FI encoding enabled. Our tests only address communication relayed over the superpeer, since E2E delivery of messages is based on plain TCP/IP. Note, that the test described subsequently is no scalability test for groupcasting, which is addressed in the following section.

The throughput test is carried out by having the sender node emit messages at maximum rate to (1) a single receiver node (unicast) and to (2) a group of size 1 (groupcast). In both cases, the single receiver acknowledges the receipt of all messages after the last message has arrived. Throughput is calculated as the quotient between transmitted payload bytes and the time elapsed between emission of the first message and receipt of the acknowledgment[1]. As can be seen from Figures 22.4a and 22.4b, *Openfire*/*Smack* apparently has problems in dealing with large messages. Due to a hardcoded server side limit on message size installed to prevent denial of service attacks, sending messages larger than 512K is impossible. For large messages between 16 KB and 512 KB our optimizations result in an improvement

1 Note that this acknowledgment is only used to indicate to the sender when all messages have been received by the receiver and is *not* related to reliability of the message transmission. In particular, no messages are held in a retransmission buffer which could potentially introduce scalability issues.



**(a)** Unicast latency and throughput



**(b)** Groupcast latency and throughput

**Figure 22.4:** ORBWEB communication subsystem performance

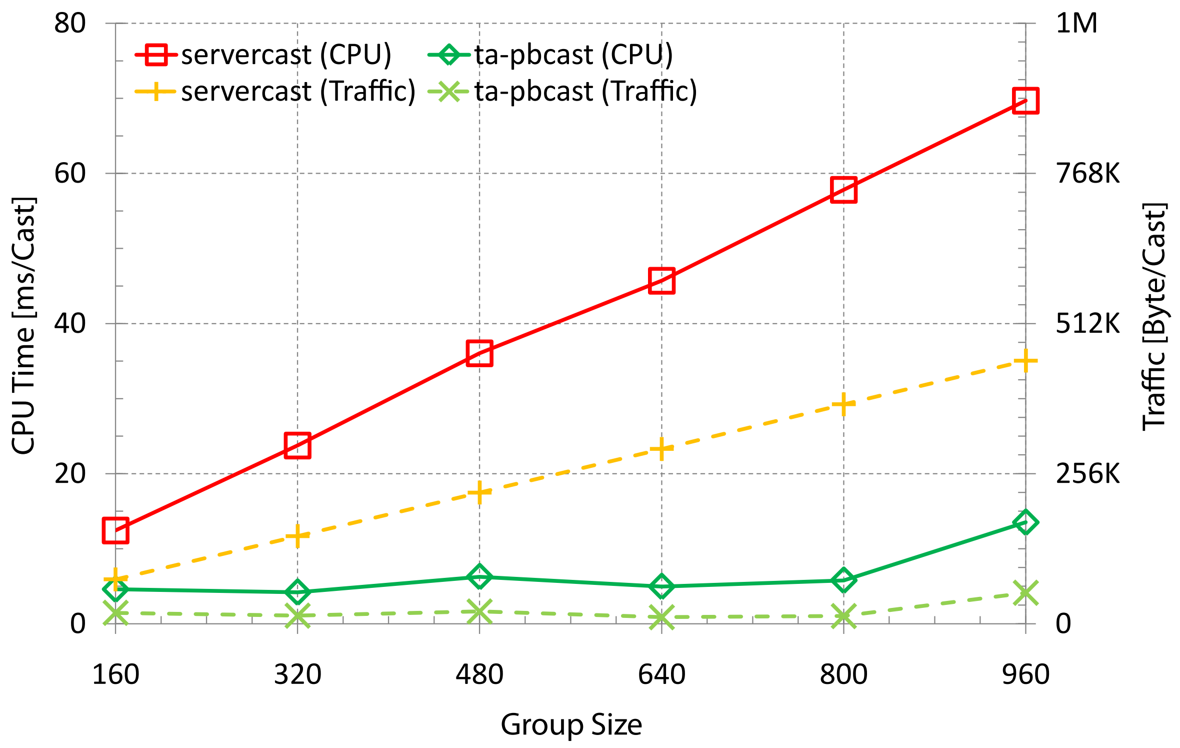
between 38% and 1038% for unicasts and between 130% and 1051% for groupcasts. With a resulting throughput of approximately 6.7 MB/s for unicast and 6.4 MB/s for groupcasts our substrate outperforms the reference implementation with 5.2 MB/s for unicasts and 5.6 MB/s for groupcasts. While our implementation is roughly on par with the reference implementation for small messages with payloads between 1 Byte and 256 Bytes, we achieve up to 29% improved throughput rates for unicasts and up to 13% for groupcasts when mid-size messages with payloads between 512 Bytes and 8 KB are transmitted. This indicates that the dissection of the input stream as described in Chapter 20 adds no significant overhead to the parsing process.

Latency measurements for both unicast and groupcast operations are carried out in a ping-pong fashion. The sender emits a message using unicast or groupcast that is echoed back by the recipient using the same kind of communication operation. As above, a group of size 1 is used in the groupcast setting. The *one-way latency* is calculated as the time between emission of the message and receipt of the echo message divided by two. Note that the latencies given here are actually two hop latencies (sender → superpeer → receiver) on the transport layer. Our optimizations result in very pronounced improvements for both unicasts and groupcasts and all messages sizes. Improvements in unicast latency are between 16% and 35% for small- and mid-size and 27% and 97% for large messages. The increases for groupcasts are between 21% and 48% for small- and mid-size and range from 28% to 99% for large messages. With 1.28 ms (unicast) and 1.36 ms (groupcasts), the minimal latencies (for a message carrying no payload) are noticeably lower for our optimized implementation than for the reference implementation (1.53 ms and 2.65 ms respectively). The unicast latency is comparable to the latency of *JXTA* sockets [AHJN05] in the case of direct communication.

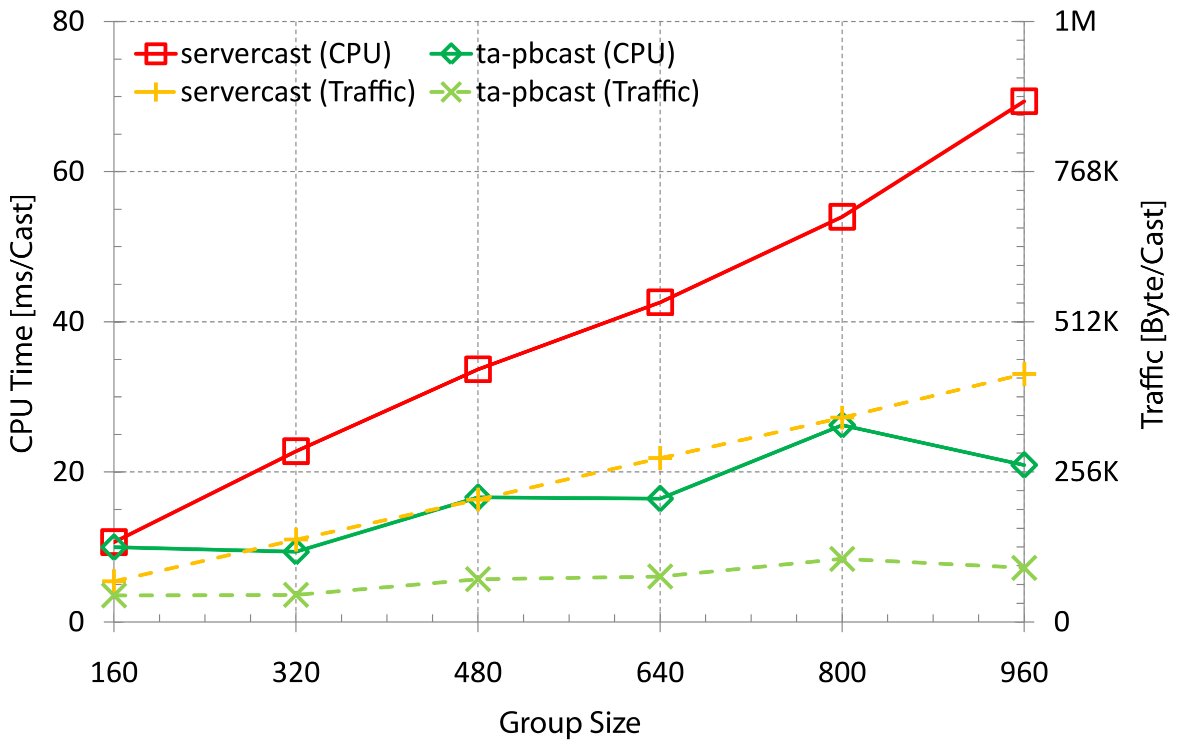
As expected, latency increases with payload size. For a 1 MB message it is roughly 300 ms in both settings, which might be considered too high for High-Performance Computing. However, the nodes of the testbed are connected by a Fast Ethernet network with 100 Mbit/s theoretical bandwidth. The actual available bandwidth within our testbed was roughly 10 MB/s between a Type I and the Type IV host. Hence, the latency for a single hop on the TCP/IP layer is 100 ms. As the message is forwarded by the superpeer not until the whole message has been received, this accumulates to 200 ms or two-thirds of the overall latency for our two-hop relayed 1 MB message transmission. Thus, the time to run through the ORBWEB protocol stack twice, i.e., down the stack at the sender, up and down on the superpeer, and finally up on receiver, only takes a reasonable 100 ms. In case E2E links are used the latency is reduced by half. Also messages as large as 1 MB are considered to be rather untypical for message passing applications. As described in Chapter 36, COHESION applications may use an alternative mechanism based on the file sharing protocol *BitTorrent* [bit] to disseminate larger payloads to all members of a group.

### 22.3.2 Groupcast Scalability

In this section, we assess the scalability and performance of the standard *Openfire*/*Smack* groupcast (*servercast*) in combination with the blind view manager and ORBWEB's Topology-Aware Bimodal Multicast (*ta-pbcast*). Note that all messages were reliably delivered for both groupcast implementations in both test scenarios described subsequently.



**(a)** Non-volatile setup



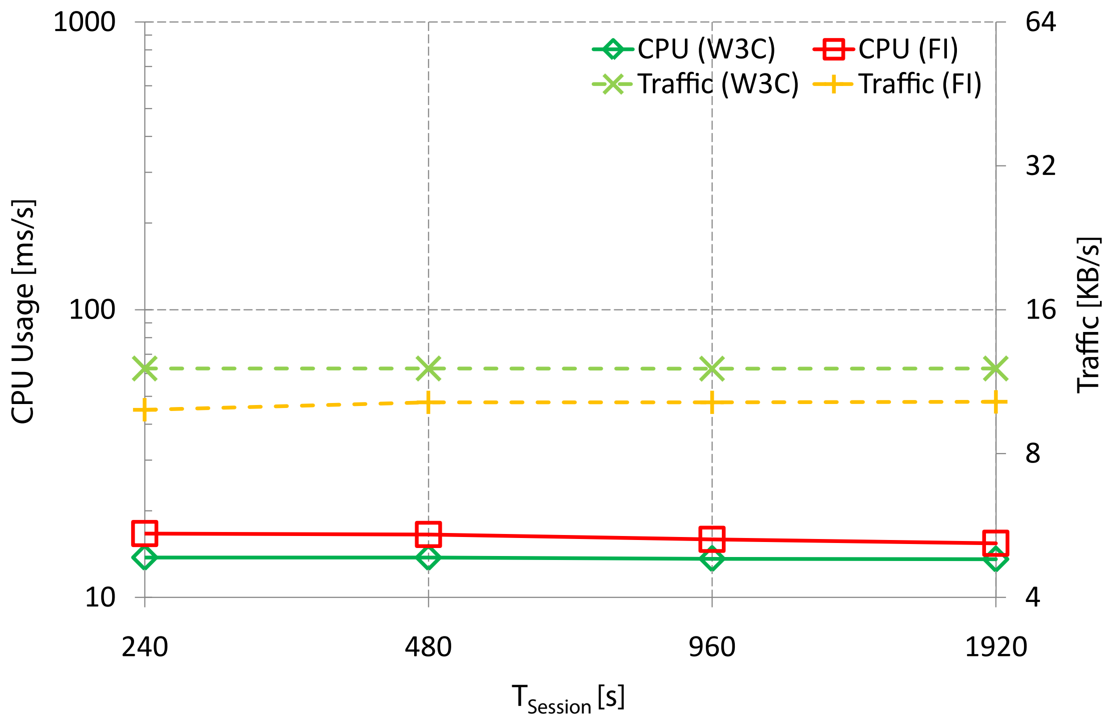
**(b)** Volatile setup

**Figure 22.5:** Resource usage on the superpeer per groupcast operation

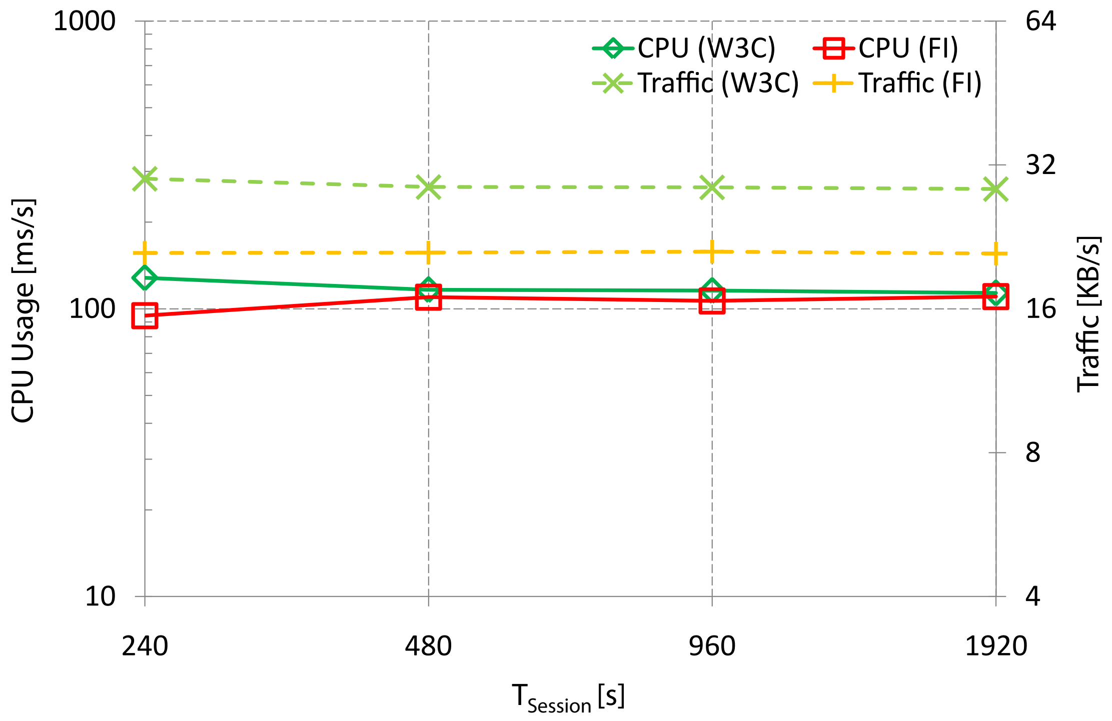
The first test scenario consists of a master peer groupcasting a message with 256 bytes of random character payload five times a second within a static (Figure 22.5a) and a volatile (Figure 22.5b) group of variable size. In the volatile setup 12.5% of the peers were configured to join and leave periodically, i.e., being offline for 30 s and then online for 30 s in an alternating manner. The progressions of CPU utilization and bandwidth usage are linear with respect to the size of the group for the servercast and are almost identical for both the static and the volatile setup. This is not surprising as peer joins and departures cause virtually no overhead when the blind view manager is used. The numbers for the volatile setup are even slightly better as the effective group size is smaller due to half of the volatile peers being offline at any given point in time on average. With 70 ms per groupcast for 960 peers our Type IV quad-core machine could ideally handle message rates of up-to 57 groupcasts per second. However, roughly 450 KB per groupcast operation would result in an overall bandwidth usage of over 25 MB/s requiring Gigabit Ethernet. For message sizes larger than 1 KB, bandwidth becomes the limiting factor rendering the superpeer-based groupcast unsuitable for large-scale applications requiring sustained high message rates.

In contrast, the superpeer-side costs per groupcast for ta-pbcast are almost constant at ≈ 5 ms per groupcast with a small linear overhead caused by component detection that executes in parallel. This results in a projected maximum message rate of 296 groupcasts per second in the static setup. The linear fraction for component detection is more pronounced for the volatile setup as the constant flux in membership triggers more E2E session negotiations. Another reason for the increased resource utilization is the small period of time that elapses until the peer is assigned to a component. During this period groupcast messages are delivered by the superpeer. As the number of such unassigned peers increases linearly with group size, this adds another linear component to the resource usage numbers. Nevertheless, the cost of groupcasting is still significantly lower than that for servercast. With 21 ms per groupcast the projected maximum message rate within a 960 peer group is 190 groupcasts per second.

Figures 22.6 and 22.7 show the resource usage for servercast and ta-pbcast on peers and the superpeer with respect to peer session time $T_{Session}$ within a group of 480 peers with and without Fast Infoset (FI) encoding enabled. In contrast to the previously described test scenario, this time all but one of the peers are configured to be volatile and to rejoin right after having left the group. A single stable master peer groupcasts messages with 1 KB of random character payload ten times a second. Both groupcast implementations work well in lightly to highly volatile setups. Resource usage is constant over the full range of mean session times both on peers and on the superpeer for the servercast and on the peers for ta-pbcast. As in the first test scenario resource usage on the superpeer for ta-pbcast is influenced by the overhead of the component detection subprotocol, the time required to detect to which component an arriving peer belongs to, and the extra work of E2E session establishment after reconfigurations of the red-black trees maintained within components. Towards lighter volatility scenarios these overheads decrease rapidly, resulting in moderate resource usage on the superpeer that is within the same order of magnitude as the respective resource consumption on peers. As expected, there is a substantial shift of load from the superpeer to the peers when using ta-pbcast instead of servercast: CPU usage drops by up to roughly 78% (W3C) and 86% (FI) for lighter volatility scenarios, traffic by between 73% and 94% for the W3C encoding and between 93% and 98% for FI. This allows the

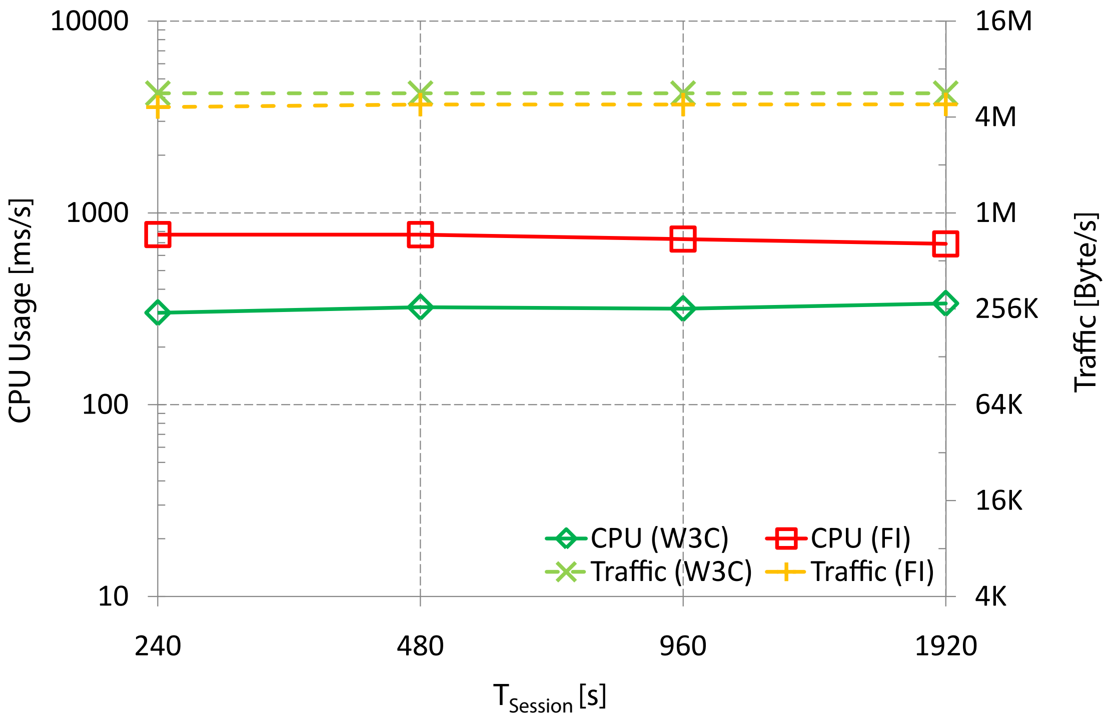


**(a)** Servercast

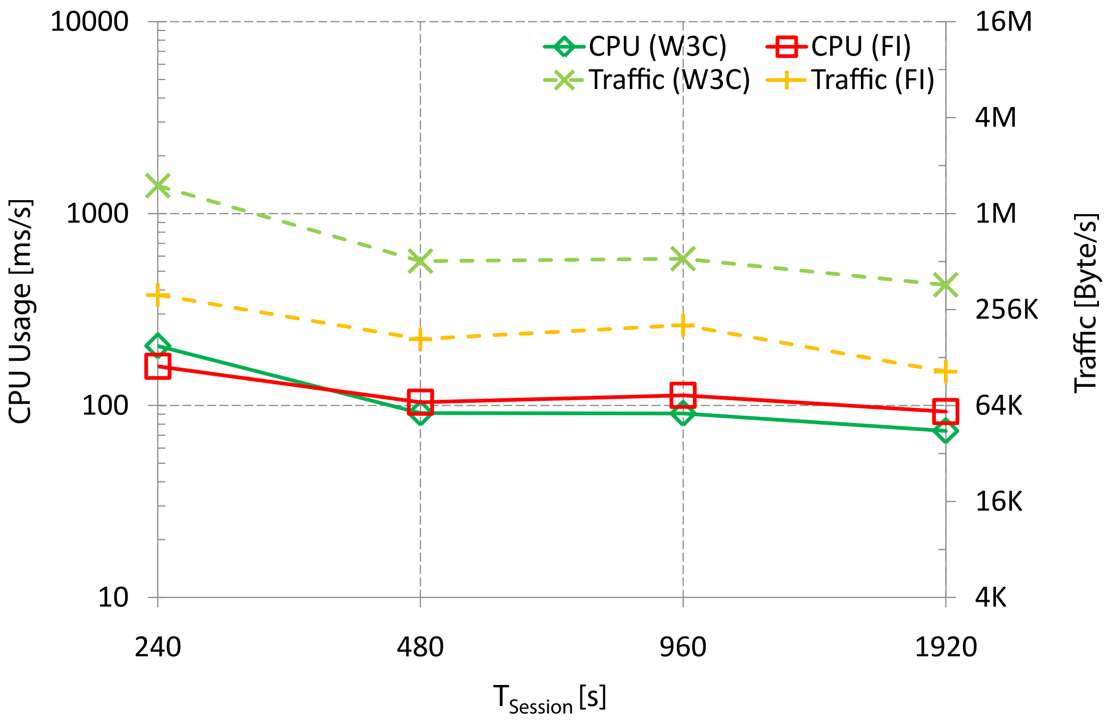


**(b)** Ta-pbcast

**Figure 22.6:** Resource usage on peers for the superpeer-based groupcast and the *ta-pbcast* with respect to session time with (FI) and without (W3C) Fast Infoset encoding enabled



**(a)** Servercast



**(b)** Ta-pbcast

**Figure 22.7:** Resource usage on the superpeer for the superpeer-based groupcast and the *ta-pbcast* with respect to session time with (FI) and without (W3C) Fast Infoset encoding enabled

superpeer to handle significantly higher groupcast rates and/or group sizes. Using FI yields considerable bandwidth savings for both servercast (15%-18%) and ta-pbcast (27%-30% on peers and 62%-79% on the superpeer respectively). The large difference between the savings for servercast and ta-pbcast is caused by differences in the distribution of message types transmitted during the experiments: While the servercast almost exclusively transmits groupcast messages with random payload that has not much potential for compression, ta-pbcast additionally transmits large quantities of highly-redundant signalling (Jingle) and component detection messages (probes). As already diagnosed in the membership management experiments described above, the FI XML processor used to conduct the experiments performs badly under certain conditions. Besides high contention scenarios (described above), this includes setups with poorly compressible XML streams. The latter in particular affects CPU usage numbers for the servercast resulting in degradation between 105% and 156%.

# 23 Related Work

This section gives an overview of alternative technologies suitable for implementing (part of) the functionality provided by ORBWEB and justifies the decision to build a substrate for P2P Desktop Grid Computing based on XMPP by discussing a number of projects from the domain of distributed computing that have already migrated to XMPP, however, without addressing its shortcomings.

## 23.1 Alternative Substrate Technologies

Many structured P2P systems have been proposed that implement packet routing in a fully distributed way [SMK+01, ZHS+03, RD01b, BBK+07, RFH+01]. These systems are able to deliver packets associated with a destination identifier to the peer with an identifier closest (with respect to some metric) to that identifier. This process typically involves forwarding the packet in a multi-hop fashion along the edges of an overlay network gradually approaching the destination peer. Such overlay routing systems can be used to implement *Distributed Hash Tables* (DHTs). In principle, these systems can be used to implement part of ORBWEB's membership management functionality in a scalable way by employing the overlay neighbor sets to establish local membership lists. However, such a solution suffers

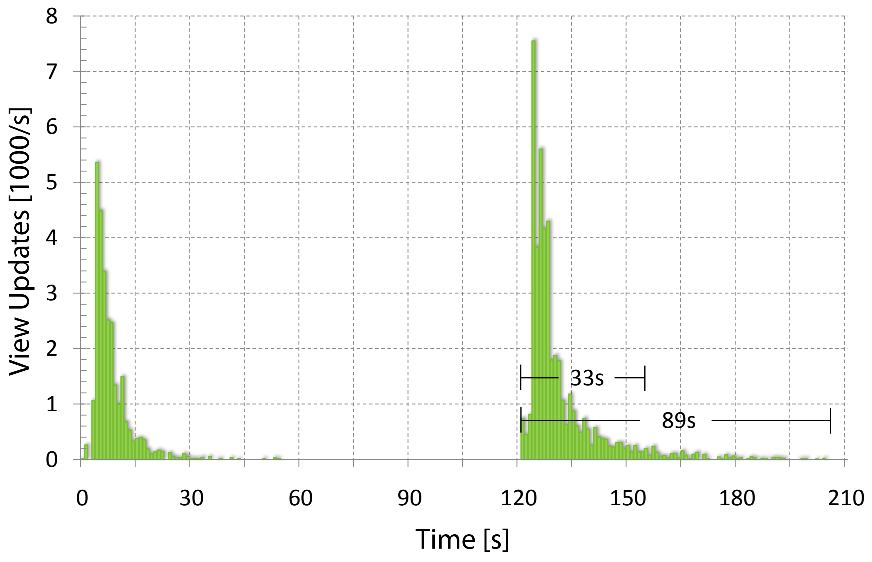


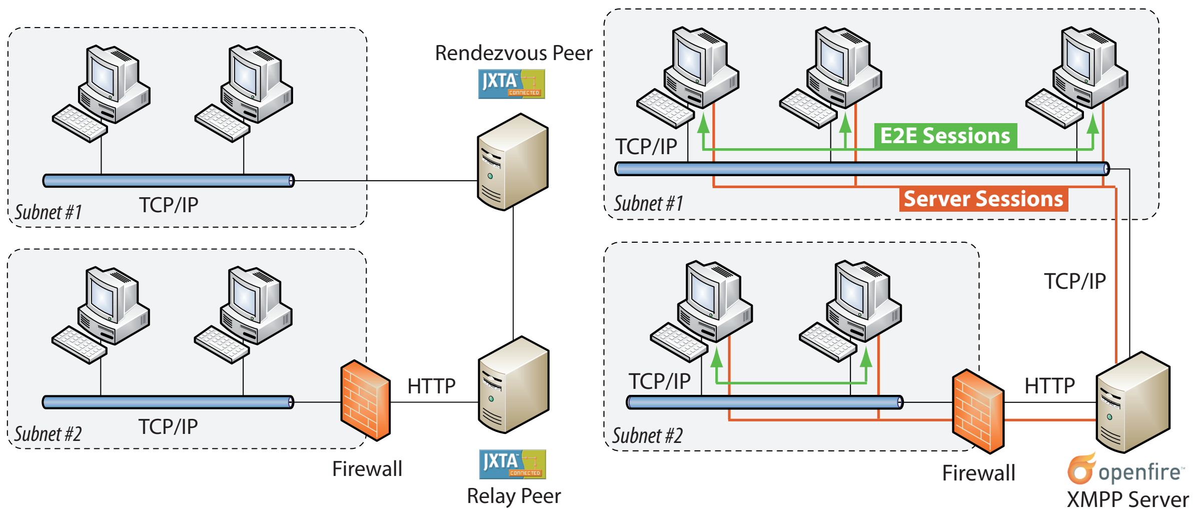
**Figure 23.1:** Number of membership view updates emitted by the superpeer on creation of a *Chord* ring consisting of 640 peers and on addition of 640 additional peers. The latter takes 33s for the 95th percentile of updates and 89s until completion.

from several shortcomings with respect to our requirements: First, only a single/limited number of virtual topologies can be provided. Second, multihop routing necessarily leads to increased communication latency. As low latency is of prime importance in High Performance Computing, messages in our architecture are routed in a single hop for directly connected peers and in two hops for peers connected to the same superpeer. Third, the churn resistance of overlay routing networks is limited as indicated by numerous evaluations conducted recently. For example, the routability of *BruNet* [BBK+07] drops to 84% when mean session time falls below 6 minutes. Additionally, restoration of routability after massive node arrivals or departures happens relatively slowly: *BruNet* needs 11 minutes to restore full routability after an insertion of 450 nodes to a fully routable network of 460 nodes [BBK+07]. Similar findings exist for *Tapestry* [ZHS+03], which needs 10 minutes to restore 95% routability after inserting 200 nodes into an existing 325-node network. Although testbed setups are not identical, the fact that ORBWEB needs only 89 seconds to restore a perfect *Chord* ring (see Chapter 18) after adding 640 nodes to an existing 640-node network (see Figure 23.1) indicates that using a hybrid implementation based on a superpeer-managed virtual topology is a promising approach, when performance is considered more important than utmost scalability. Note, that routability virtually remains unaffected when nodes join or leave the network as long as the superpeer is not overloaded. This results from the fact that predecessor and successor nodes are assigned/updated immediately when nodes join or leave the group (cf. Chapter 18).

The ongoing *Spontaneous Virtual Network* (SpoVNet) [BHMW11] project develops a platform for service overlays that can be setup with little effort and no manual configuration. SpoVNet's communication library called *Ariba* [HMM+10] can handle mobility, multi-homing, and security transparently. *Ariba* creates an overlay using a variant of *Chord* and is able to dynamically establish end-to-end connections between peers regardless of their location over various network protocols. A hierarchical broadcast system called *MCPO* [HMW10] closely related to the *NICE* [BBK02] protocol allows for one-to-many communication. However, *Ariba* does not yet support peer groups neither as a modeling tool nor for providing a multicast scope. Furthermore, there is no large-scale performance evaluation available, yet.

The problem of high end-to-end communication latency in multi-hop overlays can be alleviated by creating superpeer-assisted overlays. Merz *et al.* [MWSP08] propose a self-organizing superpeer overlay for Peer-to-Peer Desktop Grids. Edge peers are dynamically elected to become superpeers on demand and downgraded when they are no longer needed. Superpeers are interconnected by a *Chord* [SMK+01] overlay optimized for low multi-hop latency using network coordinates [CDK+04]. The proposed system is able to reduce the average end-to-end latency between peers of a $\approx 400$ node network by 50% compared to standard *Chord* resulting in twice the value of a fully meshed network. The evaluation is performed by simulation based on live data from *PlanetLab* [CCR+03].

*JXTA* [TAA+03] is an open source initiative, sparked and maintained by Sun Microsystems. Its primary goal is to provide a foundation for interoperable P2P applications. *JXTA* consists of a set of six language- and platform-independent protocol specifications. It provides basic services for generic P2P applications including peer group organization, inter-peer communication, and resource discovery. Although *JXTA* is the most advanced P2P library currently available and is in general considered to be a good choice for implementing distributed computing platforms [AHJN05], performance studies have revealed weaknesses in the Java



**(a)** *JXTA* overlay network with rendezvous and relay infrastructure peers

**(b)** ORBWEB network with E2E sessions and a superpeer

**Figure 23.2:** *JXTA* / ORBWEB infrastructure comparison

implementation of the version 2.0 protocol specification. Criticism is related to high pipe latency and low throughput for small messages [HD03], low reliability of TCP connections [Sei], as well as rendezvous network instability [BGNT04] and slow convergence [ACJD07]. Apart from the fact that these shortcomings, which are specific to the reference implementation, exist, the overall *JXTA* design is not optimized for P2P Desktop Grid Computing. In particular, it is not tailored for High Performance Computing applications. Resource discovery for example, offers no guarantee about neither the number of advertisements it will discover, nor the time the discovery is about to take [Rat08]. However, both *JXTA* and XMPP provide similar functionality on the lower protocol layers dedicated to communication. With the addition of E2E sessions, the ORBWEB network infrastructure becomes very similar to the infrastructure of the *JXTA* [TAA+03] overlay network (see Figure 23.2): The ORBWEB superpeer is, like the *rendezvous peer* in *JXTA*, responsible for delivering groupcast messages to all connected peers. If no E2E connection can be established due to NAT devices or firewalls, unicast messages are delivered by the ORBWEB superpeer. This is similar to the responsibility of *relay peers* in the *JXTA* network architecture. Thus, it would be certainly possible to implement the functionality of ORBWEB on top of these layers. However, due to its wide distribution and maturity, we think XMPP is still the better choice.

*Peer-to-Peer simplified* (P2PS) [Wan05] is an open-source project providing an infrastructure for P2P service discovery and pipe-based communication. The *P2PS* reference implementation is written in Java. While sharing many concepts with *JXTA*, *P2PS* is more focused on simplicity than on feature richness. *P2PS* peers can communicate over multiple protocols that can be replaced transparently to the application. P2PS service discovery is based on XML advertisements and queries and uses subnets to broadcast advertisements and queries efficiently. As in *JXTA*, rendezvous peers are responsible for caching and forwarding advertisements and queries to rendezvous' in other subnets. Although *P2PS* provides a peer group abstraction, there is no support for groupcasting. Thus, a substantial requirement for many distributed algorithms is not satisfied. To our knowledge there is no performance evaluation available for *P2PS*.

## 23.2 XMPP Technology Adopters

*OurGrid* [CBA+06] is a platform for Desktop Grid Computing. Former versions of *OurGrid* used *Remote Method Invocation* (RMI) over TCP/IP as the programming model. As RMI suffers from poor performance, due to large protocol overhead and also does not integrate well with restrictive firewalls and NAT devices, *OurGrid* employs an asynchronous programming model called *JDIC* [LCBF06] that is based on XMPP since version 4.

The *Distributed Infrastructure with Remote Agent Control* (*DIRAC*) [TGSR04] is a *Service Oriented Architecture* (SOA) composed of lightweight services forming a scalable robust Grid Computing environment to manage and track a large number of computing tasks. Since *DIRAC* is primarily targeted at High Throughput Computing applications, it does not pose the same high demands on the communication infrastructure as P2P Desktop Grid Computing focused on High Performance Computing does. XMPP is used to implement three different aspects of the system: inter-service messaging, state monitoring of agents, and job-level monitoring. While plain XMPP is suitable for the first two applications as the number of services (5-20) and agents (10-100) is comparatively small, the third is more critical as thousands of jobs are active at peak time. In contrast to our approach of improving and extending the XMPP protocols, *DIRAC* restricts itself to protect the system from the impact of overloaded XMPP servers by applying virtualization techniques. By adopting ORBWEB, *DIRAC* could probably avoid potential overload situations as most inter-peer traffic could be exchanged over E2E sessions.

*Xeerkat* [Mil05] is a Grid economy platform based on dynamically reconfigurable networks of agents that offer and consume services. Due to better infrastructure support and much easier setup [Mil], *Xeerkat* migrated from *JXTA* to XMPP in the 2.0 release.

The *Friend-to-Friend Computing framework* [NKVB08] project envisions so called *F2F Grids* or *Frids*. The basic idea of frids is to achieve increased ease of use compared to traditional approaches to Grid Computing. Frids allow to start a parallel application quickly with minimal administrative effort. To reach this goal, the authors propose among other things to use XMPP as the enabling communication and coordination platform. Although frids are simple to setup by exploiting existing instant messaging infrastructure, their usability for solving demanding computational problems has not yet been evaluated at a large scale.

# Part VI

# Capability-Aware Information Aggregation*

Information aggregation is the process of summarizing information across the nodes of a distributed system. In this part, we present a hierarchical information aggregation system tailored for Peer-to-Peer Desktop Grids whose resources typically exhibit a high degree of volatility and heterogeneity. Aggregation is performed in a scalable yet efficient way by merging data along the edges of a logical self-healing tree with each inner node providing a summary view of the information delivered by the nodes of the corresponding subtree. We describe different tree management methods suitable for high-efficiency and high-scalability scenarios that take host capability and stability diversity into account to attenuate the impact of slow and/or unstable peers on aggregation accuracy. As existing approaches try to distribute the onus of aggregation evenly among the nodes of the system, this *capability-awareness* is the most essential distinguishing feature of the proposed aggregation system.

## Related Publications

[SBH09] Schulz, Sven; Blochinger, Wolfgang and Hannak, Hannes: **Capability-Aware Information Aggregation in Peer-to-Peer Grids – Methods, Architecture, and Implementation**. *Journal of Grid Computing* (2009), Bd. 7(2):S. 135–167

∗ This part contains a summary of the respective topic. For further details the reader is referred to the related publications.

# 24 Overview

Fundamental distributed paradigms and algorithms are based on or can be implemented using information aggregation [vR03]. This includes leader election, voting, service and resource placement, multicast tree formation, and error recovery. Thus, an information aggregation service can serve as a basic building block for the design of sophisticated Peer-to-Peer Desktop Grid components. Despite of its importance, there are numerous open challenges in the design and implementation of aggregation systems that are considered worthy of future research [RM06]. Especially, the cost of reconfigurations caused by high node volatility can become significant if the variance in performance and stability features of peers are not taken into account.

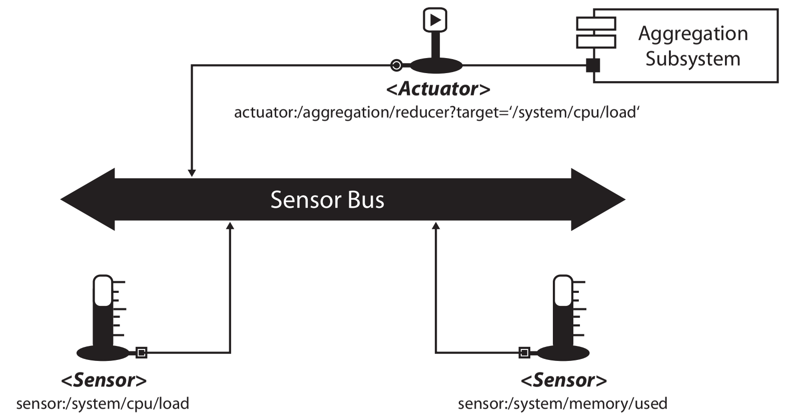
The information aggregation system described in this part complements and completes the functionality of ORBWEB by providing a generic many-to-one communication primitive (known as a *reduction* in parallel computing). While the essential features and properties of the system are discussed, many interesting aspects had to be omitted for the sake of brevity. The reader is referred to [SBH09] for a thorough in-depth discussion of the systems architecture, the aggregation tree management methods, and our implementation within the COHESION platform.

## 24.1 Architecture

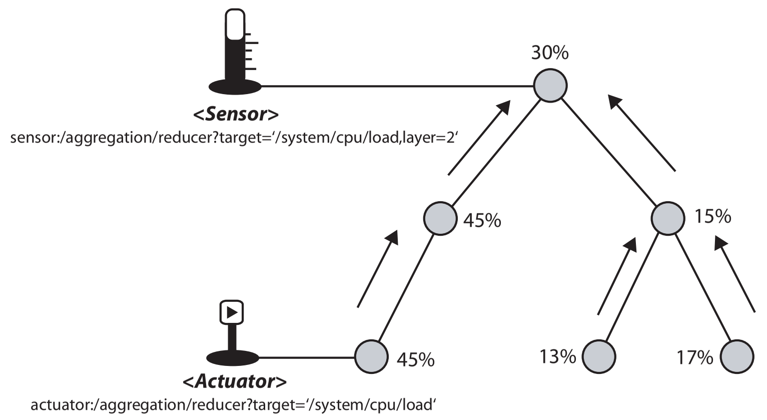
The process of information aggregation in COHESION can be decomposed into three phases (see Figure 24.1): data gathering, data aggregation, and distribution of the aggregated data.

Data gathering is addressed by providing an extensible sensing framework. The central abstraction is a *sensor bus* that leverages the microkernel design of COHESION to allow modules to create and deploy custom sensors to capture system state (see Figure 24.1a). The sensor bus actively schedules measurements using a dedicated measurement scheduler, thus enforcing resource limitations configured by the host owner and preventing duplication of measurements when aggregation of one and the same system property is performed simultaneously on behalf of several modules.

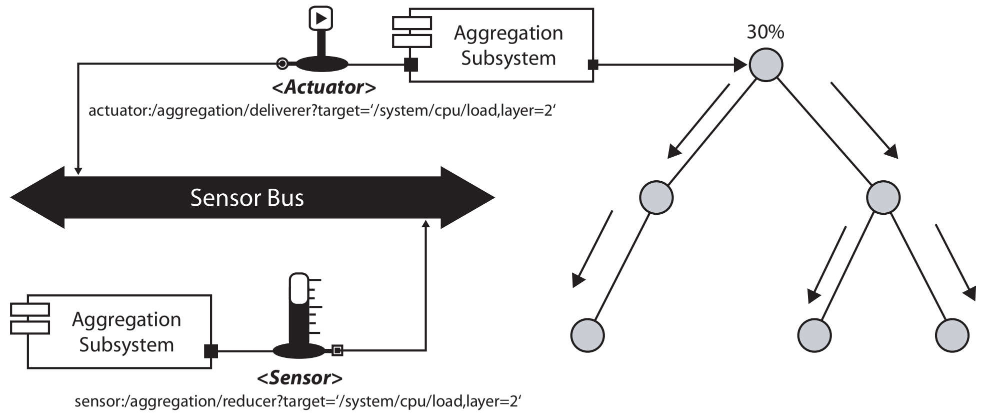
Data aggregation is done in an efficient and scalable way along the edges of a self-healing logical *aggregation tree* (see Figure 24.1b). The aggregation tree spans all peers within a logical partition of the Desktop Grid. These logical partitions are called *aggregation groups* and are mapped to ORBWEB peer groups to guarantee isolation. The aggregation tree consists of *reducers* collecting values from its children and providing aggregated values to their parent. To use the aggregation infrastructure most efficiently, a single shared reducer network per sensor is maintained that is used to satisfy an arbitrary number of parallel aggregation requests and a single aggregation tree per application that is shared among all


Aggregation Subsystem
<Actuator>
actuator:/aggregation/reducer?target='/system/cpu/load'
Sensor Bus
<Sensor>
sensor:/system/cpu/load
<Sensor>
sensor:/system/memory/used

**(a) Phase I**: *Information gathering*. Sensor values are delivered over a sensor bus to the aggregation subsystem. The set of available metrics can be easily extended by deploying custom sensors.


30%
<Sensor>
sensor:/aggregation/reducer?target='/system/cpu/load,layer=2'
45%
15%
45%
13%
17%
<Actuator>
actuator:/aggregation/reducer?target='/system/cpu/load'

**(b) Phase II**: *Information aggregation*. Sensor values from peers within the aggregation group are summarized along the edges of an aggregation tree providing increasingly condensed information towards the root.


Aggregation Subsystem
30%
<Actuator>
actuator:/aggregation/deliverer?target='/system/cpu/load,layer=2'
Sensor Bus
Aggregation Subsystem
<Sensor>
sensor:/aggregation/reducer?target='/system/cpu/load,layer=2'

**(c) Phase III**: *Delivery of aggregated values*. Aggregate values are fetched by an actuator that, e.g., reuses the aggregation tree to efficiently deliver the aggregate values to all peers in the aggregation group.

**Figure 24.1:** Phases of the information aggregation process

reducer networks.

Tree management is designed to be customizable: Different providers can be plugged-in to realize custom tree management strategies. We use this feature to realize strategies for two use cases: highly performant and highly scalable groups. Mechanisms to reduce the impact of volatility and heterogeneity of resources are incorporated into both strategies by considering specific capabilities of hosts (performance, stability, or quality of network connection) when assigning a position within the reduction network.

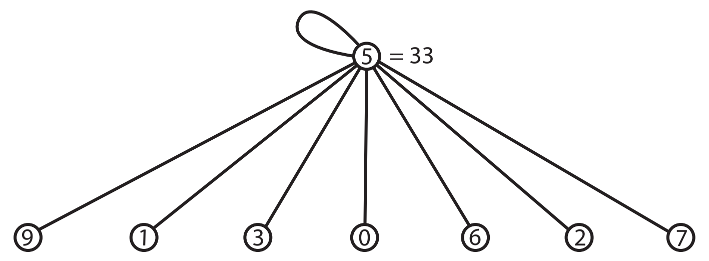
Finally, delivery of sensor/reducer data within the reduction network is accomplished by allowing to probe remote sensors across network segmentations in a fully transparent secure fashion. This is achieved by leveraging the overlay network provided by ORBWEB. Delivery of aggregate values (see Figure 24.1c) can be accomplished in an application sensitive way using a broad spectrum of strategies, including propagation along the aggregation tree or by using a groupcast protocol provided by the platform. Because of this flexibility, the system supports a wide variety of applications with diverse requirements.

## 24.2 Tree Maintenance

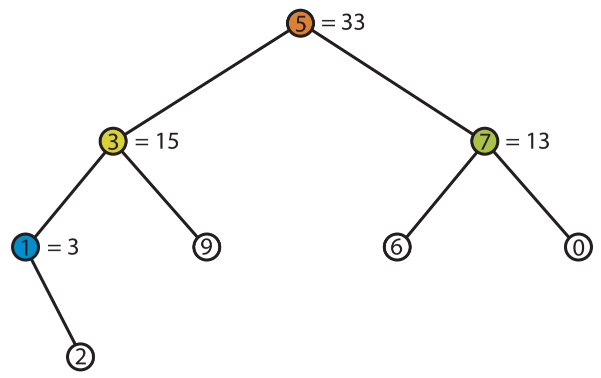
Naïvely, information aggregation could be done in a Client/Server fashion by having a particular host fetch values from all other hosts in the system (see Figure 24.2a). Obviously, such a centralized approach isn't scalable, since the server would have to handle a vast number of messages. Hence, in more sophisticated systems aggregation is performed in a Peer-to-Peer fashion by merging data along the edges of a spanning tree [YD04], which covers all the hosts in the system (see Figure 24.2b). While this approach distributes the onus of aggregation over a larger number of hosts, the indefiniteness of spanning tree construction with respect to balancedness makes consistent addressing, i.e., specifying the set of hosts the aggregate value should be computed from, a challenging task. To circumvent this limitation, we impose a logical overlay on the set of hosts instead of using a simple spanning tree: The overlay topology is a tree, where each host in the system is a leaf node (see Figure 24.2c). Upper levels of the tree are populated by having selected hosts *simulate* additional nodes. As these nodes have no direct physical counterpart, we call them *virtual nodes* or more shortly *V-Nodes*. Leaf nodes provide initial input to the aggregation tree. Each V-Node $u$ performs aggregation locally by applying an *aggregation function*

$$f(u) := f(\{v \in children(u)\}) \tag{24.1}$$

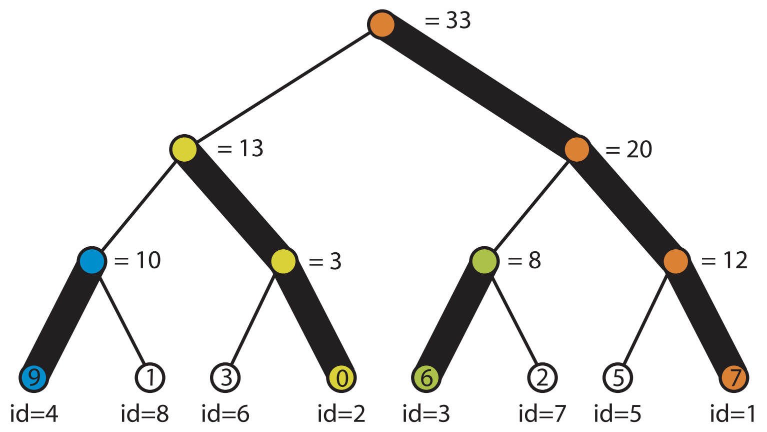
to the values provided by its child nodes. A *level-$\lambda$ aggregate*, where $\lambda \in \{0, \lambda_{max}\}$ is the value resulting from performing the aggregation process up to level $\lambda$. While a level-0 aggregate is the raw input value provided by one of the hosts, a level-$\lambda_{max}$ aggregate summarizes information from all hosts in the system. In Figure 24.2c there are four level-1 aggregates (10,3,8,12), two level-2 aggregates (13,20) and one level-3 aggregate (33). Since aggregates on the same level are characterized by the same level of detail, the tree overlay allows for *consistent addressing* using *aggregation points*: An *aggregation point* is given by a pair $(host, \lambda)$ and is resolved by following $\lambda$ parent links starting from the given host. While



**(a)** Centralized aggregation



**(b)** Spanning tree covering all hosts in the system



**(c)** Aggregation over the proposed logical tree overlay (nodes connected by thick lines represent V-Nodes collocated on the same host)

**Figure 24.2:** Topologies used by different aggregation methods: While the centralized approach shown in Figure (a) does not scale and plain spanning tree-based methods as illustrated in Figure (b) do not allow for consistent addressing, adopting a logical tree overlay based on V-Nodes as depicted in Figure (c) ensures that both requirements are satisfied.

(4,2) and (2,2) reference the same aggregation point, (7,2) references another aggregation point that exposes the same level of summarization, i.e., both aggregation points summarize information from four hosts.

An appropriate selection of hosts to simulate V-Nodes is critical for the overall performance of the aggregation system for two reasons: First, processing incoming values from lower levels, evaluating the aggregation function, and communicating aggregated values to higher levels consumes resources. Placing more than a single V-Node on a host may exceed local resource usage constraints. Second, the cost of reconfiguration caused by a vanishing host increases with the number of hosted V-Nodes. Hence, it is preferable to have the most stable hosts simulate V-Nodes on the highest levels. Therefore, our approach is to prefer hosts that are more capable in terms of both performance and stability to simulate V-Nodes on higher levels. We call this feature of a tree management method *capability-awareness*.

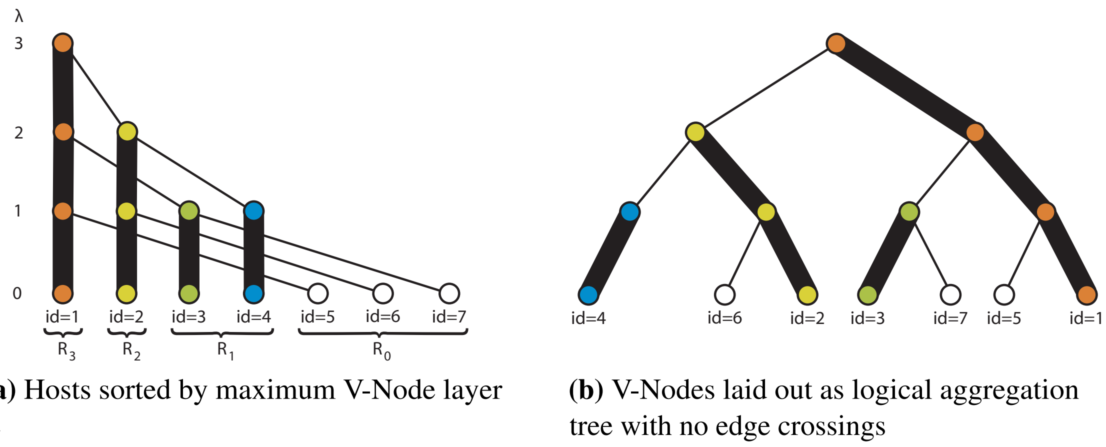
In what follows, we discuss two different tree management methods with different requirements and features (see Table 24.1) and describe how capability-awareness has been incorporated in each case.

## 24.2.1 ID-Based Method

The idea of the ID-based tree management method is to make use of information on group membership that is already available locally to construct and maintain a binary aggregation tree. The levels of the aggregation tree are populated by selecting hosts according to a

| **Method** | **Requirements** | **Features** | | |
|---|---|---|---|---|
| | View Coverage | Scalability | Efficiency | Capability Support |
| **ID** | Complete | Low | **High** | Static |
| **Competition** | Partial | **High** | Low | Dynamic |

**Table 24.1:** Feature/Requirement comparison for the topology maintenance methods



**(a)** Hosts sorted by maximum V-Node layer λ

**(b)** V-Nodes laid out as logical aggregation tree with no edge crossings

**Figure 24.3:** V-Node allocation and link associations for an aggregation group with seven hosts in the ID-based topology management scheme

total order $R$ imposed on the set of hosts within the aggregation group. The structure of tree is defined as follows (cf. Figure 24.3): Let $pos_R(u)$ be the position of host $u$ in the aggregation group $G$ with respect to $R$. Then host $u$ is simulating V-Nodes up to level $\lambda$, if

$$pos_R(u) \in R_\lambda := \begin{cases} \left] \left\lceil \frac{|G|}{2^{\lambda+1}} \right\rceil, \left\lceil \frac{|G|}{2^\lambda} \right\rceil \right] \subset \mathbb{N} & for \quad \lambda \in [0, \lceil log_2(|G|) \rceil [ \\ \{1\} & for \quad \lambda = \lceil log_2(|G|) \rceil \end{cases} \tag{24.2}$$

Additionally, a level-$\lambda$ V-Node $\tilde{u}_\lambda$ located at host $u$ is connected to the collocated V-Node $\tilde{u}_{\lambda-1}$ and to the V-Node $\tilde{v}_{\lambda-1}$ located at host $v$ with

$$pos_R(v) = pos_R(u) + \left\lceil \frac{|G|}{2^\lambda} \right\rceil, \tag{24.3}$$

if it exists.

The algorithm executed by each host to create and maintain this topology works as follows: After becoming a member of the aggregation group, host $u$ is provided with a dynamic view of the hosts within the group. As full knowledge about the group members is necessary to compute $pos_R(u)$, we must employ a group model with complete views. Whenever this view is updated due to the arrival or departure of a host, $pos_R(u)$ is recalculated. Comparison with the former value $pos_{R^*}(u)$ produced on the last update results in creation $(pos_R(u) > pos_{R^*}(u))$ or destruction $(pos_R(u) < pos_{R^*}(u))$ of local V-Nodes on the respective layers. Subsequently, links are created according to the rule given above. Note, that this scheme is highly efficient as no communication other than that for group membership management is required. However, scalability is limited by the necessity for a group model with complete membership lists. Hence, aggregation using ID-based tree management is particularly suited for applications or services, where efficiency is of prime interest.

In the case of capability agnostic topology management, the order $R$ is defined by the lexical order of the unique host identifiers that are provided by the group membership subsystem. Making this management scheme capability-aware is as simple as replacing the lexicographic order with one that reflects the capability of hosts. Therefore, we define $R$ as the ascending value order of the *joint capability* values $c_j(u)$ of the hosts $u \in G$. We use the simple *joint capability function*

$$c_j(u) = p(u)a(u) \tag{24.4}$$

where $p(u)$ is the relative performance and $a(u)$ is the availability ratio of host $u$. The relative performance is computed using the CPU2006 benchmark result database published by the *Standard Performance Evaluation Corporation* (SPEC) [Sta08] as the quotient of the benchmark result of the host $spec\_cpu(u)$ compared to the benchmark result of a reference machine $spec\_cpu_{ref}$ that may be selected arbitrarily but has to be used consistently by all hosts in the aggregation group

$$p(u) = \frac{spec\_cpu(u)}{spec\_cpu_{ref}}. \tag{24.5}$$

The availability ratio $a(u)$ is estimated using the join timestamps of the hosts $T_{join} = \{t_{join}(u) \mid u \in G\}$, where $t_{join}(u)$ is the join timestamp of host $u$, as

$$a(u) \propto \frac{min\ T_{join}}{t_{join}(u)}. \tag{24.6}$$

The joint capability values of all peers must be made available to all peers. This is accomplished either by encoding them statically into peer identifiers or by dynamically exchanging them over a simple Query/Response protocol. For an in-depth discussion see [SBH09].

## 24.2.2 Competition-Based Method

The functional principle of the competition-based tree management method is to have peers compete for becoming higher-level V-Nodes. Initially, all peers host a level-0 V-Node only. If the parent link of a V-Node is vacant for a certain period of time, the hosting peer *promotes* to remedy the shortage by creating a V-Node on the next higher level. Link partners are exchanged between the peers of the aggregation group by dissemination of *link descriptors*. When a matching link partner is found the local and the remote peer negotiate and establish the corresponding link. An established link is replaced by a link to a V-Node hosted by a peer with a lexicographically smaller identifier. If the child links of a V-Node remain vacant or a parented V-Node has only a single child for a certain period of time the V-Node is destroyed. Together these mechanisms result in V-Nodes on higher levels compete for V-Nodes on lower levels in a market-oriented manner. We thus refer to this method as *competition-based*. In contrast to the ID-based method the only requirement on the group model is that at least a single other member is announced to each member of the aggregation group. This is necessary to decide whether a V-Node on level $(\lambda + 1)$ is required, i.e., iff there is at least another member in the level-$\lambda$ layer group $G_\lambda$.

Capability-awareness is incorporated into the competition-based management scheme by using the joint capability function defined above (see Equation 24.4) and by introducing two additional behavioral patterns: First, V-Nodes no longer look for link partners with lexicographically smaller identifiers when their parent link is assigned, but for those that are more capable. Consequently, V-Nodes with higher capability values are more attractive to V-Nodes on the next lower level. To support this behavior, we attach the joint capability value of the hosting peer to the link descriptors disseminated for link establishment. A receiving peer can then quickly check whether another candidate is more capable than the one it is currently connected to. Second, a peer promotes when the joint capability value of the local V-Node is greater than that of a newly assigned parent link partner. This modification is necessary to set off suboptimal steady states. Details can be found in

[SBH09].

## 24.3 Related Work

Existing approaches to information aggregation in loosely coupled distributed systems are either based on flat, gossip-style communication models with special termination/convergence properties or employ a tree overlay topology to hierarchically compute aggregate values.

In [MJB04] an aggregation method of the former type based on an epidemic protocol is discussed. Every peer periodically selects a neighbor peer at random, exchanges values denoting the system state, and performs an aggregation specific computation. Based on a basic protocol to compute averages, several other aggregation functions, like `sum` and `variance`, are realized. While epidemic protocols are known to be exceptionally robust, their efficiency is limited.

Tree topology based aggregation methods can be classified in static and dynamic approaches, depending on the way the tree topology is defined and maintained. Dynamic approaches either depend on an unstructured communication model or leverage structured network overlay technology. Subsequently, we discuss representative examples of these classes.

*Ganglia* [MCC04] is a scalable distributed monitoring system mainly targeted at federations of clusters. It employs a listen/announce protocol based on multicast for monitoring individual clusters. Thus, all nodes of a cluster collect and store the state of all other nodes. Membership is maintained by a multicast heartbeat protocol. Within federated clusters, *Ganglia* uses a tree-based protocol for information aggregation, where leaves of the tree are representative nodes of each cluster. For handling failures, multiple nodes of a single cluster can be specified as representatives. Aggregation at inner nodes of the tree is accomplished by periodically polling child nodes. Configuration files are used to specify the structure of the aggregation tree, which typically reflects the administrative topology. As *Ganglia* takes a static approach, it is not appropriate for highly dynamic P2P Desktop Grids.

*Astrolabe* [RBV03] is a robust and scalable monitoring and management solution for distributed systems. Information aggregation is based on a tree structure which reflects the administrative organization of the distributed system. The topology is maintained using an unstructured gossip protocol that is also used for information dissemination. All aggregate values of a subtree are replicated on every node of the subtree, such that all respective queries can be answered with local information. *Astrolabe* uses a restricted form of mobile code based on SQL syntax for specifying aggregation functions.

The aggregation method of *SDIMS* [YD04] leverages the internal routing protocols of *Distributed Hash Tables* (DHT) to establish a tree based aggregation hierarchy. With this approach, the union of the search/routing paths for a key from all nodes forms a tree. As keys are derived from attribute names, different attribute names are mapped to different trees, such that each node acts as an intermediate point of aggregation for some attributes. Thus, the onus of aggregation can be distributed among the participating hosts. In order to achieve administrative isolation, so called *Autonomous Distributed Hash Tables* (ADHT) are employed which ensure that search paths are always contained in the smallest possible domain and that search paths for a key from different nodes of a domain converge at a node part of that domain. On top of the ADHT layer, the *Aggregation Management Layer* (AML) is responsible for maintaining attribute tuples, performing aggregations, and storing aggregated values. For increasing robustness in case of network reconfigurations, the AML layer performs replication both in time (lazy and on-demand) and space.

Neither *Astrolabe* nor *SDIMS* take varying host capabilities and stability into account. Thus, the efficiency of the aggregation process is limited by the speed of slow hosts and the accuracy may be seriously impaired by unstable hosts.

Sensor networks share some of the characteristics of P2P Desktop Grids, like constrained resources, limited view of the whole system, and a high degree of volatility. *TAG* [MFHH02] is an aggregation system especially addressing these sensor network specific issues. The system represents an in-network aggregation approach and is based on routing trees. Trees are constructed in an ad-hoc fashion leveraging the range restricted broadcast capabilities of individual sensors.

# Part VII

# Distributed Task Pool

Due to their inherent irregular nature a task parallel approach with support for dynamic problem decomposition is imperative to solve ISPs efficiently. However, distributed task parallel programs require platform support in terms of a task pool abstraction that can be used by peers to fetch and store tasks. COHESION's distributed task pool supersedes the centralized task pool approaches of existing Desktop Grid platforms that do not support the Peer-to-Peer interaction model. The task pool employs randomized load balancing, which is known to be optimal with high probability in this context, a task-tracking checkpoint/restart fault-tolerance scheme, and a novel termination detection algorithm that is resilient to task duplication.

## Related Publications

[SB10b] SCHULZ, Sven and BLOCHINGER, Wolfgang: **Parallel SAT-Solving on Peer-to-Peer Desktop Grids**. *Journal of Grid Computing* (2010), Bd. 8(3):S. 443–471

There are two fundamentally different kinds of parallelism: *data* and *task parallelism*. Problem decomposition in data parallel programs – classified as SIMD (*Single Instruction Multiple Data*) or SPMD (*Single Program Multiple Data*) systems according to Flynn's taxonomy [Fly72] – is done by partitioning the usually large and structured input data (like one or more arrays). Partitions are mapped to processors and are processed independently. By contrast, in task parallel programs – MPMD (*Multiple Programs Multiple Data*) or MISD (*Multiple Instructions Multiple Data*) systems according to Flynn – each processor performs a different operation usually on the whole input.

Suppose we apply a data parallel approach to an ISP. Due to its inherent irregular nature, the processing time for subtasks generated by data partitioning would vary considerably. Assigning a single subproblem to each processor consequently would result in significant processor idling and limited parallel efficiency. While reducing the granularity, i.e., creating a larger number of subproblems and assigning multiple of them to each processor, can alleviate the differences in workload, the resulting overhead, however, also impairs efficiency. Hence, data parallel approaches are in general not suitable for solving ISPs. As will be described in Part VIII, this is particularly true for distributed SAT solving.

Due to the unpredictability of task run times, problem decomposition for ISPs has to go hand in hand with execution. Such an on-line decomposition approach is called *dynamic decomposition* [GGKK03]: Starting from a single task consisting of the whole input, new tasks are derived iteratively by decomposition whenever a processor becomes idle. In contrast to data parallel approaches, task parallelism with dynamic decomposition requires sophisticated support by the parallel execution environment. This is most often achieved using an abstraction called a *task pool*. Module 24.1 defines a rudimentary interface for a Task Pool module. The functionality includes submission of a *job* using the ⟨ Submit ⟩ request and a mechanism to notify clients of the completion of a job by means of a ⟨ Completed ⟩ indication. Note that the task pool abstraction implemented in Cohesion provides a significantly richer interface that includes operations to cancel queued submissions, to abort jobs that are currently executed, and to retrieve a wide range of metrics that can be used for monitoring purposes. However, for reasons of clarity, we exclude these functionality from our discussion. The same is true for application level error handling, an aspect that is definitely important for an implementation to be usable, but not required to understand the core protocols discussed within this thesis.

Module 24.1 shows the external interface of the Task Pool module only. Internally, a task pool provides operations to store and manage the tasks created throughout the execution of a job. These operations are provided by submodules and are described in subsequent chapters. All processes constituting the distributed task pool have access to these internal operations so that they can fetch tasks for execution on demand and enqueue tasks dynamically created during execution.

The Task Pool module definition includes a property called *Eventual Completion* (TP1). It guarantees that a job that is submitted to the task pool on a peer $p_c$, which is referred to as the *coordinator* (for that job) in the following, will be completed eventually, if $p_c$ is

**Module 24.1** Interface and Properties of the Task Pool module

**Module:**

**Name:** Task Pool, **instance** *tp*.

**Requests:**

$\langle$ *tp*, Submit | *job* $\rangle$: Submits a *job* for execution.

**Indications:**

$\langle$ *tp*, Completed | *job* $\rangle$: A *job* has been completed.

**Properties:**

**TP1:** *Eventual Completion:* If the peer where a job $j$ is submitted is *correct*, $j$ will be completed eventually.

correct[1]. We will show that COHESION's task pool implementation enforces this property in Chapter 29 based on the properties of the submodules discussed in the following chapters. If not stated otherwise, we assume that the coordinator peer is correct in the following.

The task pool abstraction can be realized in a centralized or in a distributed manner. In centralized approaches – provided by most existing Desktop Grid Computing platforms (cf. Chapter 10) – a master process maintains a global task queue from which idle worker processes can fetch tasks. The queue is refilled with tasks generated through master- or worker-side problem decomposition performed or triggered by the master whenever the global task queue is about to become empty. A major drawback of centralized task pools is that the master process is a sequential bottleneck as every task must be transferred from and to the master process. As will be discussed in Part VIII, this limitation is in particular impedimental in the case of distributed SAT solving as a task description may become very large for real-world SAT instances with millions of variables. Enabled by ORBWEB's support for Peer-to-Peer message exchange, COHESION provides a *distributed* task pool model, in which a task queue is located on every peer. While distribution eliminates the master process as a sequential bottleneck, the downside of this approach is that realizing essential aspects of the task pool control logic becomes much more challenging. These aspects are load balancing, fault tolerance, and termination detection.

The remainder of this part is organized as follows: First, some basic abstractions are introduced in Chapter 25. In Chapters 26-28 the submodules for load balancing, fault tolerance, and termination detection are specified and their implementation within COHESION is discussed. Chapter 29 gives a correctness argument for the task pool implementation as a whole. Finally, the implementation is subjected to a thorough performance analysis in Chapter 30 with real-world resource volatility.

## Terminology and Notation

The modular notation and the pseudo code used to specify the APIs and algorithms of this chapter are introduced in Appendix B.

1 Furthermore, as COHESION implements the task pool abstraction on top of a peer group, it is assumed that the coordinator peer stays in this group (at least) until the job is completed.

# 25 Basic Abstractions

COHESION's implementations of load balancing, fault tolerance, and termination detection are based on four basic abstractions. These abstractions are:

1. A *Task*, which is an *abstract data type* (ADT) that encapsulates a unit of work together with metainformation required by the submodules of COHESION's task pool.
2. A *Task Queue* module, which is essentially a Queue ADT that is able to indicate modifications to interested parties.
3. A *Processing Element* (PE) module that specifies a processor for tasks with the additional ability to split off a task from and to create a checkpoint of the task that is currently executed by the PE on demand.
4. A *Timer* abstraction used to schedule a callback after a certain period of time.

Each of these abstractions is described subsequently up to the level of detail that is required to understand the task pool implementation. When required, we also give some information on how the abstractions are implemented within COHESION.

## 25.1 Task

A *task* is a unit of work that carries both what should be processed and how. A *job*, as used in the definition of Module 24.1, is essentially defined by a single task to be executed in order to complete the job. This task is called the *root task* or *initial task* in the following.

At the heart of a task is its *input data* (or just *input* for short). We do not try to specify the data format for the input data as this is application dependent. However, we do specify some attributes of and operations on the Task ADT that are required by the submodules of the COHESION task pool. These attributes and operations are denoted as functions over instances of the Task ADT. In case of attributes, $\mathrm{A}(t) := v$ denotes an assignment of the value $v$ to the attribute $\mathrm{A}$ of task $t$:

The attribute $\textsc{Input}(t)$ gives the input of task $t$. When used in formal proofs, we abbreviate this to $\mathrm{I}(t)$ in the following. An input $i$ is said to *subsume* another input $j$, if and only if regarding to the result of the computation the execution of a task $t_j$ with input $j$ is optional when a task $t_i$ with input $i$ is completed. If the inputs $i$ and $j$ are unambiguously determined by the context, we also say that task $t_i$ subsumes task $t_j$. As an example for subsumption, consider the problem of searching an element $e$ in arrays: If $a$ is an array that is a subarray of array $b$, then the input $(b,e)$ to the search algorithm subsumes the input $(a,e)$. This is obvious, since if $e$ is found in $a$, it is also found in $b$. Subsumption can be applied to sets of inputs as well: Let $I$ and $J$ be such sets of inputs. Then $I$ is said to subsume $J$, if and only if the execution of *all* tasks for the inputs in $J$ are optional, when *all* tasks for the inputs in $I$ are completed.

The $\textsc{Split}(t)$ operation splits off and returns a new task from the *donator* task $t$. Splitting is done by means of the *split operator* $\prec : \mathcal{I} \mapsto \mathcal{I}^2, (i) = (j,k)$ with $\mathcal{I}$ being the space of possible inputs. While $i$ is called the *target* of the split operation, $j$ and $k$ are called its *split products*. In relation to $j$ and $k$, $i$ is referred to as the *parent* of the split products. Essential requirements on the split operator are that neither $j$ nor $k$ are *empty*, $i$ is subsumed by $(j,k)$ and vice versa, and that $j$ and $k$ are *disjunctive* in the sense that no input that is part of any possible decomposition of $j$ is subsumed by $k$ and vice versa.

If we apply the split operator recursively on inputs from a set of inputs that initially contains only a single input $r$ and we connect the target of each split operation by an edge with its split products, a tree is formed. This tree is called the *decomposition tree* of $r$. A path within this tree from the root $r$ to an input $i$ is called the *decomposition path* of $i$.

The $\mathrm{ID}(t)$ attribute returns a globally unique identifier for task $t$. Our implementation uses a pair $(j,s)$ of *Peer Unique Identifiers* (PUID) where $j$ is the *job identifier* and $s$ is the so called *split suffix*. PUIDs are a concatenation of the unique identifier of the peer where the PUID is generated and a peer local monotonic counter. The counter is incremented every time a PUID is generated by invoking $PUID()$. When a job is submitted, its identifier is set to $(PUID(), \perp)$. The task identifier for a split off task is derived from the original task's identifier by replacing the split suffix with the $PUID()$ evaluated on the peer where the split happens. When we talk of a task in the following, we refer to a task with a specific identifier and *not* to a task with a specific input. While the former is fixed, the latter may change during the lifetime of a task.

The $\mathrm{UC}(t)$ attribute is a monotonic counter that is used to detect and discard delayed messages in $\textsc{Cohesion}$'s fault-tolerance protocol. $\mathrm{UC}(t){+}{+}$ increments the counter by one.

$\textsc{Compensates}(t)$ is an attribute required to coordinate the interaction between the fault-tolerance and the termination detection protocol. It is assigned either to some Task ADT instance or to $\perp$.

Finally, the attribute $\textsc{IsRoot}(t)$ indicates whether task $t$ is the *root task* of a *job* and the attribute $\textsc{Coordinator}(t)$ gives the peer identifier of the coordinating peer for task $t$. Due to the way we construct them, both attributes can be easily computed from the identifier of a task: $\textsc{IsRoot}$ by checking whether there is a split suffix present and $\textsc{Coordinator}$ by extracting the peer identifier from the PUID used as the job identifier.

In addition, we make the following assumptions regarding the properties and behavior of tasks:

1. Task execution is *idempotent*, which means that it doesn't matter how often a task is completed as long as it is completed once. This is required as tasks may be duplicated by the fault-tolerance mechanism of $\textsc{Cohesion}$.

2. Every input can be processed in finite time. Otherwise, termination of the computation can not be guaranteed.

## 25.2 Task Queue

**Module 25.1** Interface of the Task Queue module

**Module:**

**Name:** Task Queue, **instance** *tq*.

**Requests:**

⟨ *tq*, Enqueue | *task* ⟩: Enqueues a *task*.

⟨ *tq*, Dequeue | *task* ⟩: Dequeues a *task*.

⟨ *tq*, Lookup | *taskId*, *task* ⟩: Gets the *task* with *taskId*, if contained.

⟨ *tq*, Size | *size* ⟩: Gets the number of contained elements.

**Indications:**

⟨ *tq*, Modified | *size* ⟩: The queue has been modified due to an ⟨ Enqueue ⟩ or ⟨ Dequeue ⟩ operation. *size* is the number of contained elements after the modification.

As described in the introduction to this part, COHESION's decentralized task pool maintains a local task queue on every peer. Module 25.1 shows the interface definition for such a task queue. ⟨ Enqueue ⟩ and ⟨ Dequeue ⟩ are mutating operations from the well-known Queue ADT to enqueue and dequeue a task, respectively. The latter returns $\perp$ in case the queue is empty. ⟨ Lookup ⟩ returns the task with the given task identifier or $\perp$ in case no such task is contained in the queue. ⟨ Size ⟩ gives the number of elements contained in the task queue. Finally, ⟨ Modified ⟩ indicates a modification to the data structure triggered by one of the mutating requests. The indication carries the number of elements contained in the queue after the modification by means of the *size* parameter.

COHESION by default uses a standard *First-In-First-Out* (FIFO) queue as the underlying ADT to realize the Task Queue module. However, this is designed to be configurable on a per task pool instance basis in order to allow application developers to tune the task pool to applications where using a non-FIFO queue, like for example a priority queue, yields better results.

## 25.3 Processing Element

Module 25.2 shows the interface of the Processing Element (PE) module. A PE is the abstraction of a compute engine that fetches tasks from the local task queue and processes them. Whenever a task is completed by the PE, a ⟨ Completed ⟩ indication is triggered. A PE also allows to trigger a split operation by means of the ⟨ Split ⟩ request. Internally, this interrupts processing of the currently executing task and invokes the SPLIT operation of the Task ADT that splits off a new task in an application-specific manner. The split off task is stored in the *task* output parameter to be used by the invoking protocol. Finally, a ⟨ Split ⟩ indication is triggered that carries both the original task with updated input and the split off task by means of the *donator* parameter and the *task* parameter, respectively. The ⟨ Checkpoint ⟩ operation is required for implementing Periodic Checkpointing, an optimization to COHESION's fault-tolerance protocol described in Section 27.4. Finally, the PE interface allows for getting the task that is currently processed by means of the ⟨ Executing ⟩ request. $\perp$ is returned in case the PE is idle. The same is true for the ⟨ Split ⟩ request.

**Module 25.2** Interface of the Processing Element module

**Module:**

**Name:** Processing Element, **instance** *pe*.

**Requests:**

⟨ *pe*, Split | *task* ⟩: Splits off a new *task* from the currently running task.

⟨ *pe*, Executing | *task* ⟩: Returns the *task* that is currently processed.

⟨ *pe*, Checkpoint | *input* ⟩: Returns an *input* that subsumes the original input in the context of the execution. The input of the currently processed task is replaced by *input*.

**Indications:**

⟨ *pe*, Completed | *task* ⟩: A *task* has been completed.

⟨ *pe*, Split | *donator*, *task* ⟩: A new *task* has been split off from a *donator* task .

Today's computers can work on more than a single thread (of execution) in parallel. To exploit this capability, COHESION allows for instantiating any number of PEs on a peer which all fetch tasks from the same shared task queue. However, for certain applications this strategy of instantiating one PE per hardware thread is disadvantageous because other resources than CPU cycles (like I/O capacity or memory bandwidth) are not available to a sufficient extent. Such non-scaling applications will not benefit from adding more PEs and may even exhibit slowdowns. SAT solving (see Part VIII) is a good example for this behavior as a single solver core may already saturate the cache capacity and the memory bandwidth available on COTS processors. Hence, COHESION allows to configure the number of PEs on a per task pool basis. By default, a PE is instantiated for each available hardware thread. However, for reasons of clarity, we restrict our discussion to a task pool with a single PE per peer. An extension of the protocols to the general case is straightforward.

## 25.4 Timers

**Module 25.3** Interface of the Timer module

**Module:**

**Name:** Timer, **instance** *t*.

**Requests:**

⟨ *t*, Schedule | *delay*, *token* ⟩: Schedules an ⟨ Elapsed ⟩ indication carrying a *token* to be triggered after the *delay* has elapsed.

⟨ *t*, Cancel | *token* ⟩: Cancels the scheduled indication for the given *token*.

⟨ *t*, Scheduled | *token*, *scheduled* ⟩: Checks whether an ⟨ Elapsed ⟩ indication is *scheduled* for a *token*.

**Indications:**

⟨ *t*, Elapsed | *token* ⟩: The delay for a scheduled indication with *token* has elapsed.

Module 25.3 shows the interface of the Timer module. A *Timer* allows for scheduling future indications using the ⟨ Schedule ⟩ request. An ⟨ Elapsed ⟩ indication is triggered after the specified *delay* has elapsed. It carries an arbitrary *token* for demultiplexing. The ⟨ Cancel ⟩ request can be used to cancel a previously scheduled indication. It does nothing when no indication with the given *token* has been scheduled or the indication has already been triggered. The ⟨ Scheduled ⟩ predicate can be used to check whether an ⟨ Elapsed ⟩ indication is scheduled for a specific *token* that has not been triggered yet nor has been canceled.

# 26 Load Balancing

*Task scheduling* is the assignment of tasks to processors according to a given policy. The underlying theoretical problem is the NP-complete *generalized scheduling problem* [VD76]. Our view on the topic is highly focused on the special case of ISPs. For a comprehensive treatment of the vast subject the reader is referred to [SKH95].

A main discriminator among task scheduling techniques is whether they are static or dynamic. Static scheduling is done before program execution and typically requires detailed knowledge regarding both the tasks to be executed and the environment they should be executed in. Dynamic techniques require less knowledge and are executed concurrently to the computation. They are based on transferring tasks from heavily loaded processors to lightly loaded processors and thus are often referred to as *load balancing* techniques.

A key characteristic of ISPs is that many properties (like run time, memory usage, etc.) of its tasks cannot be determined in advance. However, the efficiency of schedules computed by static scheduling algorithms is crucially dependent on the estimation accuracy for these properties. Since such estimations are exceedingly difficult and more often even impossible for ISPs, static scheduling techniques are in general not applicable for them. Hence, Cohesion implements load balancing for task scheduling. Module 26.1 shows the interface of the Load Balancing module. It consists of a single indication ⟨ Transferred ⟩ that is emitted when a *task* has been transferred from the local task queue to a *target* peer. Note that the term *transferred* in this context does not mean that the delivery is acknowledged by the remote peer. It only means that is has been dequeued locally and sent to the remote peer.

Blumofe and Leiserson [BL99] have shown that for applications where task sizes are not known a priori a simple randomized load balancing algorithm is optimal with high probability. Well-known representatives of these algorithms are *random stealing* and *random pushing* [GGKK03]. While in random stealing protocols idle nodes pull or *steal* tasks from randomly selected neighbors, random pushing protocols go for the opposite approach, where nodes with excess tasks push them to randomly selected target nodes. We prefer random stealing over random pushing as it is demand-driven and thus avoids unnecessary transferral of potentially large tasks.

Protocol 26.2 shows the framework for Cohesion's random stealing implementation that is used as a basis for the concrete implementations described subsequently. It does not include the logic that triggers task decomposition and does not handle incoming load balancing requests. Hence, it is denoted as *abstract*. The implementation periodically checks whether the PE is idle and no task is available from the local task queue (ARS-1). If this condition holds, a `Steal` message is sent to a randomly selected member from the local view commonly referred to as the *victim* (of the potential theft). This selection is done by means of the Random function that selects a random element from the provided set. In case the argument is the empty set, $\perp$ is returned. To minimize processor idling the same

**Module 26.1** Interface of the Load Balancing module

```
Module:
    Name: Load Balancing, instance lb.
Indications:
    ⟨ lb, Transferred | task, target ⟩: A task transferral from the local task queue to the peer
  with identifier target has been initiated.
```

**Protocol 26.2** Abstract Random Stealing protocol

```
Implements:
    Load Balancing, instance ars with balancing period and group name.

Uses:
    Timer, instance t.
    Task Queue, instance tq.
    Peer Group Management, instance pgm.
    Processing Element, instance pe.

upon event ⟨ ars, Init | period, name ⟩                                        ▷ ARS-0
    trigger ⟨ t, Schedule | 0, name ⟩;

upon event ⟨ pe, Completed | task ⟩ or upon event ⟨ t, Elapsed | token ⟩ such that
token = name do                                                                ▷ ARS-1
    call ⟨ tq, Size | s ⟩;
    call ⟨ pe, Executing | task ⟩;
    if $s = 0 \wedge task = \perp$ then
        call ⟨ pgm, View | members ⟩;
        $victim := \textsc{Random}(members \setminus \text{pgm}.localPeerId)$;
        if $victim \neq \perp$ then
            trigger ⟨ pgm, Unicast | victim, Steal[] ⟩;
    trigger ⟨ t, Schedule | period, name ⟩;

upon event ⟨ pgm, Deliver | source, Transfer[task] ⟩ do                        ▷ ARS-2
    trigger ⟨ tq, Enqueue | task ⟩;
```

handler is also triggered immediately whenever the PE becomes idle. When a `Transfer` message is delivered locally, the contained task is enqueued into the task queue (ARS-2).

COHESION supports both an eager and an on-demand task decomposition strategy. Both protocols extend the Abstract Random Stealing (ARS) protocol. ARS with *Eager Task Decomposition*, as shown in Protocol 26.3, splits off subtasks when the number of tasks in the local queue drops below a configurable *threshold* (ETD-0). When a `Steal` message is delivered, a task is dequeued from the local task queue, wrapped up in a `Transfer` message, and send by unicast to the requesting peer (ETD-1). As all messages send by the protocols described in this chapter, the potentially large `Transfer` message is sent via E2E links when available. Finally, a ⟨ Transferred ⟩ indication is triggered to notify upstream modules about this event. If no task is available locally, nothing happens.

In contrast, ARS with *On-Demand Decomposition* as shown in Protocol 26.4 initiates decomposition only, when the transferral of a task is explicitly requested by a remote peer.

**Protocol 26.3** Random Stealing with Eager Task Decomposition protocol

```
Extends:
  Abstract Random Stealing with Eager Task Decomposition, instance etd with split threshold.

upon event ⟨ tq, Modified | $size$ ⟩ do                                    ▷ ETD-0
  if $size < threshold$ then
    call ⟨ pe, Split | $task$ ⟩;
    if $task \neq \perp$ then
      trigger ⟨ tq, Enqueue | $task$ ⟩;

upon event ⟨ pgm, Deliver | $source$, Steal[] ⟩ do                         ▷ ETD-1
  call ⟨ tq, Dequeue | $task$ ⟩;
  if $task \neq \perp$ then
    trigger ⟨ pgm, Unicast | $source$, Transfer[$task$] ⟩;
    trigger ⟨ lb, Transferred | $task$, $source$ ⟩;
```

**Protocol 26.4** Random Stealing with On Demand Decomposition protocol

```
Extends:
  Abstract Random Stealing with On-Demand Decomposition, instance odd.

upon event ⟨ pgm, Deliver | $source$, Steal[] ⟩ do                         ▷ ODD-0
  call ⟨ tq, Dequeue | $task$ ⟩;
  if $task = \perp$ then
    call ⟨ pe, Split | $task$ ⟩;
  if $task \neq \perp$ then
    trigger ⟨ pgm, Unicast | $source$, Transfer[$task$] ⟩;
    trigger ⟨ lb, Transferred | $task$, $source$ ⟩;
```

This behavior is realized by listening for incoming `Steal` messages (ODD-0). If no task is available in the local task queue when such a message is delivered, a ⟨ Split ⟩ request is triggered on the PE. The task, either split off or taken from the task queue, is wrapped up in a `Transfer` message and sent to the requesting peer. Finally, a ⟨ Transferred ⟩ indication is triggered to notify upstream modules about the transfer.

By default, COHESION instantiates the ARS with On-Demand Decomposition protocol when creating a task pool, because, in general, processing more task induces more excess computation.

# 27 Fault Tolerance

High Performance Computing (HPC) applications may have long run times in the range of days. However, as discussed in Part I, Desktop Grid environments are highly volatile as hosts and network links are non-dedicated and characterized by increased failure probabilities. Hence, the occurrence of failures during a prolonged application run is the rule rather than the exception. In contrast to embarrassingly parallel applications, tasks in a HPC application are not independent. Thus, the remaining part of the computation may depend on the outstanding outcome of a task that got lost due to a failure. Hence, the underlying middleware must restart failed tasks as soon as possible. To achieve this, COHESION employs a checkpoint/restart fault-tolerance scheme that tracks tasks over their entire lifespan until they have been fully processed. In the following, we describe and discuss this protocol and some optimizations. In accordance with our system model defined in Chapter 16, we subsequently assume a dynamic set $P = \{p_1, \ldots, p_n\}$ of crash-stop peers. However, as we use a mix of E2E links and the FIFO-Order Perfect Links from Chapter 16, the distributed system model exposed to the application is no longer fail-stop. The peer $p_c \in P$ on which a job is submitted takes the role of the *coordinator*. $p_c$ is assumed to be correct until the computation has been completed.

## 27.1 Task-Tracking Checkpoint/Restart Module

Module 27.1 shows the interface of COHESION's Task-Tracking Checkpoint/Restart module. The module is purely reactive and triggers internal operations by means of listening to indications from other modules of the task pool and the network substrate. Thus, it has

**Module 27.1** Interface and properties of the Task-Tracking Checkpoint/Restart module

**Module:**

**Name:** Task-Tracking Checkpoint/Restart, **instance** *ttcr*.

**Requests:**

⟨ *ttcr*, Tracked | *task*, *tracked* ⟩: Checks whether a *task* is *tracked* locally.

**Properties:**

**TTCR1:** *Eventual Completion:* Every tracked task or a restored substitute is eventually completed.

**TTCR2:** *Safe Split:* If a tracked task $t$ is split updating $\mathrm{I}(t)$ to $I'$ and creating a new task $t^*$ by means of the split operation $\prec(\mathrm{I}(t)) = (I', I^*)$, then either $t$ remains tracked with unmodified input, or $t$ and $t^*$ are tracked with $\mathrm{I}(t) = I'$ and $\mathrm{I}(t^*) = I^*$ and $\mathrm{I}(t)$ is subsumed by $I'$ and $I^*$.

**TTCR3:** *Eventual Subsumption*: If a task $t$ is tracked with input $i = \mathrm{I}(t)$, eventually a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{\mathrm{I}(t_1), \ldots, \mathrm{I}(t_n)\}$ is completed such that the inputs $I$ collectively subsume $i$.

no public API in the form of requests and indications, except the ⟨ Tracked ⟩ request that allows for checking whether a given task is tracked locally. It is introduced for formal reasons only as it is required to formulate the preconditions of the module's guaranteed properties TTCR1-TTCR3 by means of the following definition:

**Definition 27.1.1.** *A task $t$ for which ⟨ Tracked | $t$ ⟩ yields true on the coordinator peer $p_c$ is called* tracked*.*

TTCR1 says that every tracked task – uniquely identified by its task identifier – or, in case it gets lost due to a failure, a surrogate task restored by the protocol is completed eventually. It says nothing about the inputs of these tasks. This is different for TTCR2, which guarantees that if a tracked task is split, either the original task with the original input is tracked or both split products are tracked and their inputs together subsume the original task's input. Stated even simpler, TTCR2 says that a split is safe in the sense that the same overall job is performed even though any number of tasks are split during the computation. Finally, TTCR3 extends the guarantee of TTCR2 from split operations to every possible execution and additionally assures that the computation will eventually terminate[1].

## 27.2 Basic Protocol

Protocol 27.3 implements the Task-Tracking Checkpoint/Restart module. Its purpose is to keep record of the locations and inputs of all uncompleted tasks in a data structure on the coordinator, subsequently called the *task database*. The data stored therein is used to restart tasks in case of failure. The protocol is an extension to Protocol 27.2 that tracks the lifecycle of local tasks and notifies the coordinator in case of relevant changes. Every peer runs an instance of the combined protocol and takes over the role of the coordinator for locally submitted jobs.

A change to the location or input of a task is communicated to the coordinator through `Update`$(t,\ p_i,\ p_f)$ messages, where $t$ is a representation of the affected task including all the attributes described in Section 25.1, and $p_i$ and $p_f$ are the initial and final location of the task, respectively. While `Update`$(t,\ \perp,\ p_f)$ means that task $t$ has been created on peer $p_f$, either as the initial task of a new job or as one of the split products from a split operation, `Update`$(t,\ p_i,\ \perp)$ means that task $t$ has been completed on $p_i$ and should thus be removed from the task database. An `Update` message can also contain more than one update. This is denoted as `Update`$(u_1,\ldots,u_n)$ with tuples $u_j = \left(t^j, p_i^j, p_f^j\right)$ for $0 < j \leq n$.

All `Update` messages are sent over Cohesion's FIFO-Order Perfect Links (see Chapter 16). Hence, `Update` messages originating from the same peer are delivered in FIFO order on the coordinator and thus in the order they are generated. However, this is not true for `Update` messages (for the same task) generated at different peers. For example, if a task is migrated from one peer to another and subsequently split, the `Update` message for the latter event may arrive before that for the former. This is because the `Transfer` messages used for task transferral are send over E2E links. Without further measures, the sequence

1 This only holds due to our assumption that the coordinator is stable.

**Protocol 27.2** Abstract Task-Tracking Checkpoint/Restart protocol

```
Implements:
    Task-Tracking Checkpoint/Restart, instance ttcr.

Uses:
    Load Balancing, instance lb.
    Peer Group Management, instance pgm.
    Processing Element, instance pe.
    Task Queue, instance tq.

upon event ⟨ tq, Enqueue | task ⟩ such that $\textsc{isRoot}(task)$ do                ▷ TTCR-0
    $\textsc{coordinator}(task)$ :=pgm.$localPeerId$;
    trigger ⟨ pgm, Unicast | pgm.$localPeerId$, Update[$task$, $\perp$, pgm.$localPeerId$] ⟩;

upon event ⟨ pe, Completed | task ⟩ do                                              ▷ TTCR-1
    $p_c := \textsc{coordinator}(task)$;
    $\mathrm{UC}(task)$++;
    trigger ⟨ pgm, Unicast | $p_c$, Update[$task$, pgm.$localPeerId$, $\perp$] ⟩;

upon event ⟨ tq, Split | donator, task ⟩ do                                          ▷ TTCR-2
    $p_c := \textsc{coordinator}(task)$;
    $\mathrm{UC}(donator)$++;
    trigger ⟨ pgm, Unicast | $p_c$, Update[{ ($task$, $\perp$, pgm.$localPeerId$), ($donator$,
pgm.$localPeerId$, pgm.$localPeerId$) }] ⟩;

upon event ⟨ lb, Transferred | task, target ⟩ do                                     ▷ TTCR-3
    $p_c := \textsc{coordinator}(task)$;
    $\mathrm{UC}(task)$++;
    trigger ⟨ pgm, Unicast | $p_c$, Update[$task$, pgm.$localPeerId$, $target$] ⟩;
```

of events related to a task can not be reconstructed reliably on the coordinator. To cope with this issue, all event handlers of Protocol 27.2 increment the update counter $\mathrm{UC}$ for a task when either its location or its input is updated. As described below, this is used by the coordinator to detect and discard delayed `Update` messages that would render the reconstruction of the actual update sequences impossible.

TTCR-0 is executed for the root task of a job when it is submitted to the task pool using the ⟨ Submit ⟩ request of the Task Pool module. As the local peer becomes the coordinator for all locally submitted jobs, the $\textsc{Coordinator}$ attribute of the task is set to the local peer identifier. Finally, the peer sends an `Update` message to itself indicating that the root task has been created locally.

TTCR-1 is the counterpart to TTCR-0: It is invoked when a task has been completed and thus should no longer be tracked. As in the case of TTCR-0 an `Update` message is sent to the coordinator that triggers the corresponding change to the task database. As we are not necessarily on the coordinator this time, its peer identifier is read from the task using the $\textsc{Coordinator}$ attribute of the Task ADT.

When a split operation is performed, TTCR-2 is executed. It consists of two update operations: one for each split product. If the update for the donator task was correctly reported but that for the split off task was not, a crash of the peer where the split took place

**Protocol 27.3** COHESION's Task-Tracking Checkpoint/Restart Coordination protocol

```
Extends:
   Abstract Task-Tracking Checkpoint/Restart with Coordination, instance ttcr-c with parameter task restoration timeout.

Uses:
   Timer, instance t.

upon event ⟨ ttcr-c, Init | $timeout$ ⟩ do                                        ▷ TTCR-C-0
   $\mathcal{L} := \varnothing$;
   $\mathcal{T} := \varnothing$;

upon event ⟨ ttcr, Tracked | $task$, $tracked$ ⟩ do                               ▷ TTCR-C-1
   $tracked := \text{COORDINATOR}(task) = \text{pgm}.\mathit{localPeerId} \wedge \mathcal{L}(\text{ID}(task)) \neq \perp$;

upon event ⟨ pgm, Deliver | $source$, Update[$task$, $p_i$, $p_f$] ⟩ such that $\mathcal{T}(Id := \text{ID}(task)) = \perp \vee \text{UC}(\mathcal{T}(Id)) < \text{UC}(task)$ do   ▷ TTCR-C-2
   if $p_i \neq \perp$ then
      trigger ⟨ t, Cancel | $id$ ⟩;
      $\mathcal{L}(id) := \perp$;
   if $p_f \neq \perp$ then
      $\mathcal{L}(id) := p_f$;
      $\mathcal{T}(id) := task$;
      call ⟨ pgm, View | $v$ ⟩;
      if $p_f \notin v$ then
         trigger ⟨ t, Schedule | $timeout$, $id$ ⟩;

upon event ⟨ pgm, ViewUpdate | $p$, $state$ ⟩ such that $state = \text{LEFT}$ or upon event ⟨ pgm, Deliver | $p$, Steal[] ⟩ do   ▷ TTCR-C-3
   for each $(taskId, l) \in \mathcal{L} : l = p$ do
      trigger ⟨ t, Schedule | $timeout$, $taskId$ ⟩;

upon event ⟨ t, Elapsed | $taskId$ ⟩ do                                            ▷ TTCR-C-4
   $task := \mathcal{T}(taskId)$;
   $\text{COMPENSATES}(task) := task$;
   $\text{ID}(task) := (PUID(), \perp)$;
   trigger ⟨ tq, Enqueue | $task$ ⟩;
```

would be unrecoverable[1]. This is because the split off task would have been never stored in the coordinator's task database. Consequently, property TTCR2 of the Task-Tracking Checkpoint/Restart module would be violated. To avoid this situation, either both or none of the updates must be delivered. As ORBWEB guarantees that message delivery is atomic, this requirement can be satisfied easily by using a single `Update` message for transmitting both updates.

Finally, TTCR-3 is executed when a task is transferred to another peer as part of a load

1 This is not true for the opposite case. However, the impact on performance would be significant in case the donator task was lost and restored subsequently, as then all the work – including that split off – would be done again, which could be the whole job in the worst case.

balancing operation. It simply triggers an update to the location of the transferred task by sending an appropriate `Update` message to the coordinator peer.

The event handlers of Protocol 27.3 implement the task tracking logic on the coordinator. The ⟨ Init ⟩ request (TTCR-C-0) is triggered by the environment when the protocol is instantiated. It initializes the *task database* that consists of two relations: $\mathcal{L}$ and $\mathcal{T}$. $\mathcal{L}$ contains pairs $(taskId, location)$. Each such pair associates a task – uniquely identified by its task identifier – with its last known location (in form of a peer identifier). $\mathcal{L}(taskId)$ evaluates to the location associated with $taskId$ or to $\perp$, if no such association exists. $\mathcal{T}$ contains pairs $(taskId, task)$ and is used to lookup the last known task description for a given task identifier. $\mathcal{T}(taskId)$ evaluates to the instance of the Task ADT associated with $taskId$ or to $\perp$, if no such association exists.

TTCR-C-1 implements the only request of the Task-Tracking Checkpoint/Restart module: ⟨ Tracked ⟩. The predicate is computed by checking whether the local peer is the coordinator of the given task and whether an associated entry exists in $\mathcal{L}$. If and only if both requirements are satisfied, the task is tracked by the local peer, and $true$ is returned.

TTCR-C-2 is the most complex handler of the protocol and is executed when an `Update` message is delivered. Although not captured explicitly here, an incoming `Update` message carrying more than one update (as sent by TTCR-2) is transparently translated into several `Update` messages each carrying exactly one update and processed in direct succession. There are three different possible scenarios:

1. The task is unknown, i.e., it has been newly created or all previous `Updates` have been delayed or omitted due to a crash of the emitting peer. In the former case, this is a result of some peer executing TTCR-0 or TTCR-2.

2. A transferral of the task to another peer has been initiated, which is signaled by some peer executing TTCR-3.

3. The task should be tracked no longer as it has been completed by some peer resulting in the execution of TTCR-1.

In all three cases TTCR-C-2 updates the task database accordingly.

If a final location $p_j \neq \perp$ is specified in an incoming `Update`$(t,\ p_i,\ p_f)$ message, the coordinator checks whether peer $p_f$ is still correct by consulting the local membership view. If not, the task is assumed to be lost and its restoration is scheduled by means of a ⟨ Schedule ⟩ request on the Timer instance. The restoration is not performed immediately but delayed for a certain period referred to as the *restoration timeout*, because a transferral of the task to another peer $q$ could have been initiated just before $p_f$ has crashed without being able to sent the corresponding `Update` message of TTCR-3 to the coordinator. In case the task is updated subsequently on $q$, the restoration is canceled as part of the resulting repeated invocation of TTCR-C-2. The restoration timeout is an input parameter to the protocol and should be chosen to be large compared to the average round-trip time of the largest expected `Update` message in the underlying network, which can be in the order of hundreds to thousands of milliseconds in typical area networks spanned by Desktop Grids.

Intuitively, it's clear that if the updates for a task do *not* arrive in the order they are generated, it's, in general, impossible to track a task's location while it moves from peer

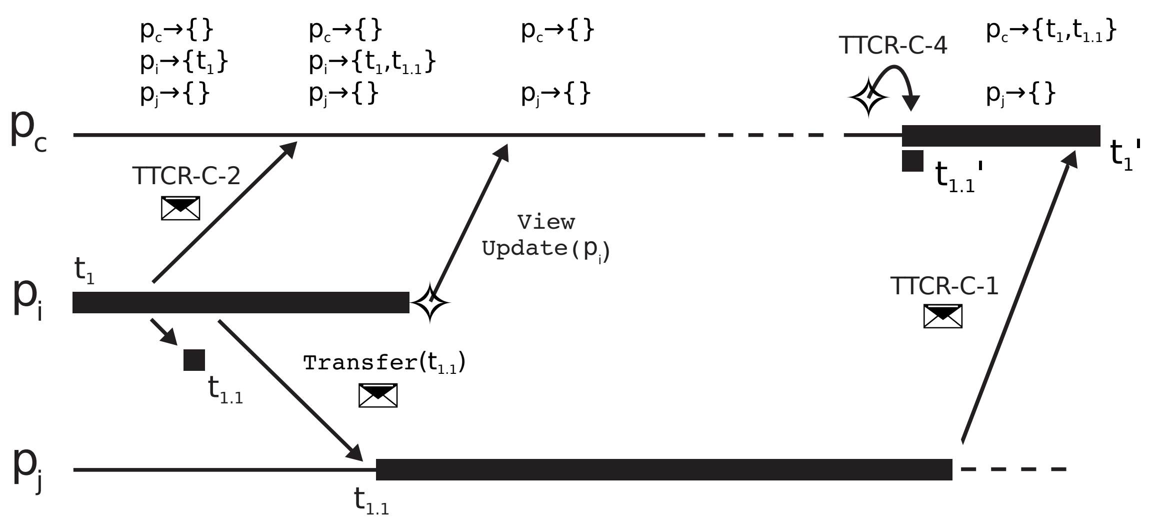


**Figure 27.1:** Scenario where a task is duplicated by COHESION's Task-Tracking Checkpoint/Restart protocol ($t_i \hat{=}$ Task with symbolic identifier $i$, $t_{i.j} \hat{=}$ subtask of task $T_i$, ✉$\hat{=}$ `Update` message emitted by execution of handler with the given name or `Transfer`($t$) message containing task $t$, `ViewUpdate`($p$) $\hat{=}$ ⟨ *pgm*, ViewUpdate ⟩ indication triggered by departing peer $p$).

to peer and its state across intermediate split operations. To avoid that delayed `Update` messages override the assignments of previously delivered but later generated ones, TTCR-C-2 is protected by a guard condition that only accepts `Update` messages that were generated more recently than any `Update` message received so far. Although rejected updates are discarded this way, we will see in the correctness proof given in Section 27.3 that all necessary information to track a task is available to the coordinator due to the fact that update sequences for tasks always match a certain pattern.

The purpose of TTCR-C-3 is to monitor the membership within the task pool's peer group. It is triggered when a peer departs either deliberately or by crashing. The protocol relies on the ⟨ ViewUpdate ⟩ indication emitted by ORBWEB's Peer Group Membership (PGM) module described in Chapter 15. On receipt of such an indication, the coordinator consults the task database and schedules the restoration of all tasks that are assumed to be located on the vanishing peer.

A second scenario where TTCR-C-3 is triggered is when the coordinator receives a `Steal` request from some peer $p$. In that case the coordinator knows, that the PE at $p$ was idle and that the task queue of $p$ was empty (cf. Chapter 26) at the time the `Steal` message has been sent. As above, all tasks that are assumed to be located on $p$, but are probably not as of this indication, are scheduled for restoration.

The restoration of a lost task itself is done by TTCR-C-4, the last event handler of the protocol. It is triggered when the timer, started as part of TTCR-C-2 or TTCR-C-3, expires. The handler recreates the task from the task description stored in the task database for the task identifier used previously as the timer token. The task is enqueued with an entirely new task identifier, and therefore is treated as a *different* task by the protocol. Additionally, the substituted task is assigned to the COMPENSATES attribute. As will be explained in

Chapter 28, this is required for the correctness of the termination detection protocol.

### 27.2.1 Task Duplication

A characteristic feature of the Task-Tracking Checkpoint/Restart protocol is that tasks may be duplicated under rare circumstances. Figure 27.1 depicts such a situation where a task $t_{1.1}$ is unnecessarily restored: After $t_{1.1}$ has been split off on peer $p_i$ and its transferral to $p_j$ has been initiated by the Load Balancing module, $p_i$ crashes before it can send the `Update` message generated by TTCR-3 that signals the transferral to the coordinator. Since the last known location of $t_{1.1}$ is $p_i$, it is assumed to be lost and its restoration is scheduled. As no further update is made to $t_{1.1}$ on $p_j$ during the restoration timeout, the restoration is not canceled in time and finally performed. Consequently, two tasks $t_{1.1}$ and $t'_{1.1}$ are present for the same input.

## 27.3 Correctness

In this section, we prove that our Protocols 27.2 and 27.3 together actually guarantee the properties TTCR1-TTCR3 of the Task-Tracking Checkpoint/Restart module.

### 27.3.1 Eventual Completion

To facilitate the proof of the *Eventual Completion* property (TTCR1), we first need a definition of the different states a tracked task can be in:

**Definition 27.3.1.** *A tracked task can be in two states: manifested and lost. It is called* manifested at peer $p$*, when it is currently processed by the PE hosted by* $p$ *or contained in the local task queue of* $p$*. It is called* manifested*, if it is manifested at any peer* $p \in P$ *or in transit on a communication link wrapped in a* $\mathit{Transfer}$ *message. If a tracked task is* not manifested*, it is called* lost*. A tracked task that has been lost but is manifested again is called* restored*.*

The proof for TTCR1 consists of three steps: First, we show that an invariant holds for tracked tasks and the Task-Tracking Checkpoint/Restart Coordination protocol. Using this invariant, we prove that every tracked task that gets lost is eventually restored. This is finally used to prove that the *Eventual Completion* property (TTCR1) is guaranteed.

**Lemma 27.3.1.** *Let* $\textsc{View}(p_c)$ *denote the membership view at peer* $p$ *as returned by the* $\langle$ *View* $\rangle$ *request of the Peer Group Management module and let* $\textsc{Scheduled}(token)$ *denote the scheduling status for the given* $token$ *as returned by the* $\langle$ *Scheduled* $\rangle$ *request of the Timer module. Then for every tracked task* $t$ *with* $id := \mathrm{ID}(t)$ *the invariant* $\mathcal{L}(id) = p \wedge (p \in \textsc{View}(p_c) \vee \textsc{Scheduled}(id))$ *for some* $p \in P$ *holds.*

*Proof.* We prove the lemma by induction. Let $t$ be a fixed but arbitrary task with $id := \mathrm{ID}(t)$. As $t$ is tracked by assumption $\mathcal{L}(id) = p$ must hold for $t$ and some $p \in P$. This is obvious from Definition 27.1.1 and the body of TTCR-C-1. As apart from initialization in TTCR-C-0, $\mathcal{L}$ is set to some value different from $\perp$ only in the second if-clause of TTCR-C-2, TTCR-C-2 must have been triggered with $u := \textsc{Update}(t,\ q,\ p)$ and arbitrary $q$. In fact TTCR-C-2 could have been triggered more than once, the last invocation, however, must have been for $u$. At that time $p$ was either in the view of $p_c$ or not. In both cases $\mathcal{L}(id)$ is set to $p$. Hence, in case $p \in \textsc{View}(p_c)$, the invariant holds obviously. The same is true for $p \notin \textsc{View}(p_c)$, since in this case ⟨ t, Scheduled ⟩ is invoked for $id$ and consequently $\textsc{Scheduled}(id)$ yields true.

For the inductive step, we assume that the invariant holds at some arbitrary point in time for some $p \in P$ and show that it still holds after every possible protocol step. TTCR-C-1 is a handler with read only behavior with respect to $\mathcal{L}(id)$ and the Timer instance. Hence, it is not relevant here. The same is true for TTCR-C-4. For TTCR-C-2, we have to consider all possible incoming `Update`$(id,\ p_i,\ p_j)$ messages. The possible combinations for $(p_i, p_j)$ are $(\bot,\bot)$, $(\bot,q)$, $(q,\bot)$ and $(q,r)$ for arbitrary $q,r \in P$. $(\bot,\bot)$ is not emitted by any event handler of our protocols and thus is not relevant. For all other combinations $p_j \neq \bot$ holds and hence the invariant holds for the same reasons as in the argument for the induction basis above.

TTCR-C-3 is triggered either if some peer $q$ leaves the group or a `Steal` message from some peer $q$ is delivered. If $p \neq q$ holds, the execution of TTCR-C-3 is not relevant with respect to $t$ as $\mathcal{L}(id) = p$ holds due to our inductive hypothesis. Hence, we assume $p = q$. As $\mathcal{L}$ is not modified by TTCR-C-3 our invariant becomes $p \in \textsc{View}(p_c) \vee \textsc{Scheduled}(id)$. If TTCR-C-3 was triggered due to $p$ leaving the group, then for all entries $(id,p) \in \mathcal{L}$ ⟨ t, Scheduled ⟩ is triggered. As $\mathcal{L}(id) = p$ holds due to our inductive hypothesis, this includes an invocation for $t$, which results in the predicate $\textsc{Scheduled}(id)$ becoming true. Hence, the invariant holds. In case TTCR-C-3 was triggered by a `Steal` message from $p$, the validity of the invariant is also not affected. This is true because $p \in \textsc{View}(p_c)$ holds after the invocation when it held before. The same is true for $\textsc{Scheduled}(id)$. As one of both conditions held before due to our inductive hypothesis, one still holds after the invocation.

Since both the basis and the inductive step of the induction proof have been performed, the proposition holds for all tracked tasks and all valid protocol executions. ■

Next, we show that in case a tracked task is lost, it is eventually restored. To facilitate this, we first refine Definition 27.3.1 to a more precise statement about the states in which a tracked task can be.

**Lemma 27.3.2.** *A task $t$ with $id := \mathrm{ID}(t)$ is either manifested at exactly one peer, contained in a `Transfer` message, or lost.*

*Proof.* This is obvious from the protocols and the fact that a task is restored in TTCR-C-4 under a new identity. ■

As a corollary to Lemma 27.3.2, we can easily specify the conditions under which a tracked task is lost.

**Corollary 27.3.1.** *A tracked task can get lost if and only if the peer where it is manifested crashes or if a `Transfer` message that contains the task is not delivered at the receiver due to a link failure.*

*Proof.* Trivially follows from Lemma 27.3.2. ■

The next step is to prove that a tracked task that got lost is eventually restored. We do this by proving the following two lemmas that together say that the restoration happens under both possible conditions specified by Corollary 27.3.1. Before we start, we give a definition of *idle* and *busy* peers.

**Definition 27.3.2.** *A peer is called* idle*, if and only if the local PE is not executing a task and there are no tasks in the local task queue. Otherwise, it is called* busy*.*

Based on this definition, we prove an auxiliary lemma required subsequently.

**Lemma 27.3.3.** *Every correct peer $p \in P$ with $p \neq p_c$ that is busy becomes idle eventually.*

*Proof.* As long as $p$ is busy, no new tasks are enqueued to $p$'s task queue despite those that are created by splitting[1]. Remember that load balancing is done by the Abstract Random Stealing protocol (see Protocol 26.2) through the emission of `Steal` messages. As can be seen from ARS-1, no such messages are emitted as long as the task queue is not empty or the PE is executing some task. This, however, matches the definition of a busy peer. As $p$ is correct and every input can be processed in finite time, $p$ will become idle eventually. ■

Now, we can go ahead with the following lemma:

**Lemma 27.3.4.** *A tracked task that got lost due to a crash of the peer where it is manifested is eventually restored.*

*Proof.* Let $q$ be the crashing peer and $t$ the task with $id := \mathrm{ID}(t)$ that got lost. We know that the invariant from Lemma 27.3.1 holds for some $p \in P$. We have to distinguish between two cases: $q = p$ and $q \neq p$. We handle both of them separately.

$\mathbf{q = p}$: The crash of $q$ eventually results in a $\langle$ *pgm*, ViewUpdate $\rangle$ indication on $p_c$ as both the superpeer and the coordinator are assumed to be correct. This triggers TTCR-C-3. Since $p$ crashes, $p \notin \textsc{View}(p_c)$ holds. As our invariant must still hold after the execution of TTCR-C-3, $\textsc{Scheduled}(id)$ must be true. The `Update` message that was delivered last on $p_c$ and set $\mathcal{L}(id) = p$ was the last one generated for that $t$, because $t$ was manifested on the crashed peer $p$. Hence, all `Update` messages not yet delivered on $p_c$ do not trigger TTCR-C-2 as they are rejected by its guard. As TTCR-C-2 is the only handler that possibly cancels a scheduled restoration, the scheduled restoration of $t$ is not canceled and $t$ is finally restored.

$\mathbf{q \neq p}$: Let $\mathcal{U}$ be the possibly empty set of `Update` messages not delivered on $p_c$ when $q$ crashes. $|\mathcal{U}|$ is monotonically decreasing over time as $t$ is lost and thus no new `Update` messages are generated. Let $u$ denote the `Update` message that triggers the last invocation of TTCR-C-2. After this invocation the invariant from Lemma 27.3.1 must still hold for some $r \in P$. If $r$ has crashed already when $u$ is processed by TTCR-C-2 than $\textsc{Scheduled}(id)$ must be true, as otherwise the invariant is violated. If $r$ crashes after $u$ is processed by TTCR-C-2, the restoration of $t$ is scheduled in TTCR-C-3 as $\mathcal{L}(id) = r$ due to the invariant and will never be changed again. If $r$ is correct, it will eventually become idle as of Lemma 27.3.3 and emit `Steal` messages to random peer group members. Eventually, such a

1 This is not true for the coordinator $p_c$, as on $p_c$ tasks may also be created by restoration in TTCR-C-4.

message will be sent to and delivered on the coordinator. This triggers TTCR-C-3, which involves the scheduling of the restoration of $t$. Since $u$ is by assumption the last `Update` message triggering TTCR-C-2, the scheduled restoration of $t$ is not canceled and $t$ is finally restored. ∎

**Lemma 27.3.5.** *A tracked task that got lost due to a* `Transfer` *message not being delivered is eventually restored.*

*Proof.* The proof is almost the same as that for case $q \neq p$ in the proof of Lemma 27.3.4. The only difference is that, $\mathcal{U}$ is now the possibly empty set of `Update` messages not delivered on $p_c$ when the critical `Transfer` message is sent. ∎

Finally, we can give the proof for the desired theorem:

**Theorem 27.3.1.** *Every tracked task or a restored substitute is eventually completed.*

*Proof.* There are exactly three possible types of tracked tasks: Those that get never lost, those that get finitely often lost, and those that get infinitely often lost. The existence of a task of the last type, would contradict the proposition. Hence, we show that such a task can not exist.

The overall work required to solve a task is finite by assumption (cf. Section 25.1). This is true for a root task as well. The coordinator $p_c$ is assumed to be correct and hence, makes steady progress. Thus, a tracked task that is never lost must be eventually completed.

A task that is finitely often lost is restored every time it got lost as of Lemmas 27.3.4 and 27.3.5 as a substitute. After it got lost the last time, it never gets lost again and hence becomes a task of the first type.

Now assume that there is a task $t$ that gets lost an infinite number of times. After a sufficiently long period of time, a substitute for $t$ must be the last task since all the others are tasks of the first two types and are therefore eventually completed. According to Lemmas 27.3.4 and 27.3.5 and TTCR-C-4, $t$ or one of its substitutes is restored on the coordinator every time it gets lost. Let $s$ be the substitute created by the last restoration after there are no other tasks left. Since $s$ is the last task, the PE of $p_c$ must be idle and hence starts to process $t$. As $p_c$ is assumed to be correct, $t$ is eventually completed and thus can not be lost again. This is a contradiction to the assumption that $t$ gets lost an infinite number of times. Hence, there is no task that can get lost an infinite number of times. As the other types of tasks are all eventually completed the proposition follows.

∎

### 27.3.2 Safe Split

To facilitate the proof of the *Safe Split* property (TTCR2), we first need to prove two properties that hold for all possible sequences of `Update` messages for a task *after* a split has been performed. For reasons of brevity, we subsequently refer to `Update` messages as *updates*. The properties are defined by the following Lemmas 27.3.6 and 27.3.7. For both, we assume a sequence $\mathcal{U} = u_1, \ldots, u_n$ of updates. This sequence describes the generation order of the updates for a task generated across the peers in $P$ during the distributed computation. Such a sequence exists and is unique for every task due to Lemma 27.3.2.

**Lemma 27.3.6.** *Let $\mathcal{U} = u_1, \dots, u_i, \dots, u_n$ be the sequence of updates generated for task $t$. If $u_i$ is generated due to a split operation by TTCR-2 on peer $p$, then all subsequently generated $u_k$ with $k > i$ must be generated on $p$ and be due to $t$ being split again or completed.*

*Proof.* Since $u_i$ is generated due to a split operation by TTCR-2 on $p$, $t$ must be currently processed by the PE at $p$. Otherwise, it could not have been the target of a split operation. Obviously, $t$ must have been created previously and can be created only once. Furthermore, $t$ can not be migrated after a split as this is possible for tasks in the task queue only. Hence, all updates generated for $t$ subsequently must be generated on $p$ and either be due to another split or the completion of $t$. ■

**Lemma 27.3.7.** *If the conditions of Lemma 27.3.6 hold, then if $u_i$ is omitted (never delivered at $p_c$), then then all $u_k$ with $k > i$ are omitted.*

*Proof.* As of Lemma 27.3.6 all updates $u_k$ with $k \geq i$ are sent by the same peer $p$. This means that they are sent using the same FIFO-Order Perfect Link provided by ORBWEB. These, however, guarantee as of Lemma 16.4.4 that if the `Update` message $u_i$ is sent before another one $u_k$ with $k > i$, $u_k$ is not delivered before $u_i$. As $u_i$ is never delivered by assumption, $u_k$ is also never delivered. ■

This property can be used to prove the *Safe Split* property (TTCR2) of our Task-Tracking Checkpoint/Restart protocol.

**Theorem 27.3.2.** *If a tracked task $t$ is split, updating $\mathrm{I}(t)$ to $I'$ and creating a new task $t^*$ due to an invocation of TTCR-2 and the split operation $\prec(\mathrm{I}(t)) = (I', I^*)$, then either $t$ remains tracked and no newer updates for $t$ arrive, or $t$ and $t^*$ are tracked with $\mathrm{I}(t) = I'$ and $\mathrm{I}(t^*) = I^*$, and $I'$ and $I^*$ together subsume $\mathrm{I}(t)$.*

*Proof.* As can be seen from TTCR-2, the updates for both split products are sent using a single `Update` message. Due to the *Atomic Delivery* property (PGM3) of the PGM module (see Module 15.1), either both updates are delivered on the coordinator or none of them. In the former case, both updates are processed immediately after each other due to the FIFO processing order of our event handlers (cf. Chapter B). Thus, $t$ and $t^*$ are tracked with $\mathcal{I}(t) = I'$ and $\mathcal{I}(t^*) = I^*$ after the second update has been processed. The subsumption of $\mathcal{I}(t)$ by $I'$ and $I^*$ follows directly from using the split operator to create the inputs for $t$ and $t^*$. If the `Update` message is omitted, then $t$ is tracked, as it was tracked before by assumption. Due to Lemma 27.3.7 no further updates are delivered for $t$. As the proposition holds in both possible cases, the proposition holds in general. ■

## 27.3.3 Eventual Subsumption

We first look at the sequence of updates as they are generated while a task runs through its lifecycle.

**Lemma 27.3.8.** *The sequence of* generated *updates $\mathcal{U}$ (as defined above) for any tracked task always matches the EBNF [Ebn96] pattern $Cr, \{M\}, \{S\}, [Cp]$. Where $Cr$ (Create)* *denotes an update generated by TTCR-0 or by TTCR-2 for the split off task, $M$ (Migrate) denotes an update generated by TTCR-3, $S$ (Split) denotes an update generated by TTCR-2 for the donator task, and $Cp$ (Completed) denotes an update generated by TTCR-1.*

*Proof.* The first step for any task $t$ is always its creation as the root task of a job or due to a split. After that $t$ can be optionally migrated any number of times until it is fetched from some task queue by a PE. From that point on $t$ can not migrate any more because only tasks in a task queue can be migrated. Hence, $t$ is subsequently either completed or subjected to a split operation. According to Lemma 27.3.6, $t$ can subsequently only be split again or completed in the latter case. Between each of these steps $t$ can get lost due to a crashing peer or an omitted `Transfer` message. As all parts of the pattern except $Cr$ are optional the resulting sequence matches the pattern as well. ∎

Lemma 27.3.8 makes a statement about the flow sequence of updates as they are generated. However, as has been explained in the protocol description in the previous section, the order of delivery of updates for the same task from different peers on the coordinator is arbitrary. Despite this fact, the following lemma holds:

**Lemma 27.3.9.** *For every tracked task $t$ with $id := \mathrm{ID}(t)$, the sequence of assignments to $\mathcal{T}(id)$ is a prefix of the path in the decomposition tree from $t$'s creation until it gets lost or is completed.*

*Proof.* We have to distinguish two cases: no split happens or at least one split happens.

In case no split happens, only creation, migration, and completion updates are generated. Although, $\mathcal{T}(id)$ may be assigned by each of them, it is never assigned to a different value than it has been assigned to before. Hence, the proposition holds trivially.

In case at least one split is performed on $t$, the situation is more complex. Now all types of updates may occur. However, the order of arrival at $p_c$ of split and completion updates among each other is the same as the order in which they are generated. This is due to the fact that all of them are generated on the same peer according to Lemma 27.3.6 and the FIFO ordering guaranteed by the FIFO-Order Perfect Links used to transmit them. The creation and migration updates, in contrast, may be generated on other peers and hence may arrive in any order, in particular in between subsequent split updates or between a split and the completion update. However, as their update counter is smaller than that of all split updates and the completion update because they have been created earlier, they do not trigger TTCR-C-2 due the guard of that handler that filters out all updates older than the newest received. Hence, the order in which the updates are applied is given by the EBNF rule $[Cr],\{M\},\{S\},[Cp]$. Up to the first split update $\mathcal{T}(id)$ is set to the input assigned on creation of the task. As the split updates are applied in order and the completion update does not modify the assignment, $\mathcal{T}(id)$ is successively set to the inputs along the respective path in the decomposition tree. Hence, the proposition holds. ∎

Using this lemma, we can prove the next lemma. It guarantees that if a tracked task is lost, it is restored in a way such that at least the input space covered by the lost task is covered by the restored task.

**Lemma 27.3.10.** *A tracked task $t$ that got lost is restored as a new task with an input that subsumes the input of $t$ at the time $t$ got lost.*

*Proof.* The restoration of $t$ is guaranteed by Lemmas 27.3.4 and 27.3.5. As of Lemma 27.3.9 $t$ is restored by TTCR-C-4 with an input from the prefix of the path in the decomposition tree from $t$'s creation until it got lost. Such an input, however, subsumes the input of $t$ due to the properties of the split operator (see Section 25.1). ∎

Now, we need to extend this result to arbitrary executions that include all kinds of updates that a task can undergo, in particular split operations.

**Lemma 27.3.11.** *If a tasked $t$ is tracked with input $i = \mathrm{I}(t)$ at time $\tau$, then a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{I(t_1), \ldots, \ldots, I(t_n)\}$ exists starting from some point in time $\tau' \geq \tau$ such that every $t \in T$ is tracked or completed and the inputs $I$ collectively subsume $i$.*

*Proof.* We show this by induction. The proposition holds trivially at time $\tau$, since $\mathrm{I}(t)$ always subsumes itself. For the inductive step, we assume that the proposition holds at some time $\tau' \geq \tau'' \geq \tau$ and show that it still holds after every possible event that happens next. Let $t' \in T$ be the task with $id := \mathrm{ID}(t')$ that is affected by this event. The possible events are a migration of $t'$, $t'$ is subject of a split, $t'$ is completed, or $t'$ gets lost. As we have seen at the end of the proof of Lemma 27.3.9, a migration of $t'$ never changes $\mathcal{T}(id)$. Hence, the proposition still holds after a migration. As can be seen from TTCR-C-2, the same is true for the completion of $t'$, since $\mathcal{T}(id)$ is not assigned in this case. For splits and lost tasks the proposition holds due to Theorem 27.3.2 and Lemma 27.3.10, respectively. Hence, the proposition holds for every possible event. By mathematical induction, we can conclude that the proposition holds starting from some point in time $\tau' \geq \tau$. ∎

Finally, we can give the proof that our protocols satisfy the *Eventual Subsumption* property (TTCR3) of the Task-Tracking Checkpoint/Restart module.

**Theorem 27.3.3.** *If a task $t$ is tracked with input $i = \mathrm{I}(t)$, eventually a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{I(t_1), \ldots, \ldots, I(t_n)\}$ is completed such that the inputs $I$ collectively subsume $i$.*

*Proof.* By premise, $t$ is tracked. Hence, Lemma 27.3.11 applies. As all tracked tasks or their restored equivalents are eventually completed due to Lemma 27.3.1 and the input of every restored substitute as of Lemma 27.3.10 subsumes the input of the original task when it got lost, the proposition follows. ∎

## 27.4 Optimizations

The basic Task-Tracking Checkpoint/Restart protocol described in Section 27.2 ensures due to property TTCR3 that all work that has to be done to complete a job is eventually done. However, certain aspects are not optimal with respect to efficiency and scalability. This includes the amount of work performed twice in case a task is lost and the amount of information to be exchanged with the coordinator peer. In the following, we propose three simple optimizations to improve the basic protocol in these regards.

**Protocol 27.4** COHESION's Task-Tracking Checkpoint/Restart protocol with False Positive Reduction

```
Extends:
  Task-Tracking Checkpoint/Restart with Coordination with False Positive Reduction, instance
  ttcr-c-fpr.

upon event ⟨ pgm, ViewUpdate | $p$, $state$ ⟩ such that $state = \textsc{Left}$ or upon event ⟨ pgm,
Deliver | $p$, Steal[] ⟩ do                                                    ▷ TTCR-C-FPR-0
  for each $(taskId, l) \in \mathcal{L} : l = p$ do
    trigger ⟨ pgm, Groupcast | Locate[$taskId$] ⟩;
    trigger ⟨ t, Schedule | $timeout$, $taskId$ ⟩;

upon event ⟨ pgm, Deliver | Locate[$taskId$] ⟩ do                              ▷ TTCR-C-FPR-1
  call ⟨ pe, Executing | $t_e$ ⟩;
  call ⟨ tq, Lookup | $taskId$, $t_l$ ⟩;
  if $\mathrm{ID}(t_e) = taskId \lor t_l \neq \bot$ then
    $task$ = if $t_l \neq \bot$ then $t_l$ else $t_e$;
    $p_c := \textsc{Coordinator}(task)$;
    trigger ⟨ pgm, Unicast | $p_c$, Update[$task$, pgm.$localPeerId$, pgm.$localPeerId$] ⟩;
```

## 27.4.1 False Positive Reduction

As discussed in the description of the basic protocol in Section 27.2.1, tasks are potentially restored *although* they have never been lost. Depending on how often these *false positives* occur, the associated excess computation may be significant. To reduce the probability of their occurence, the coordinator additionally groupcasts a `Locate` message as part of TTCR-C-FPR-0 in the optimized protocol shown in Protocol 27.4. TTCR-C-FPR-0 replaces TTCR-C-3 of the basic protocol. If the task is present in the task queue or is in execution at the PE of a peer $p$ that delivers the `Locate` message, $p$ responds with a corresponding `Update` message (TTCR-C-FPR-1). On receipt of that message, the coordinator is able to update the location of the supposedly lost task and cancels its restoration (TTCR-C-2). If no peer responds until the restoration timer expires, the coordinator assumes that the task is actually lost and restores it.

## 27.4.2 Periodic Checkpointing

*Checkpointing* is a technique to limit the impact of a system failure by storing a snapshot of the state of a computation, and using that snapshot to restore the execution state in case of failure.

COHESION applies checkpointing on the task level. Note that checkpoints are implicitly created by our basic protocol whenever a task is split. However, the impact on performance can be high, when a task, that has been processed for a long time without being split, gets lost due to a crash of the hosting peer. To limit the damage in such situations, COHESION creates additional checkpoints periodically. Protocol 27.5 shows the pseudo code for this optimization. TTCR-C-PC-1 is triggered periodically. It creates a checkpoint $c$ of the task $t$ currently executed by the local PE and stores it in $\mathcal{C}$ as a pair $(\mathrm{ID}(t), c)$, if

**Protocol 27.5** COHESION's Task-Tracking Checkpoint/Restart protocol with Periodic Checkpointing

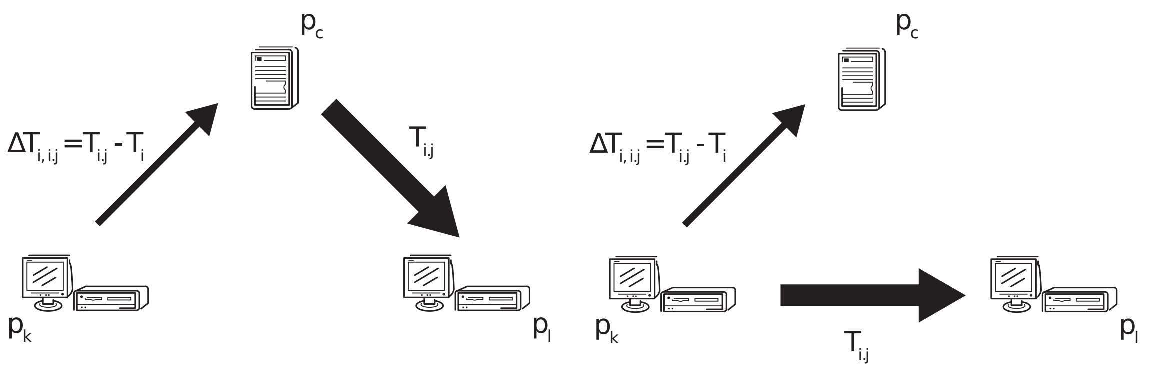
```
Extends:
  Task-Tracking Checkpoint/Restart with Coordination with Periodic Checkpointing, instance
  ttcr-c-pc with checkpoint period.

upon event ⟨ ttcr-c-pc, Init | $period$ ⟩ do                          ▷ TTCR-C-PC-0
  $\mathcal{C} := \varnothing$;
  trigger ⟨ t, Schedule | $period$, 'CHECKPOINT' ⟩;

upon event ⟨ t, Elapsed | 'CHECKPOINT' ⟩ do                           ▷ TTCR-C-PC-1
  call ⟨ pe, Executing | $t$ ⟩;
  if $t \neq \bot$ then
    call ⟨ pe, Checkpoint | $c$ ⟩;
    $id := \mathrm{ID}(t)$;
    if $c \neq \mathcal{C}(id)$ then
      $\mathcal{C}(id) := c$;
      $p_c := \textsc{Coordinator}(task)$;
      trigger ⟨ pgm, Unicast | $p_c$, Update[$t$, pgm.$localPeerId$, pgm.$localPeerId$] ⟩;
  trigger ⟨ t, Schedule | $period$, 'CHECKPOINT' ⟩;

upon event ⟨ t, Elapsed | $taskId$ ⟩ do                               ▷ TTCR-C-PC-2
  $task := \mathcal{T}(taskId)$;
  $\mathrm{ID}(task) := (\pi_1(\mathrm{ID}(task)), PUID())$;
  $\textsc{Input}(task) := \mathcal{C}(taskId)$;
  trigger ⟨ tq, Enqueue | $task$ ⟩;
```

progress has been made compared to the previous checkpoint or if no checkpoint has been created previously. The task is updated accordingly and sent as an `Update` message to the coordinator. This way, only the work performed after the creation of the last checkpoint has to be repeated in case the task gets lost and is restored by TTCR-C-PC-2 (that replaces TTCR-C-4 of the basic protocol) subsequently.

The correctness of the basic protocol is not affected by this optimization, as the creation of a checkpoint is conceptually the same as a split operation, where the input of the donator task is replaced with an input that subsumes the previous input and the split off task does nothing.

### 27.4.3 Differential Updates

The actual structure of `Update` messages is omitted from the Protocols 27.2 and 27.3 for reasons of simplicity. For certain kinds of problems – including SAT solving and other search problems – updates need not to carry the full input, but only the delta to the input of the parent task in the decomposition tree (or to the last checkpoint, if the Periodic Checkpointing optimization is applied). We refer to this optimization as *differential updates*. Figure 27.2 compares the resulting data flows for a centralized task pool on the one hand, and for COHESION's distributed task pool on the other hand. Thanks to differential updates, the bulk of data is transferred in a Peer-to-Peer fashion between the transfer partners when


$p_c$
$\Delta T_{i,i.j} = T_{i.j} - T_i$
$T_{i.j}$
$p_k$
$p_l$
$p_c$
$\Delta T_{i,i.j} = T_{i.j} - T_i$
$p_k$
$T_{i.j}$
$p_l$

**(a)** Centralized task pool **(b)** Decentralized task pool

**Figure 27.2:** Comparison of the data flow in case of a split operation for task $T_i$ on peer $p_k$ with subsequent transferral of the split off task $T_{i.j}$ to another peer $p_l$ for a centralized task pool with implicit fault-tolerance (all communication including task transferals is relayed by the coordinator) on the left and COHESION's distributed task pool with its Task-Tracking Checkpoint/Restart protocol and the Differential Update optimization on the right side ($\Delta T_{i,i.j}$ denotes the delta between the inputs for tasks $T_i$ and $T_{i.j}$).

a split off task is migrated to another peer. The overhead of taking the indirection over the coordinator is eliminated.

Differential updates are feasible due to the fact that all split operations for a task are performed on the same peer (see Lemma 27.3.6). As `Update` messages are sent over ORBWEB's FIFO-Order Perfect Links, the `Update` messages generated by these split operations are delivered in FIFO order on the coordinator. Hence, the inputs of the split products can always be reconstructed from the incremental task descriptions carried by the differential updates.

# 28 Termination Detection

The problem of *distributed termination detection* was independently introduced by Dijkstra *et al.* [DS80] and Francez [Fra80] in 1980. Due to its theoretical and practical importance the problem has received a lot of attention since then. In asynchronous distributed systems with message-based communication the detection of termination is a non-trivial problem as there is neither global time nor full knowledge of the global state available. However, termination detection is a simpler problem than distributed consensus. This is due to the fact that termination is a *persistent* property of the global system state, i.e., once termination is reached the property will never change again [Mat87].

The meaning of the term *distributed termination* is inseparably linked to the way a computation is performed. The latter is called a *computational model*. The canonical computational model used throughout the literature on termination detection is called the *Diffusing Computation Model* (DCM). Using the basic abstractions of Chapter 25 and the ORBWEB abstractions of Part V, the model can be described as follows: Let $\mathcal{G}$ be a peer group with members $P$. Every peer $p \in P$ is either *active*, i.e., at least one local PE is executing some task and/or the local task queue is not empty, or *passive* otherwise. Active peers can become passive at any time by completing the last task that is available locally. Active peers can send `Transfer` messages containing a task in response to load balancing requests. A passive peer can only become active when it receives such a message. The computation is initiated by the coordinator peer. With this computation model distributed termination is defined as follows:

**Definition 28.0.1.** *A distributed computation is* terminated*, if and only if every member of $\mathcal{G}$ is passive and no potentially activating `Transfer` messages are in transit.*

Many protocols have been proposed for termination detection under the DCM. The two most important classes are *parental responsibility* and *wave-based* algorithms. The former detect termination by establishing a tree over the set of participating processes that reflects the order of activation. They use a computation model that is slightly different from the DCM as a peer is allowed to become passive only, if all its child peers are passive. Termination is detected, when the root of the tree becomes passive. Wave-based algorithms take a different approach. They periodically propagate waves of control messages throughout the network to collect information about the global state of the system.

Many algorithms from both of these classes make restrictive assumptions about the distributed system model. Only more recent algorithms are applicable for communication models where messages may arrive out-of-order or may be delayed for arbitrary but finite time [DIR97, Stu02, MVP04, XL96]. Another common restriction is that dynamic environments in which processes are created and destroyed throughout the computation are not fully supported: Some are designed for environments where processes may be created but not

**Module 28.1** Interface of the Termination Detection module

**Module:**

**Name:** Termination Detection, **instance** *td*.

**Indications:**

⟨ *td*, Announce | *id* ⟩: The job identified by *id* has been completed.

**Properties:**

**TD1:** *Accurate Detection:* A job is detected as completed, if and only if a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{\mathrm{I}(t_1), \ldots, \mathrm{I}(t_n)\}$ have been completed such that the inputs $I$ collectively subsume the input $\mathrm{I}(t)$ of the root task.

destroyed [CL82, DS80, MC82]. Others require a process to participate in the termination detection protocol after it has been destroyed [Lai86].

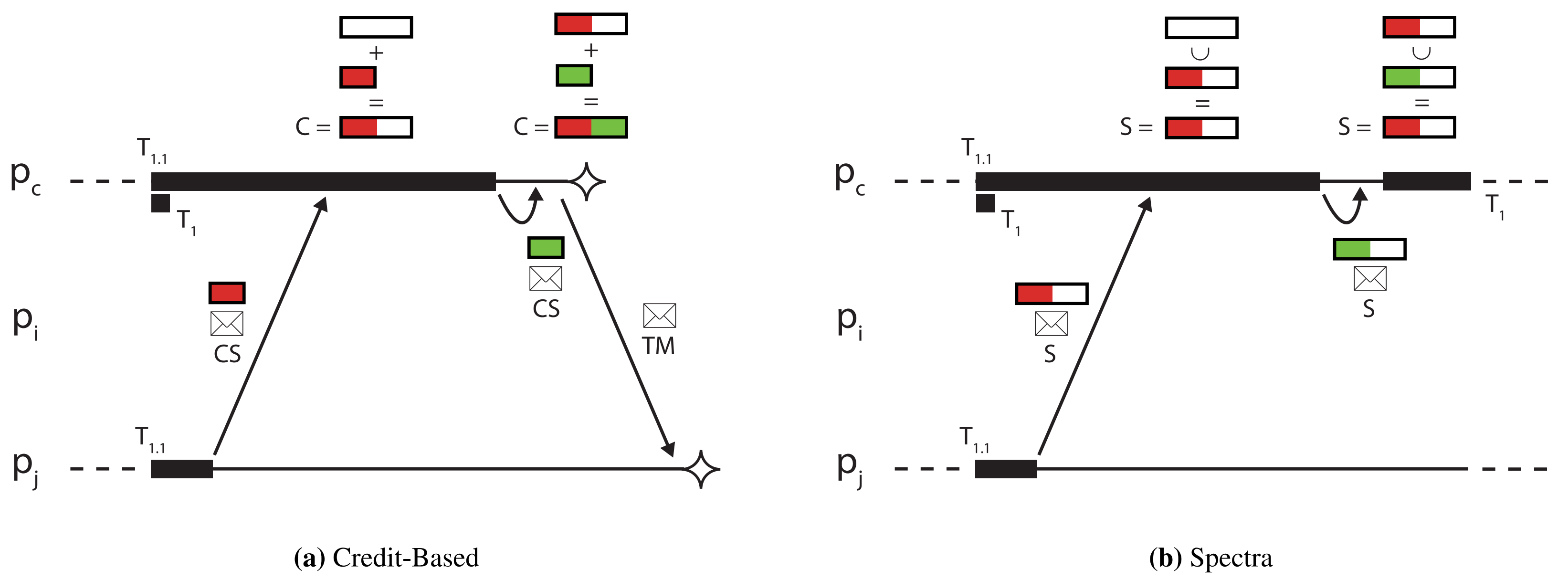
We do not go into the details of existing protocols for the DCM. For a comprehensive overview see [MC98, DTE07]. Considering the fact that COHESION's Task-Tracking Checkpoint/Restart protocol duplicates tasks under certain circumstances, the associated definition of termination as defined above is, despite being applicable, not optimal. This is due to the fact that a corresponding termination detection algorithm announces termination only after *each and every* task created throughout the computation is either lost or completed, which may include any number of restored tasks whose execution is unnecessary, since they are subsumed by tasks that have been completed before. To avoid this inefficiency, we use a different notion of distributed termination that is precisely aligned with the abstractions and properties of our task pool implementation:

**Definition 28.0.2.** *A job with root task $t$ is completed, if and only if a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{\mathrm{I}(t_1), \ldots, \mathrm{I}(t_n)\}$ has been completed such that the inputs $I$ collectively subsume the input $\mathrm{I}(t)$ of the root task.*

Module 28.1 defines the interface and properties of COHESION's Termination Detection module. One might expect that such a module would feature a request to initiate the detection process for a job to be called when it is submitted to the task pool. However, this happens automatically in the task pool implementation of COHESION by listening for ⟨ *tq*, Enqueue ⟩ indications for root tasks. Hence, no such request is provided. ⟨ Announce ⟩ is triggered when the completion of a job with the given identifier has been detected. Besides this indication, the module definition includes a single property called *Accurate Detection* (TD1) that reflects our definition of distributed termination. We will show that our implementation satisfies this property in Section 28.2.3.

## 28.1 Mattern's Credit/Recovery Algorithm

The *Credit/Recovery* termination detection algorithm devised by Mattern [Mat89] is applicable for asynchronous systems and fully supports dynamic environments. Applied to the task pool execution model, the basic operating principle of the algorithm is as follows: Every task $t$ is associated with a certain credit share $C(t) \in (0,1] \subset \mathbb{R}$. Whenever a new task is created by splitting, half of the donator's credit share is subducted and assigned to the split off task. On completion of a task, its associated credit share is sent back to the



**(a)** Credit-Based

**(b)** Spectra

**Figure 28.1:** Comparison of the behavior of Mattern's *Credit/Recovery* and COHESION's *Spectra* termination detection algorithms in case a task is duplicated by COHESION's Task-Tracking Checkpoint/Restart fault-tolerance protocol. The interaction diagrams are continuations of that of Figure 27.1. While the scalar arithmetic of the former is insufficient to handle credit share duplication correctly and therefore announces termination prematurely, the *Spectra* algorithm tolerates such conditions as the addition of intervals is idempotent (CS ≙ Credit Share, S ≙ Spectrum, TM ≙ Termination Message).

coordinator peer. When the accumulated credit at the coordinator equals the *initial credit* that was given to the root task (usually 1), the distributed computation has terminated. A fundamental property of the algorithm is that, with $T$ denoting the set of tasks created so far and $C_0$ being the credit assigned to the initial task, the following invariant holds at all times:

$$\sum_{t_i \in T} C(t_i) = C_0. \tag{28.1}$$

Even though Mattern's algorithm has many benefits including conceptual simplicity and low requirements on the distributed system model, we can *not* use it in its original formulation, as it is not resilient to credit duplication: If a task has been duplicated, both the original task and its duplicate carry the same credit share. Hence, the above invariant is violated and termination would be detected wrongly, as soon as one of both was completed (see Figure 28.1a). Thus, the algorithm is not compatible with COHESION's Task-Tracking Checkpoint/Restart protocol. This inadequacy is a direct consequence of using scalar values as credit shares and can be remedied easily as described in the following section.

## 28.2 Spectra Termination Detection

To retrofit Mattern's algorithm with resilience to task duplication, we conceived a variant called *Spectra*[1] that uses real intervals $[a,b) \subset \mathbb{R}$ as credit shares instead of real values. A credit share is referred to as *spectrum* (pl. *spectra*) in the following. As illustrated in Figure 28.1b – a proof of correctness is given in Section 28.2.3 –, the effect of premature termination announcement caused by credit share duplication can not occur with *Spectra*.

### 28.2.1 Protocol Description

*Spectra* is shown in Protocol 28.2. Every peer $p \in P$ is supposed to run an instance of it. Note that the protocol framework of Mattern remains unchanged in principle.

⟨ Init ⟩ (STD-0) is triggered by the environment and initializes the relations $\mathcal{S}$ and $\mathcal{C}$ to the empty set. $\mathcal{S}$ is used to track the spectra for a job. It contains pairs $(jobId, \{s_1, \ldots, s_n\})$. Each such pair associates the identifier of a job with the set of spectra $s_i, 1 < i < n$ received so far for that job. $\mathcal{S}(jobId)$ is assumed to return the associated set or $\perp$ in case no such association exists. The role of $\mathcal{C}$ and all parts of the protocol that deal with it, will be discussed below and should be ignored for now.

STD-1 is triggered when a root task $t$ is enqueued to the local task queue. As will be shown in Chapter 29, this happens when a job is submitted to the task pool. The body of the handler sets the protocol-specific SPECTRUM attribute of $t$ to the *initial spectrum* of $[0,1)$, which is the counterpart to the initial credit in Mattern's algorithm.

1 As the credit algorithm is also known under the alias *fixed energy* termination detection algorithm and a collection of energy levels is called a *spectrum* in physics, we refer to our variant as the *spectra* termination detection algorithm.

**Protocol 28.2** Spectra Termination Detection

```
Implements:
    Termination Detection, instance std.

Uses:
    Peer Group Management, instance pgm.
    Processing Element, instance pe.
    Task Queue, instance tq.

upon event ⟨ std, Init ⟩ do                                              ▷ STD-0
    $\mathcal{S} := \emptyset$;
    $\mathcal{C} := \emptyset$;

upon event ⟨ tq, Enqueue | task ⟩ such that $\textsc{isRoot}(task)$ do   ▷ STD-1
    $\textsc{Spectrum}(task) := [0,1)$;
    if $(c := \textsc{Compensates}(task)) \neq \bot$ then
        $\mathcal{C}(\pi_1(\mathrm{ID}(task))) := (\pi_1(\mathrm{ID}(c)), \textsc{Spectrum}(c))$;

upon event ⟨ pe, Completed | task ⟩ do                                   ▷ STD-2
    $p_c := \textsc{Coordinator}(task)$;
    trigger ⟨ pgm, Unicast | $p_c$, Recover[$\pi_1(\mathrm{ID}(task))$, $\textsc{Spectrum}(task)$] ⟩;

upon event ⟨ tq, Split | donator, task ⟩ do                              ▷ STD-3
    $[a,b) := \textsc{Spectrum}(donator)$;
    $m := a + \frac{b-a}{2}$;
    $\textsc{Spectrum}(donator) := [a,m)$;
    $\textsc{Spectrum}(task) := [m,b)$;

upon event ⟨ pgm, Deliver | Recover[jobId, spectrum] ⟩ do                ▷ STD-4
    $\mathcal{S}(jobId) :=$(if $(s := \mathcal{S}(jobId)) \neq \bot$ then $s$ else $\emptyset) \cup spectrum$;
    $\textsc{Compactify}(\mathcal{S}(jobId))$;
    if $\mathcal{S}(jobId) \equiv \{[0,1)\}$ then
        if $(c := \mathcal{C}(jobId)) \neq \bot$ then
            trigger ⟨ pgm, Unicast | pgm.$localPeerId$, Recover[$\pi_1(c)$, $\pi_2(c)$] ⟩;
        else
            trigger ⟨ td, Announce | jobId ⟩;
```

When a task $t$ is completed by the PE, STD-2 is triggered. Wrapped up in a `Recover` message, the handler sends the spectrum associated with $t$ and the job identifier for $t$ to the coordinator. The job identifier is extracted from the task identifier pair using the projection operator $\pi$ (cf. Section 25.1 and page 115).

Whenever a task is split, STD-3 dissects the spectrum associated with the donator task into two halves and assigns each of them to one of the split products.

Finally, STD-4 is responsible for checking whether a job has been completed whenever a `Recover` message is delivered. For this purpose, the received spectrum is first added to the set of spectra $s := \mathcal{S}(jobId)$ for the affected job with identifier $jobId$. After that, a call to $\textsc{Compactify}$ condenses $s$ by recursively replacing adjacent spectra $[a,b)$ and $[b,c)$ in $s$ with $[a,c)$ until a fix point is reached. If the resulting set contains a single spectrum only and that spectrum is equal to the initial spectrum $[0,1)$, the job has been completed and the ⟨ Announce ⟩ indication is triggered.

Now, let's come back to $\mathcal{C}$: If the Task-Tracking Checkpoint/Restart protocol would restore tasks in TTCR-C-4 of Protocol 27.3 naively as part of the same job, the termination detection algorithm described so far would not be correct. To understand why this is the case, consider the following example: Assume a job with a root task whose input is a list $(1,2,3,4)$. Assume further that the task $((4),[0,0.5))$, where the first element of the tuple is the input of the task and the second element is the associated spectrum, has been completed already and another one $u = ((1,2,3),[0.5,1))$ is manifested but not completed. Finally, assume that $u$ has been considered wrongly as lost by the coordinator and has been restored as task $v$. $v$ inherits the spectrum from $u$. If $u$ and $v$ are both split subsequently creating the split products $u' = ((1),[0.5,0.75))$, $u'' = ((2,3),[0.75,1))$, $v' = ((1,2),[0.5,0.75))$ and $v'' = ((3),[0.75,1))$, the completion of $u'$ and $v''$ would result in the announcement of termination although the job is not completed as the input of the root task is not subsumed.

To avoid this problem, TTCR-C-4 restores a task as a *new* job whose spectra will be tracked independently. In the following, such a job will be referred to as a *child* of the job the substituted task belongs to. When a child job is completed, the spectrum inherited from the substituted task is attributed as a whole to the parent job. Thus, the problem described above can not occur. $\mathcal{C}$ is used to track the parent/child relationships between jobs. It contains pairs $\left(id_c,\left(id_p,s\right)\right)$. Each such pair associates the identifier $id_c$ of a child job with a pair consisting of the identifier $id_p$ of the parent job and the spectrum $s$ assigned to the substituted task. $\mathcal{C}(jobId)$ returns the associated pair or $\perp$ if no such association exists. Entries of $\mathcal{C}$ are created in STD-1, if the COMPENSATES attribute of the enqueued task has been set in TTCR-C-4 of the Task-Tracking Checkpoint/Restart protocol. These entries are used in STD-4 to send the spectrum associated with a job to its parent job.

## 28.2.2 Spectra Encoding

COHESION's implementation of the *Spectra* protocol uses Java's arbitrary-precision signed decimal numbers[1], to represent the interval boundaries of spectra. Since these boundaries are always sums of reciprocals of powers of 2, the memory consumption per boundary at depth $k$ of the decomposition tree is $k$ bits. For comparison, the credit shares of Mattern's *Credit/Recovery* algorithm are plain reciprocals of powers of 2 and thus can be represented by a single integer consuming $\lceil log_2(k)\rceil$ bits only. During our experiments the depth of the decomposition tree has been in the order of tens. Thus, the space overhead that a spectrum added to `Transfer` and `Update` messages has never been significant.

## 28.2.3 Correctness

To prove the correctness of the *Spectra* protocol, we have to show that it satisfies the *Accurate Detection* property (TP1) of the Termination Detection module. As the announcement of

1 implemented by the `BigDecimal` class in the `java.lang` package

the completion of a job with job identifier $id$ translates to the condition $\mathcal{S}(id) \equiv \{[0,1)\}$ for *Spectra*, TP1 can be formulated as

**Theorem 28.2.1.** *If the root task of a job with job identifier $id$ and input $I_r$ is enqueued to the task queue at any peer $p \in P$, the condition $\mathcal{S}(id) \equiv \{[0,1)\}$ holds, if and only if a set of tasks $T = \{t_1, \ldots, t_n\}$ with inputs $I = \{\mathrm{I}(t_1), \ldots, \mathrm{I}(t_n)\}$ have been completed such that the inputs $I$ collectively subsume $I_r$.*

*Proof.* „$\Rightarrow$“ (Safety): We first assume that no task is restored by the Task-Tracking Checkpoint/Restart protocol. All spectra created throughout the computation are mutually disjunctive. This can be easily shown by induction using the fact that a split operation creates disjunctive spectra in STD-3. As no tasks are restored by assumption, the mapping between the set of spectra and the inputs in the leaf set $\mathcal{L}$ of the decomposition tree is a bijection. If no task for input $i$ from $\mathcal{L}$ is completed, the associated spectrum $s$ is not delivered at the coordinator. Together with the fact that all spectra are mutually disjunctive, it follows that $s$ is missing from $\mathcal{S}(id)$, so $\mathcal{S}(id) \neq \{[0,1)\}$. This is a contradiction to the premise that $\mathcal{S}(id) \equiv \{[0,1)\}$ holds. Therefore, all inputs must have been completed and $I_r$ is subsumed.

Now, we drop the restriction that no tasks are restored and assume that $I_r$ is *not* subsumed. Due to the latter, a task $t$ with input $i$ from $\mathcal{L}$ must have been lost for which $I \cup i$ subsumes $I_r$ holds. Due to our premise $\mathcal{S}(id) \equiv \{[0,1)\}$, some spectrum $s$ with $s \supseteq \textsc{Spectrum}(t)$ must have been delivered on the coordinator instead. Due to the way spectra are propagated by split operations, $s$ can only come from a task with the same input or an ancestor of this input in the decomposition tree. As the split operator guarantees that an input is subsumed by both itself and any ancestor, $i$ was subsumed. This, however, is a contradiction to the assumption that $I_r$ is not subsumed. Hence, the conclusion holds.

„$\Leftarrow$“ (Liveness): The split of inputs and the decomposition of spectra are interlocked in the sense that whenever the one happens, the other happens as well. If we restrict our reasoning to the case that no tasks are restored, the resulting mapping between the set of all generated inputs and that of all generated spectra is a bijection. All inputs $i \in I$ are mutually disjunctive due to our assumption that no task is restored and due to the requirement on the split operator that the created subproblems are disjunct. According to our premise, $I_r$ is subsumed by the inputs in $I$. Thus, all generated tasks must have been completed. Due to the one-to-one correspondence between tasks and spectra, all generated spectra must have been recovered and their union is $[0,1)$.

Now, we drop the restriction that no tasks are restored. A direct consequence of Lemma 27.3.10 and the fact that the split operator creates disjunctive inputs is that a lost task $t$ with input $i$ is restored as a task $t'$ with either the same input $i$ or an input from an ancestor of $i$ in the decomposition tree. Due to the way spectra are distributed along the paths of the decomposition tree, this translates into $\textsc{Spectrum}(t) \subseteq \textsc{Spectrum}(t')$. If $t'$ is completed, $\textsc{Spectrum}(t')$ is delivered at the coordinator. $\textsc{Spectrum}(t) \cup \bigcup_{u \neq t} \textsc{Spectrum}(u) = [0,1)$ held before $t$ was lost and with $\textsc{Spectrum}(t) \subseteq \textsc{Spectrum}(t')$ becomes $\textsc{Spectrum}(t') \cup \bigcup_{u \neq t} \textsc{Spectrum}(u) = [0,1)$ after $t$ has been restored as $t'$. If $t'$ is not completed, it got lost and is restored again as $t''$ due to Lemmas 27.3.4 and 27.3.5. In this case the above argument can be applied again for $t'$ and $t''$. This chain can be repeated until the restored task is completed. Hence, the conclusion holds as well in the case that tasks are restored. ■

# 29 Task Pool Implementation

With the modules and protocols described in Chapters 25-28, the implementation of a task pool, as specified by Module 24.1, becomes almost trivial. As shown in Protocol 29.1, COHESION's implementation makes use of instances of the Task Queue module, the Processing Element module, one of the load balancing protocols described in Chapter 26, the Task-Tracking Checkpoint/Restart protocol, and the *Spectra* termination detection protocol. Note that most of the coordination work is done *under the hood* by having these modules listen to each others' indications and using each others' requests.

The initialization handler, DTP-0, is triggered by the environment. It's only purpose is to initialize the relation $\mathcal{J}$ to the empty set. $\mathcal{J}$ is used to store job definitions during the time they are processed by the task pool and is therefore referred to as the *job database*. It contains pairs $(jobId, job)$. Each such pair associates the identifier of a job $jobId$ with the $job$ itself. $\mathcal{J}(jobId)$ is assumed to return the associated job or $\perp$ in case no such association exists.

DTP-1 is triggered when a job is submitted to the task pool. First, a job identifier is created and used as the task identifier for the root task that is extracted from the Job ADT instance using the TASK attribute. Before the root task is enqueued to the local task queue, the job is stored in the job database.

**Protocol 29.1** Distributed Task Pool protocol

```
Implements:
    Task Pool, instance dtp.

Uses:
    Task Queue, instance tq.
    Processing Element, instance pe.
    Load Balancing, instance lb.
    Task-Tracking Checkpoint/Restart, instance ttcr.
    Spectra Termination Detection, instance std.

upon event ⟨ dtp, Init ⟩ do                                    ▷ DTP-0
    $\mathcal{J} := \emptyset$;

upon event ⟨ dtp, Submit | $job$ ⟩ do                          ▷ DTP-1
    $t := \textsc{Task}(job)$;
    $\mathrm{ID}(t) := (jobId := PUID(), \perp)$;
    $\mathcal{J}(jobId) := job$;
    trigger ⟨ tq, Enqueue | $t$ ⟩;

upon event ⟨ std, Announce | $jobId$ ⟩ do                      ▷ DTP-2
    trigger ⟨ dtp, Completed | $\mathcal{J}(jobId)$ ⟩;
```

When the completion of a job is announced by the termination detection protocol using the ⟨ Announce ⟩ indication, DTP-2 simply translates the indication into a ⟨ Completed ⟩ indication. For this purpose, the provided job identifier is resolved into the submitted job instance using the job database.

The fact that property TP1 of the Task Pool module is actually guaranteed by the above implementation follows directly from property TTCR3 of the Task-Tracking Checkpoint/Restart module and property TD1 of the Termination Detection module, which both have been proven to be satisfied by our protocols.

# 30 Performance Evaluation

To substantiate the efficiency of Cohesion's distributed task pool in highly demanding Desktop Grid environments, we conducted a performance study on a heterogeneous Desktop Grid. Scalability and performance of the task pool are proven in the presence of volatility and faults by means of a synthetic benchmark application that mimics the characteristics of ISPs. Before the results of the evaluation are presented in Section 30.2, the evaluation methodology and the testbed setup are described in the following section.

## 30.1 Methodology and Testbed Setup

We conducted our evaluation on a dedicated testbed consisting of 40 hosts distributed over three firewalled Fast Ethernet Local Area Networks (100 Mbit/s nominal bandwidth) connected by a campus network as depicted in Figure 30.1. The hard- and software setup is summarized in Table 30.1. All hosts run up to four Cohesion peers in parallel and a single machine is configured to additionally serve as an Orbweb superpeer. For experiments with

| **Type** | **Hardware** | | **Software** | | **RAPI**[1] |
|---|---|---|---|---|---|
| | CPU | Memory | OS | Kernel | |
| **I** | AMD®Athlon™64 X2<br>2 Cores @ 2.4GHz<br>512KB Cache / Core | 3GB | Linux | 2.6.22-14<br>(generic) | 0.54 |
| **II** | Intel®Xeon™<br>2 Processors @ 2.67GHz<br>512KB Cache / Processor | 2GB | Linux | 2.6.22-9 | 0.29 |
| **III** | Intel®Pentium™D<br>2 Cores @ 3.40GHz<br>2048KB Cache / Core | 2GB | Linux | 2.6.23<br>(gentoo-r8) | 0.51 |
| **IV** | Intel®Core™2 Q6600<br>4 Cores @ 2.40GHz<br>2048KB Cache / Core | 8GB | Linux | 2.6.22-14<br>(server) | 1.0 |

**Table 30.1:** Hardware and software configuration of the hosts of our testbed. We define the *Relative Application Performance Index* (RAPI) as $RAPI(p;I) := (1/|I|)\sum_{i\in I} T_{Seq}(p_{ref},i)/T_{Seq}(p,i)$, where $I$ is a representative subset of the SAT benchmark instances used in Section 37, $T_{Seq}(p,i)$ is the sequential runtime of instance $i$ on host $p$ and $p_{ref}$ is the fastest host (Type IV).

1 The RAPI is relevant to the performance evaluation of Satciety in Chapter 37 only.

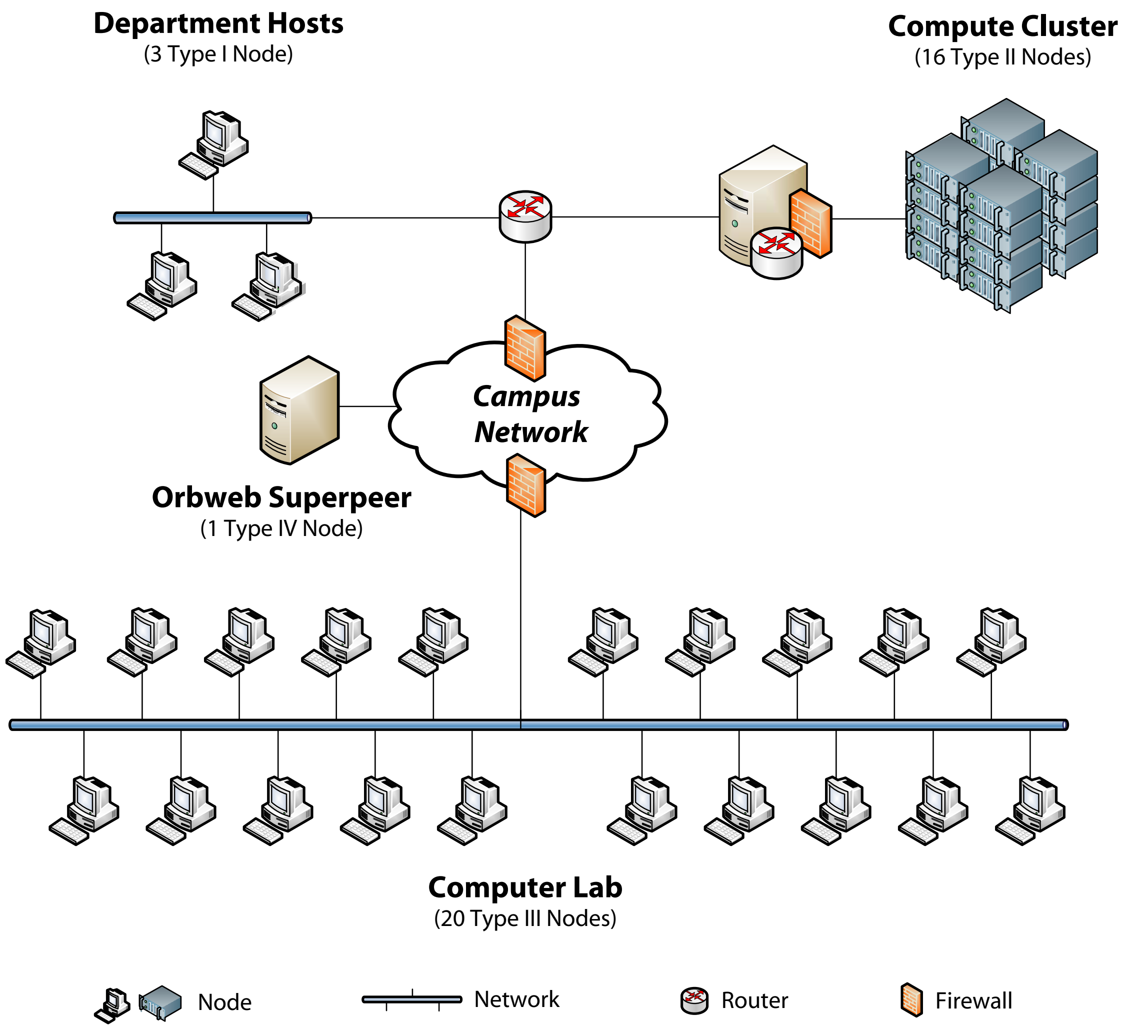


**Figure 30.1:** Testbed Desktop Grid spanning 40 nodes and three networks

more than 40 peers, the additional peers are scattered over all hosts to distribute load fairly, i.e., for a setup with 120 peers each host runs three peers. All peers were run on *Sun JRE* v1.6.0_12 JVMs. Evaluation runs were repeated several times (30 times for the distributed task pool, ten times for provisioning). The presented confidence intervals are based on a 95% confidence level.

### 30.1.1 Benchmark Description

In principal, we could have performed our evaluation by means of Satciety, the distributed SAT solver realized on top of Cohesion described in Part VIII. However, the performance of parallel SAT solvers is heavily influenced by algorithmic effects like work-anomalies [Blo06] and superlinear speedups (see Appendix A). To get a clear understanding of the potential and limits of our distributed task pool, we assess its performance using a synthetic benchmark that is not subject to these effects while still reflecting the characteristics of

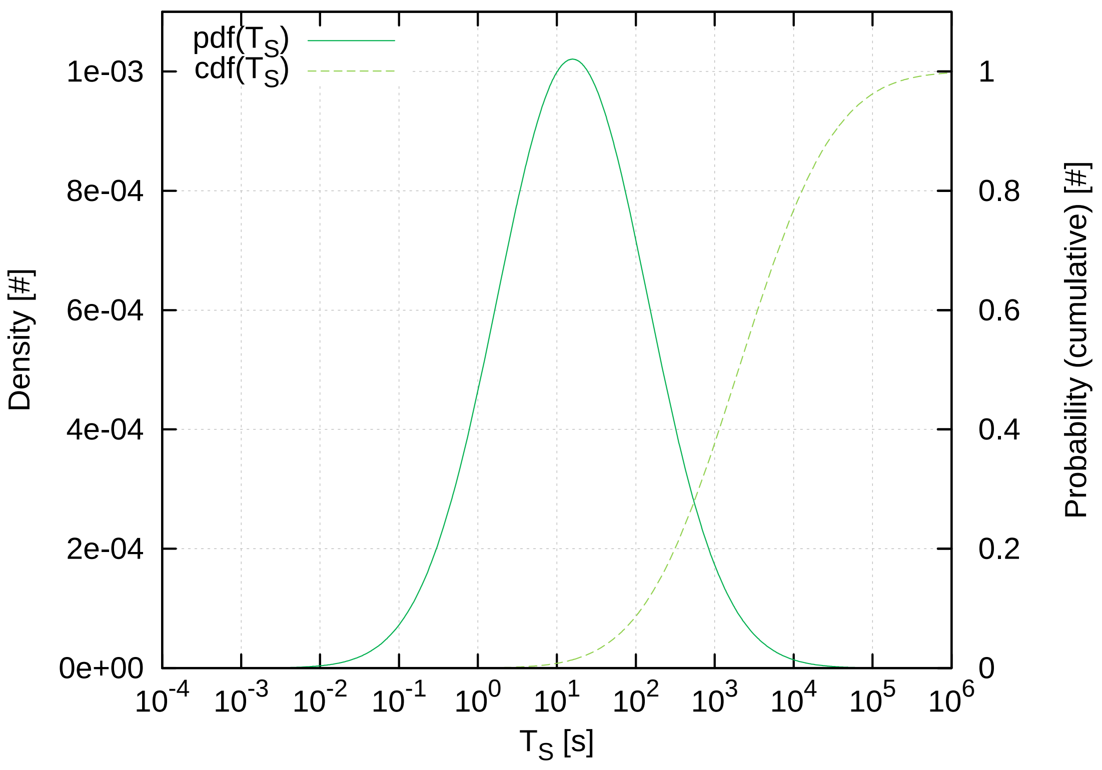


**Figure 30.2:** Probability density (pdf) and cumulative distribution (cdf) functions for the log-normal session time distribution for $\mu = 7.6$ and $\sigma = 2.2$ (note the logarithmic scale of the x-axis) after Wolski *et al.* [WNB07]

ISPs. Furthermore, by using a less resource demanding synthetic benchmark, a sufficiently large number of peers can be hosted on a single physical host to demonstrate the potential of our approach.

In our synthetic benchmark a task is defined by the number of milliseconds $T$ a processor must wait to process the task. Splitting a task $Task_A$ is performed by subtracting a random fraction $T_F$ of the remaining wait time $T_R$ and creating a new task $Task_B$ for $T_F$:

$$Task_A\{T_R\} \overset{Split}{\rightarrow} (Task_{A'}\{T_R - T_F\}, Task_B\{T_F\}). \tag{30.1}$$

### 30.1.2 Volatility Simulation

Wolski *et al.* have shown [WNB07] that machine availability in Desktop Grids is best described by a log-normal distributed session time ($T_S > 0$) with the probability density function

$$pdf(T_S;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma T_S} e^{\frac{-(lnT_S-\mu)}{2\sigma^2}} \tag{30.2}$$

with parameters $\mu = 7.6$ and $\sigma = 2.2$ (see Figure 30.2). We therefore configured peers to exhibit the corresponding random session times ($s$ = seconds)

$$T_S(\nu) = e^{7.6+2.2\nu}s \tag{30.3}$$

with $\nu$ being a random variate drawn from the normal distribution with mean 0 and standard deviation 1. $T_S$ is log-normal distributed with mean 22471 s ($\approx 6$ hours), standard deviation 251709 s ($\approx 3$ days) and the quartiles[1] 453 s ($\approx 8$ minutes), 1998 s ($\approx 33$ minutes), and 8812 s ($\approx 2$ hours). These values illustrate that the distribution is highly skewed towards low values. On the one hand, this means that there is considerable volatility despite the high mean session time. On the other hand, the high value of the upper quartile implies that we can find a stable peer for performing the coordination tasks, i.e., termination detection and fault-tolerance, with high probability.

Volatility is *simulated* by having peers join the underlying computation group, participate in the computation for $T_S$ seconds, leave the group and rejoin with a new identity immediately after discarding the peer's internal state. Peer failures are modeled by dropping tasks with a given probability $P_{Error}$ when leaving the computation group. Unfortunately, to our knowledge, there is no study available yet that describes the reasons for host unavailability and quantifies their respective share in total unavailability. Thus, we have to use an estimated value. We have chosen a supposedly high default error probability of $P_{Error} = 1\%$ meaning that on average 1 out of 100 peer departures is attributed to failure resulting in dropping tasks currently located at the departing node.

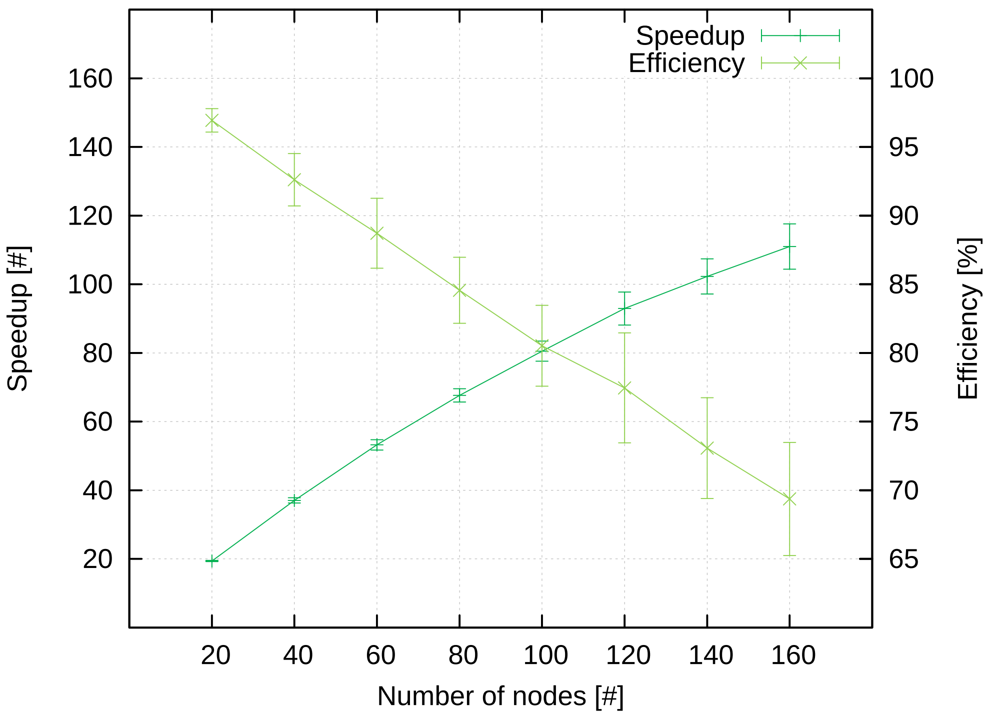
## 30.2 Results

### 30.2.1 Scalability

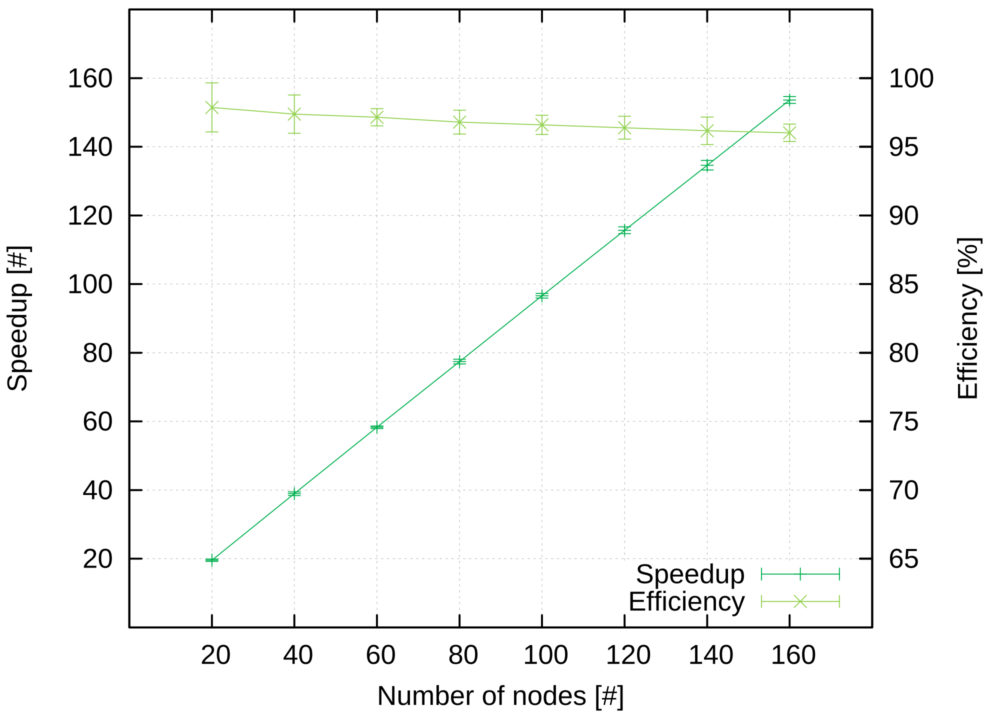
To assess the scalability of the task pool, we use two standard metrics: *speedup* $S$ and *parallel efficiency* $E$ [2]. The runtime of the best sequential algorithm for a given problem size used as reference to compute both metrics is the initial overall wait time $T_{Initial}$. Figure 30.3a shows *strong scalability* metrics for $T_{Initial} = 600s$ and up to 160 non-volatile peers. While the speedup is almost optimal for small peer counts, it increasingly deviates from perfect speedups for larger node counts resulting in a speedup of $111.0 \pm 6.6$ and a corresponding efficiency of roughly $69.4\% \pm 4.1\%$ for 160 peers. As can be seen from Figure 30.4, the reason for this behavior is the increasing parallel overhead caused by idle peers at the beginning and the end of the computation. The time required to find an active peer for work-stealing at a given point in time is proportional to the number of active peers at that time. Thus, work diffusion at the beginning of the computation requires time $T_{Growth} \propto \log(p)$ (where $p$ is the number of participating peers) until all peers are active. For the same reason, the end of the computation is governed by exponential decay creating an analogue dependency

1 The second quartile is equivalent to the *median*.

2 For an introduction to scalability and parallel performance metrics see Appendix A

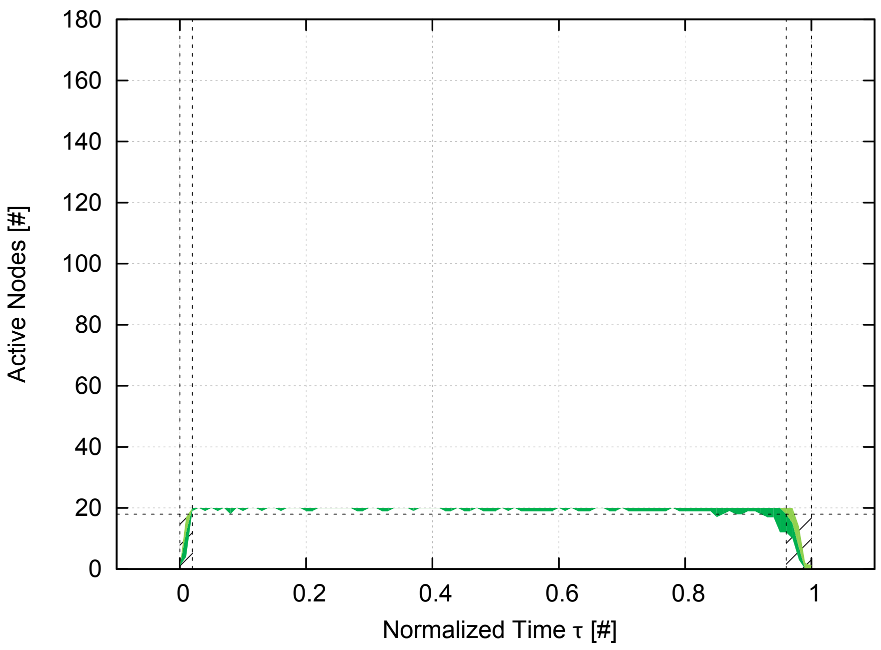


**(a)** Strong scalability for $T_{Initial} = 600s$

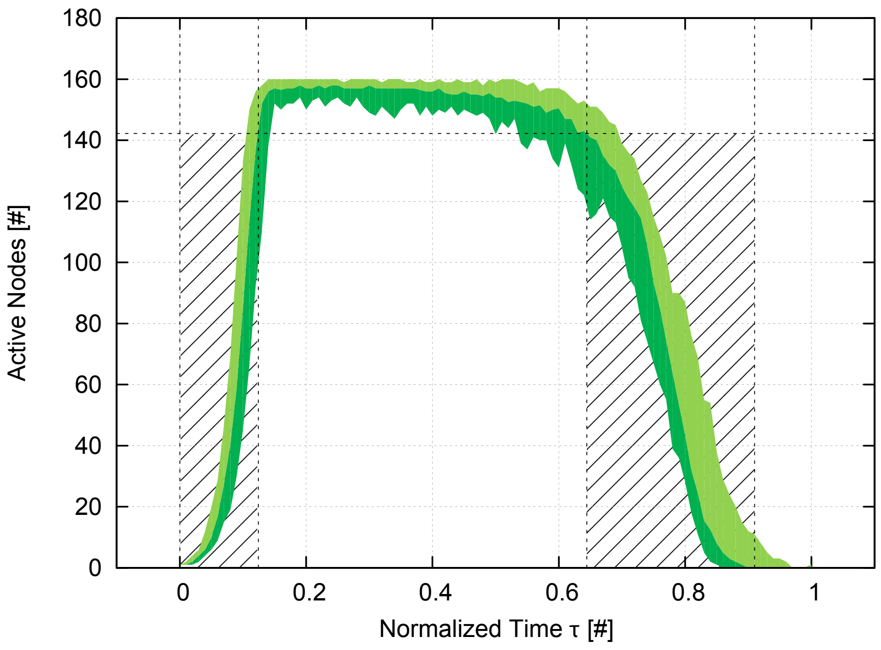


**(b)** Weak scalability for $T_{Initial} = p\,600s$

**Figure 30.3:** Scalability of COHESION's distributed task pool for the synthetic benchmark application



**(a)** Node activity over normalized time for $p = 20$ nodes



**(b)** Node activity over normalized time for $p = 160$ nodes

**Figure 30.4:** Node activity over normalized time $\tau(t;p) = \frac{t}{T(p)}$ (where $T(p)$ denotes the parallel execution time for $p$ processors) for the synthetic benchmark application in 20 and 160 node setups for $T_{Initial} = 600s$. The figures show the minimum, mean, and maximum number of active nodes at normalized time $\tau$ within the dataset collected over 30 runs. Shaded are those periods where the number of active nodes is below 90% of the number of available nodes.

$T_{Decay} \propto \log(p)$. However, the impact of these overheads on overall efficiency depends on the ratio between problem size and peer count.

The *weak scalability* ($T_{Initial} = p600s$) of the system is much better. As can be seen in Figure 30.3b, speedups grow linearly for increasing peer count resulting in a slowly dropping parallel efficiency between $97.9\% \pm 1.8\%$ for 20 and $96.0\% \pm 0.6\%$ for 160 peers.

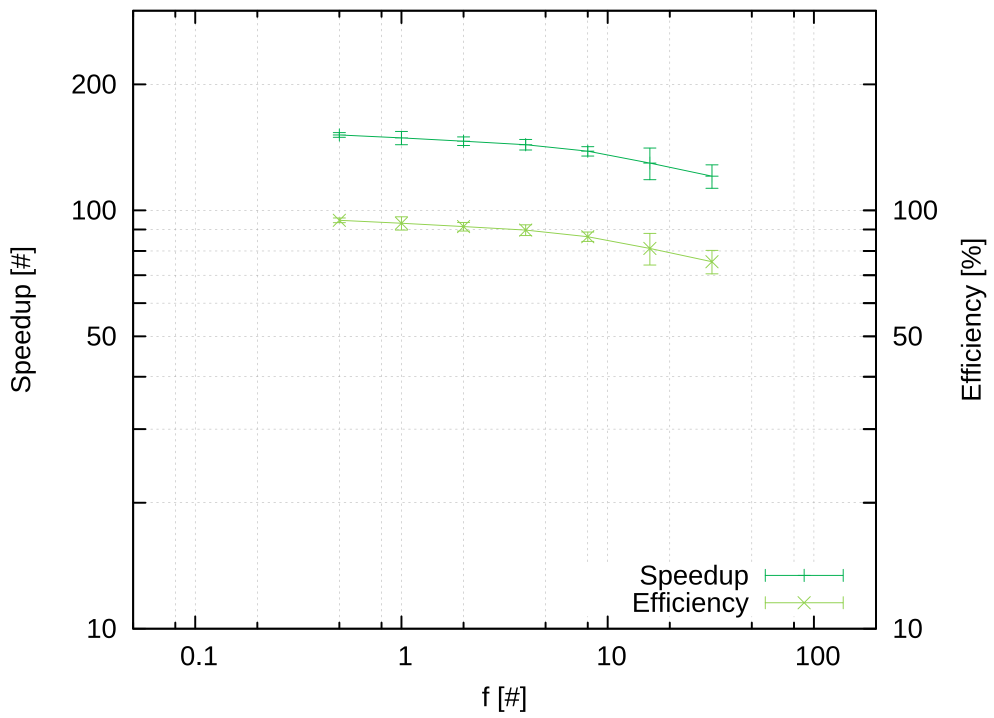
### 30.2.2 Volatility Tolerance

Tolerating deliberate and failure induced peer departures is one of the most important requirements for a Desktop Grid Computing system. Besides correctness, these factors may also severely impair performance. To substantiate that COHESION doesn't suffer from such inadequacies, we measured speedup and efficiency for the synthetic benchmark application under volatility.

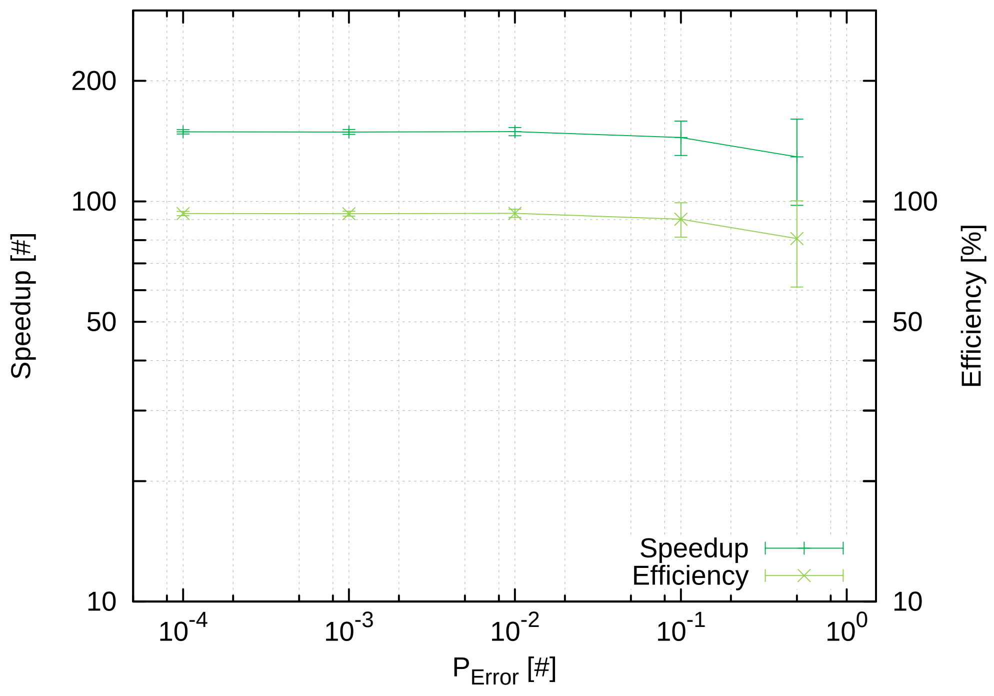
Figure 30.5a depicts the impact of a varying mean session time $T_S^* = \frac{1}{f}T_s$ with $f > 0$ for a fixed error probability of $1\%$ in a 160 peer setup and a problem size of $N = 160 \cdot 600s = 96{,}000s$. For $f = 1$ — resulting in the session time distribution described in Section 30.1 — the speedup is $149.0 \pm 5.5$ and the efficiency is $93.1\% \pm 3.4\%$. The penalty when compared to the non-volatile setup from above is roughly $3\%$. For increasing $f$, speedups drop moderately. Even in the most demanding scenario evaluated here with a 32 times shorter mean session time, we still get a considerable speedup of $120.6 \pm 7.8$ or an efficiency of $75.4\% \pm 4.9\%$.

Figure 30.5b shows the performance penalties for different error probabilities $P_{Error}$ in a 160 peer setup with the real-world session time distribution ($f = 1$). For $P_{Error} = 1\%$, we see a speedup of $149.4 \pm 3.5$ and an efficiency of $93.3\% \pm 2.2\%$. Decreasing the error probabilities further results in nearly no further improvement. Noticeably, increasing $P_{Error}$ to as much as $50\%$ still yields an acceptable speedup of $129.2 \pm 31.4$. The large error of $31.4$ is due to the fact that departures are comparatively infrequent so that the size $T$ of the dropped task becomes decisive for the actual impact on efficiency.

Besides the dominating overhead associated with work repetition and restoration costs for lost tasks, a small fraction of the difference of roughly $3\%$ in efficiency between the non-volatile ($96\%$) and the volatile setup ($93.1\%/93.3\%$) can be attributed to the slightly smaller effective group size resulting from the small pause when peers leave and rejoin the computation group.



**(a)** Performance for varying mean session time $T_S^* = \frac{1}{f} T_s$



**(b)** Performance for varying failure probability $P_{Error}$

**Figure 30.5:** Volatility tolerance of COHESION's distributed task pool

# Part VIII

# Distributed Satisfiability Solving

SATCIETY is a distributed-memory parallel SAT solver for Peer-to-Peer Desktop Grids. It is based on guiding path decomposition realized on top of COHESION's distributed task pool. SATCIETY employs a multi-stage preprocessing pipeline to compactify and simplify SAT formulae before they are efficiently distributed within the Desktop Grid via *BitTorrent*. A state-of-the-art CDCL solver core is amended with a memory manager that protects peers from memory exhaustion. Solver cores exchange knowledge in an adaptive topology-aware distributed dynamic learning scheme. SATCIETY is to the best of our knowledge the first parallel SAT solver for Grid environments that employs a scalable decentralized execution model. Despite the demanding conditions prevailing in Desktop Grids, SATCIETY achieves considerable speedups compared to state-of-the-art sequential SAT solvers.

## Related Publications

[SB10b] SCHULZ, Sven and BLOCHINGER, Wolfgang: **Parallel SAT-Solving on Peer-to-Peer Desktop Grids**. *Journal of Grid Computing* (2010), Bd. 8(3):S. 443–471

[SB10a] SCHULZ, Sven and BLOCHINGER, Wolfgang: **Cooperate and Compete! A Hybrid Solving Strategy for Task-Parallel SAT Solving on Peer-to-Peer Desktop Grids**, In: *Proc. of the International Conference on High Performance Computing & Simulation (HPCS 2010), Workshop on Parallel Satisfiability Solving (WPSS 2010)*, IEEE Computer Society, Caen, France, S. 314–323

SAT is the problem of finding a variable assignment such that a given Boolean formula evaluates to TRUE, respectively to prove that no such assignment exists. SAT was the first problem shown to be NP-complete [Coo71]. Besides this central role in theoretical computer science, many real world problems could have been tackled in recent years by encoding them as SAT instances. Prominent examples can be found in electronic design automation [BCCZ99, VB01, SBSV96], artificial intelligence [KS92], scheduling [CB94], and cryptography [MM00].

Despite the tremendous improvements in SAT solving methods achieved since the first efficient SAT solving algorithm was introduced by Davis *et al.* [DP60, DLL62], there are still unsolved SAT problem instances in all major application fields. Particularly, the ever increasing complexity of chip designs is a source of extremely large and hard SAT problem instances which are far too complex to be solved by state-of-the-art sequential SAT solvers. Thus, parallel computing is a line of research that promises to enable further improvements and to allow for solving previously unsolvable problems. Our specific goal is to use the massive computational power of Desktop Grids to achieve a significant performance boost.

Our work shows that highly optimized parallel search algorithms like SAT solving are prime examples of parallel applications that significantly take advantage of the unrestricted interaction patterns enabled by retrofitting the Desktop Grid approach with Peer-to-Peer methods. By enabling parallel SAT solving on Desktop Grids, we contribute to settling the question which classes of problems can profit from Desktop Grid Computing. In particular, we demonstrate that Desktop Grids are not only suitable for embarrassingly parallel but also for High Performance Computing applications.

The rest of this part is organized as follows: In Chapter 31, we introduce the SAT problem and give a brief account of state-of-the-art SAT solving methods. Chapter 32 summarizes existing parallelization techniques. Subsequent to an overview of SATCIETY's architecture in Chapter 33, we give a detailed description of its inner workings in Chapters 34-36. We report on performance measurements in Chapter 37 and discuss related work in Chapter 38.

# 31 The SAT Problem

The Boolean satisfiability (SAT) problem asks whether one can find a variable assignment $\phi$ for a Boolean formula $F$ such that $F|\phi$ evaluates to TRUE.

We assume that $F$ is given in *conjunctive normal form* (CNF). In CNF, a formula is composed of conjunctions ($\wedge$) of *clauses*. A clause is the disjunction ($\vee$) of one or more *literals*. A literal is either *positive* or *negative*. The former is a variable ($v$), the latter a negated variable ($\neg v$). The *complement* of a literal is its negated literal. By convention, variables are numbered consecutively and are represented as indexed variables $x_i$ with $i \in \{1,\dots,n\}$. Note that all Boolean formulae can be transformed into the CNF representation. A *variable assignment* $\phi$ is a tuple $(\phi_1,\dots,\phi n)$ that assigns a Boolean value to all variables, i.e., $x_1 \to \phi_1,\dots,x_n \to \phi_n$. If not all variables are assigned one speaks of a *partial assignment*.

Consider the following Boolean CNF formula:

$$F = (x_1 \vee x_3) \wedge (x_2 \vee \overbrace{\neg x_3}^{literal}) \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{clause} \wedge x_3. \tag{31.1}$$

The variable assignment $\phi = (\text{FALSE},\text{TRUE},\text{TRUE})$ represents a *satisfying assignment* of $F$. For brevity, one often uses literals which are implicitly assumed to be $true$ to describe a variable assignment. Thus, the above assignment can be expressed as $(\overline{x_1},x_2,x_3)$.

A fundamental property of a formula in CNF is that it is satisfiable iff in each clause at least one literal evaluates to TRUE. If for a clause all but one literal have already been assigned to FALSE, the remaining literal must be assigned to TRUE in order to satisfy the clause. Such clauses are called *unit clauses*. A situation when all literals of a clause are assigned to FALSE is called a *conflict*, and the clause is called a *conflicting clause*.

## 31.1 DPLL/CDCL-Based SAT Solving Algorithms

The original *Davis-Putnam-Logemann-Loveland* (DPLL) SAT solving algorithm [DP60, DLL62] still serves as the algorithmic framework of modern complete[1] SAT solvers. However, in recent years sophisticated heuristics have been incorporated which are capable to significantly prune the search space for a wide spectrum of problem instances. Most beneficial advances have been achieved by employing *conflict driven backtracking* and *dynamic clause*

1 In contrast to *incomplete* solvers, *complete* solvers are able to prove unsatisfiability of a SAT formula.

*learning* [MSS96]. Solvers incorporating these techniques are referred to as CDCL solvers (CD $\hat{=}$ Conflict Driven, CL $\hat{=}$ Clause Learning).

Figure 31.1 outlines the structure of the modern CDCL SAT solving algorithm. We will restrict our discussion of the CDCL algorithm to a top-level treatment. An in-depth description can be found in [ZM02] or [Mit05]. Basically, the CDCL algorithm is a search process with backtracking. Partial variable assignments are speculatively extended to find a satisfying assignment. The procedure `decide` determines according to a heuristic [HV95, MMZ+01] which unassigned variable should be chosen next to extend the current partial variable assignment. Each such decision is recorded on an *assignment stack* along with an associated *decision level*. The decision level of the first decision made is 1. The procedure `propagate` infers additional assignments that are logical consequences of the current partial variable assignment using a technique called *unit propagation*: After making a new decision, some clauses may have become unit clauses, which imply new assignments as explained above. Such deduced assignments are called *implications*. They are recorded immediately after the decision on the assignment stack at the current decision level. Thus, the assignment stack tracks the current state of the search process. Unit propagation terminates when either no more unit clauses exist or a conflict occurs. In the first case, a new decision is made starting the next decision level. In the second case, the conflict is analyzed and resolved by the procedure `analyze_conflict`. Basically, it performs two tasks:

- **Dynamic Clause Learning**: A new clause called *lemma* is constructed by analyzing the reasons for the current conflict. A lemma reflects a minimal subset of the current assignments that implies the conflict. When appended to the input formula, a lemma prevents the search process from reproducing the same conflict in other regions of the search space. Problem clauses and lemmas constitute the *clause database*. There are different schemes for deriving lemmas [ZMMM01]. However, lemmas are always inferred by resolution and are thus logical consequences of the clause database which can be added to the input formula without affecting the correctness of the CDCL algorithm. For the same reason, lemmas can be safely removed from the clause database, e.g., for saving memory.

```
1: procedure CDCL
2:  while TRUE do
3:   if DECIDE() == VARIABLE_ASSIGNED then
4:    while PROPAGATE() == CONFLICT do
5:     if current_level == 0 then
6:      return UNSAT
7:     else
8:      new_level = ANALYZE_CONFLICT()
9:      BACK_TRACK(new_level)
10:  else
11:   return SAT
```

**Figure 31.1:** Top-level structure of the CDCL algorithm (DPLL with Dynamic Learning and Conflict Driven Backtracking)

- **Conflict Driven Backtracking**: By construction, a lemma is initially a conflicting clause. The *backtracking level* is determined as the lowest level at which the lemma becomes a unit clause. Note that at this level the current conflict is also resolved. The procedure `back_track` releases all assignments recorded on the assignment stack up to the computed backtracking level. The newly added lemma, which is now a unit clause, takes the search to a new direction.

When backtracking reaches decision level 0, the current lemma forces a variable assignment and potentially additional implications at level 0. Such variable assignments are called *top-level assignments*. Since top-level assignments do not depend on any decision, they are a necessary condition for the formula to be satisfied and are fixed for the rest of the search process. As a consequence, a conflict at decision level 0 (*top-level conflict*) cannot be resolved by releasing assignments. In that case the input formula is unsatisfiable. In contrast, if all variables have been assigned without a conflict, the input formula is satisfiable.

Although not included in the algorithm outline shown in Figure 31.1 for reasons of simplicity, *restarts* [GSC97] and *clause deletion* are two important techniques employed by all modern CDCL SAT solvers. Extensively adding lemmas to the input formula can slow down the deduction process and thus potentially outweigh the performance improvements of dynamic learning. A solution to this problem is to periodically delete learned clauses selected according to a heuristics. During a restart the search process is canceled by backtracking to the root of the search tree and immediately restarted keeping knowledge – typically the learned clauses – from the previous run. Restarts are performed periodically and frequently in state-of-the-art SAT solvers [Hua07] in order to prevent the search process from getting stuck in an infertile part of the search space. A crucial point when realizing restarts together with clause deletion is to preserve completeness [BMS00]. A solver is called *complete* if it eventually terminates either delivering a solution or indicating unsatisfiability. A prominent technique to preserve completeness of a solver that performs restarts and clause deletion is to gradually increase the restart interval.

# 32 Parallel SAT Solving

Stimulated by a lack of significant progress in sequential SAT solving during the last 10 years and the recent shift to multicore architectures, parallel SAT solving has become an active field of research. Parallel solvers are characterized by two main aspects: the solving strategy and the mechanisms of exchanging knowledge acquired throughout the solving process. In this chapter, we summarize existing approaches for these aspects and discuss which of them are (not) suitable for Peer-to-Peer Desktop Grids and why (not).

## 32.1 Parallel Solving Strategies

Since the first parallel SAT solver has been presented [BS96], numerous techniques for parallel SAT solving have been proposed. A major discriminator is the targeted parallel architecture, i.e., shared or distributed memory. The following discussion is limited to techniques suitable for distributed memory architectures and thus in principle for Desktop Grids. The main approaches that have been shown to be effective in the past can be roughly divided into two categories: *decomposition* and *portfolio* approaches.

### 32.1.1 Problem Decomposition

Decomposition approaches are based on partitioning the search space such that the resulting partitions can be processed in parallel. Three different techniques to perform this partitioning have been proposed in the literature: guiding path decomposition, scattering, and splitting by hashing.

**Guiding path decomposition** [ZBH96] defines non-overlapping regions of the search space by recording the list of variables to which a value has already been assigned. A *Guiding Path* (GP) essentially is a path from the root to an arbitrary node in the search tree and equivalent to the assignment stack introduced in Chapter 31. A GP encodes all assigned variables, their values, and whether they are set by decision (*open*) or due to implication (*closed*). Starting from a given GP one can easily construct a pair of derived GPs that encode disjoint parts of the search tree. The first step in constructing these *splits* is to select one or more open variables. While this can in principle be done in a random fashion, most parallel solvers use only a single open variable — namely the first one in the GP — as the *split variable*. The rationale behind this strategy is that splitting on shorter GPs has more potential to yield large subproblems as the remaining search space is larger than for longer GPs. As the split variable is always assigned by decision, the dynamic decomposition using GPs is closely related to branching heuristics. In fact the split variable has been selected by the variable ordering recently. Today, activity-based branching heuristics like

*Variable State Independent Decaying Sum* (VSIDS) [MMZ+01] are predominant. The idea is that variables with high activity are of particular importance for the solving process as they have increased potential of triggering many implications during unit propagation. By using decision variables as split variables decomposition is effectively driven by the same heuristic.

GP decomposition is tightly integrated with the CDCL solving process and goes hand-in-hand with the exploration of the search space. Hence, GP decomposition is an *exploratory decomposition* [GGKK03] technique. Workload balancing is accomplished by simply exchanging the comparatively small guiding paths between nodes. Another valuable property of this approach is that lemmas can be exchanged unrestrictedly between all parallel solving tasks. This is not true for the other approaches discussed below. These properties made guiding path decomposition the most frequently used decomposition technique for parallel SAT solving on distributed memory architectures [CW03, BWKW05, JLU05].

**Scattering** [HJN06] is a generalization of the guiding path technique, where each solver gets a set of guiding paths instead of a single one. The scattered formulae are created by a single modified DPLL solver and then farmed out for parallel processing to external black-box solvers.

Lemma exchange is limited to one-way exchange from a given formula to their descendants in the scattering tree. The inventors of scattering admit that it is currently an open question whether scattering outperforms decomposition based on guiding paths. Furthermore, scattering as proposed is inherently bound to the Master/Worker execution model and thus is of limited scalability.

**Splitting by hashing** [BHS09] is a straightforward decomposition method that adds *hashing clauses* to the original formula. A simple way of constructing hashing constraints is the addition of *parity constraints*

$$H_i := \sum_{x \in S} x \equiv i \pmod 2 \tag{32.1}$$

with $S$ being a subset of the variables of the formula. The resulting hashed formulae $F_1 = F \wedge H_1$ and $F_2 = F \wedge H_2$ are disjoint and cover the entire search space. By performing this decomposition $n$ times in a recursive fashion one can construct $2^n$ subproblems.

Two serious limitations of the approach are that decomposition has to be done statically, i.e., before the computation starts, which severely impacts efficiency as the work load is highly imbalanced, and lemma exchange is no longer possible as clauses learned by the emitting solver are not necessarily logical consequences of the clause set of the receiving solver.

### 32.1.2 Portfolios

Modern CDCL solvers are a combination of different algorithms, implementation techniques, and heuristics. In many cases the performance of a certain solver dominates all others, on a given problem class. This is due to the fact that solvers are often highly tuned for particular problem domains. One can take advantage of such differences by combining several algorithms into a *portfolio* [GS01, BSK03b], and running them in parallel[1]. Unfortunately, a major challenge with this approach is to find a set of algorithms exhibiting sufficiently diverse runtime distributions. While algorithm portfolios have been successfully applied for a small number of competitors [XHHLB08, HJS09b], it is yet unclear how to solve the scalability problem for massively parallel setups with hundreds or thousands of nodes. In contrast, randomization is in principal better suited to build large portfolios, although recent results [Hyv09] indicate that the scalability of this approach is limited as well since the variability of solve times for randomized portfolios is limited.

Portfolio based parallel solvers are most simple to implement as fault-tolerance is intrinsic to the approach: With all solvers working on the same problem, all but one solver can crash without compromising correctness. Supporting heterogeneous resource sets is more complicated for portfolios than for approaches based on decomposition as diversity from different solvers may be obliterated by diversity stemming from processor heterogeneity. As an example, consider the case where the best performing solver is executed on the least powerful processor, the second best solver on the second least powerful, and so on. The same argument is *not* true for dynamic search-space splitting as powerful processors will split off work from less powerful ones when they become idle.

Although communication is largely dominated by problem instance distribution and lemma exchange, portfolio approaches are less demanding than decomposition approaches in this regard since there is no need for load balancing and fault tolerance mechanisms. However, COHESION's task pool provides mechanisms to reduce the associated overheads (see Chapter 27).

## 32.2 Distributed Dynamic Learning

The dynamic learning process of sequential CDCL SAT solvers relies on accumulated knowledge continuously deduced during the solving process. Learning clauses is essential for pruning the search space and for proving the unsatisfiability of a formula[2].

In a distributed setting each solver maintains its own local clause database containing the clauses of the formula and the lemmas which have been deduced locally. Of course a lemma derived by one solver can be of value for another solver as well. Thus, it is crucial to exchange lemmas among the clause databases in order to exploit the full potential of this technique. Doing so establishes a *distributed dynamic learning* process. It is orthogonal to the parallelization of the backtracking search and specifically addresses the deduction part of modern SAT solving methods. As discussed above, unrestricted lemma exchange is not compatible with all solving strategies.

Exchanging all lemmas in an all-to-all fashion is not feasible as the total amount of deduced lemmas increases linearly with the number of processors. Thus, lemma exchange

1 The organizers of the SAT Competition 2009 introduced the concept of the *Virtual Best Solver*, which can be regarded as *a solver which would run all other solvers in parallel, bringing together all the solvers strengths* [Sat09].

2 CDCL solvers implicitly perform a resolution refutation.

must be selective. Existing approaches select lemmas according to local criteria like the length [BSK01] or activity [PKA$^+$06] of clauses. More recent approaches [HJS09b, HJS09a] are adaptive. They either try to keep the number of clauses exchanged between two solvers constant, exchange only high quality clauses, or both.

# 33 Architecture

Figure 33.1 shows the three-tier architecture of SATCIETY. It consists of

1. the solver backend that implements the **distributed SAT solver**. The solver uses exploratory decomposition based on COHESION's distributed task pool that is capable of handling volatility and random faults (see Part VII). Tasks are interchanged for load-balancing in a compressed bandwidth-friendly encoding. The solver implements distributed dynamic learning that adapts to bandwidth-utilization constraints and instance-specific lemma generation rates. To protect host systems from memory overload, SATCIETY uses a three-stage memory management approach that preserves completeness of the solver. Details of the solver are described in Chapter 34 and Chapter 35.

2. a J2EE **application server** constituting the middle-tier which is responsible for job management and instance provisioning. The latter includes SAT formula simplification through variable and clause elimination, transcoding to a space-efficient structure-preserving replacement of the commonly used DIMACS format [Dim93], and efficient instance provisioning over a P2P file sharing protocol. It is described in detail in

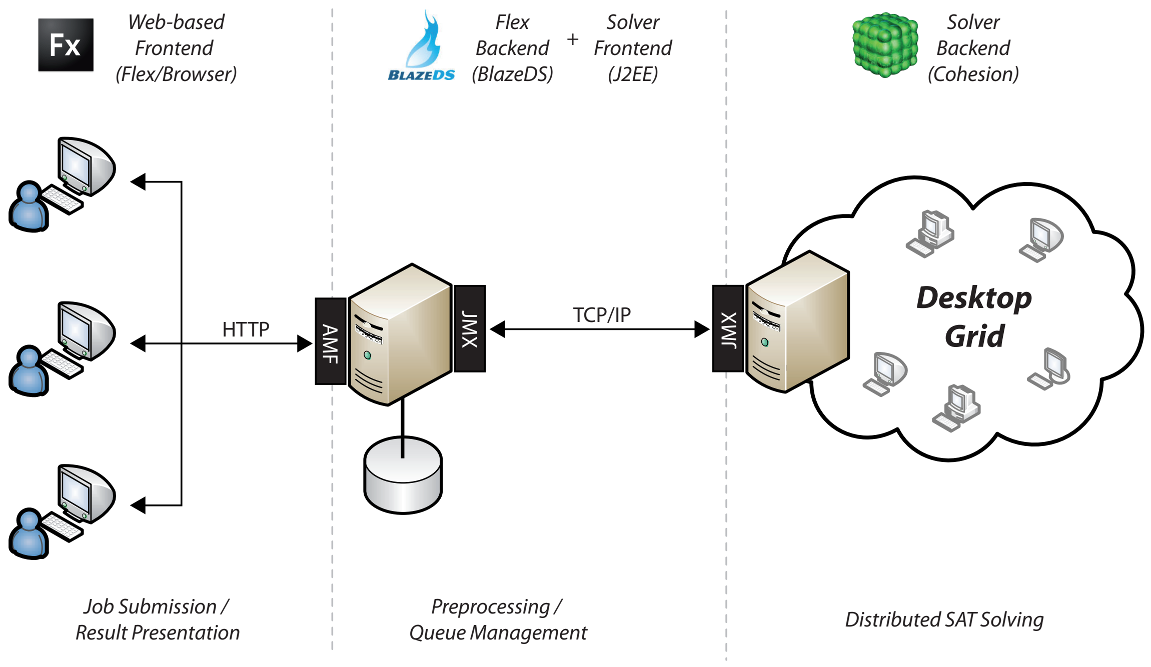
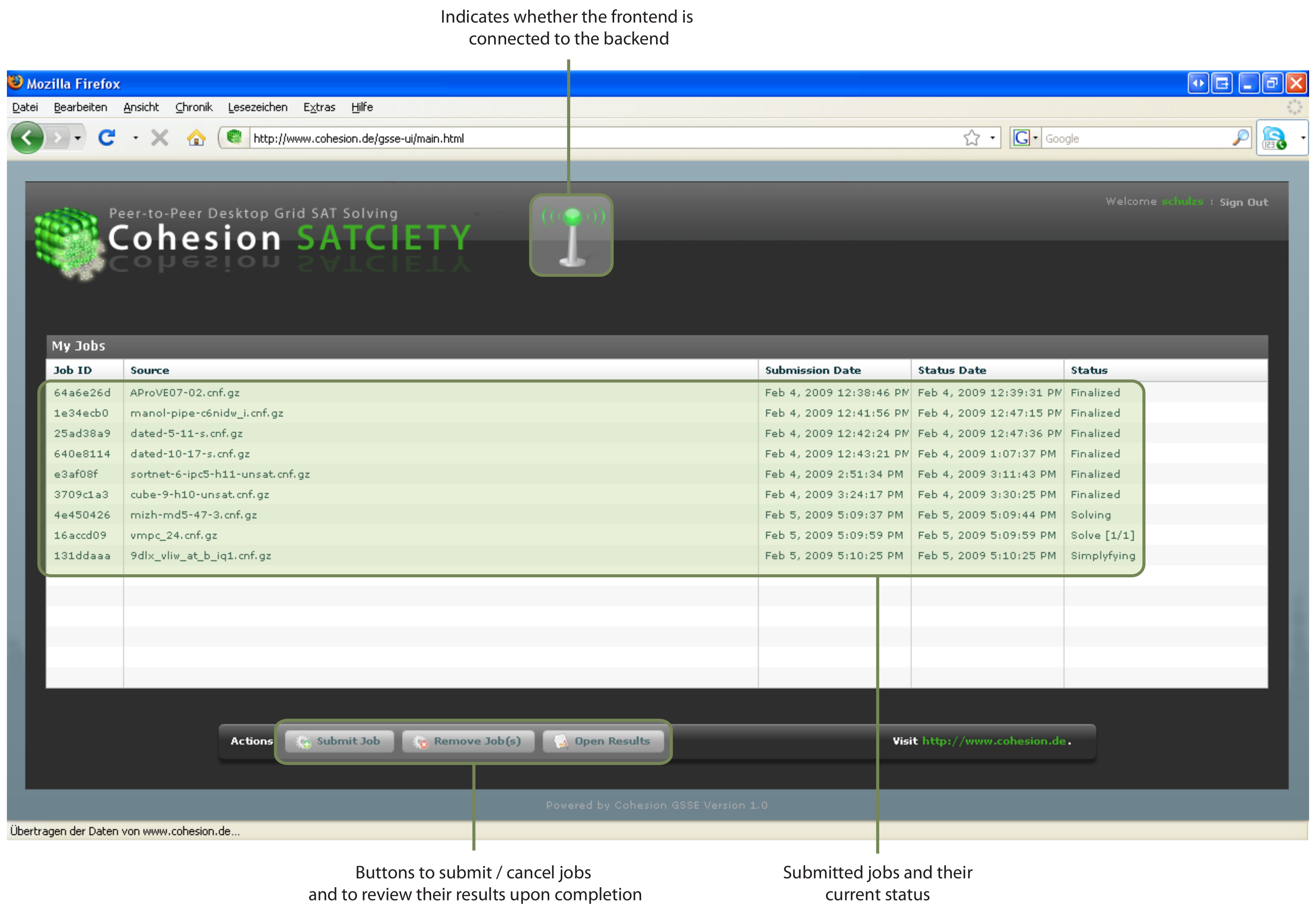

Fx
Web-based Frontend (Flex/Browser)
BLAZEDS
Flex Backend (BlazeDS)
+
Solver Frontend (J2EE)
Solver Backend (Cohesion)
HTTP
AMF
JMX
TCP/IP
JMX
Desktop Grid
Job Submission / Result Presentation
Preprocessing / Queue Management
Distributed SAT Solving

**Figure 33.1:** SATCIETY's three-tier architecture


Indicates whether the frontend is connected to the backend
Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://www.cohesion.de/gsse-ui/main.html
Google
Peer-to-Peer Desktop Grid SAT Solving
Cohesion SATCIETY
Welcome schulzs : Sign Out
My Jobs
Actions
Submit Job
Remove Job(s)
Open Results
Visit http://www.cohesion.de.
Powered by Cohesion GSSE Version 1.0
Übertragen der Daten von www.cohesion.de...
Buttons to submit / cancel jobs and to review their results upon completion
Submitted jobs and their current status

| Job ID | Source | Submission Date | Status Date | Status |
|---|---|---|---|---|
| 64a6e26d | AProVE07-02.cnf.gz | Feb 4, 2009 12:38:46 PM | Feb 4, 2009 12:39:31 PM | Finalized |
| 1e34ecb0 | manol-pipe-c6nidw_i.cnf.gz | Feb 4, 2009 12:41:56 PM | Feb 4, 2009 12:47:15 PM | Finalized |
| 25ad38a9 | dated-5-11-s.cnf.gz | Feb 4, 2009 12:42:24 PM | Feb 4, 2009 12:47:36 PM | Finalized |
| 640e8114 | dated-10-17-s.cnf.gz | Feb 4, 2009 12:43:21 PM | Feb 4, 2009 1:07:37 PM | Finalized |
| e3af08f | sortnet-6-ipc5-h11-unsat.cnf.gz | Feb 4, 2009 2:51:34 PM | Feb 4, 2009 3:11:43 PM | Finalized |
| 3709c1a3 | cube-9-h10-unsat.cnf.gz | Feb 4, 2009 3:24:17 PM | Feb 4, 2009 3:30:25 PM | Finalized |
| 4e450426 | mizh-md5-47-3.cnf.gz | Feb 5, 2009 5:09:37 PM | Feb 5, 2009 5:09:44 PM | Solving |
| 16accd09 | vmpc_24.cnf.gz | Feb 5, 2009 5:09:59 PM | Feb 5, 2009 5:09:59 PM | Solve [1/1] |
| 131ddaaa | 9dlx_vliw_at_b_iq1.cnf.gz | Feb 5, 2009 5:10:25 PM | Feb 5, 2009 5:10:25 PM | Simplyfying |

**Figure 33.2:** *Flex*-based graphical user interface of SATCIETY

Chapter 36. The middle-tier pushes job progress and result data in real-time to clients via *BlazeDS* [Adob]. The middle-tier is connected via *JMX* [KHW03] over TCP/IP to a dedicated COHESION peer in the solver backend called the *gateway*.

3. a **browser-based thin client**-tier based on *Adobe Flex* [Adoc] (Figure 33.2 shows a screenshot of the user interface) can be used by multiple users concurrently to submit, monitor, and control their SAT jobs. Clients communicate with the middle-tier over HTTP using the *Active Message Format* (AMF) [Adoa]. AMF is a binary format for *ActionScript* object serialization.

# 34 Task Pool Integration

In this chapter, we describe how Satciety is implemented on top of the task pool abstraction provided by Cohesion. In particular, we describe the structure of a Satciety task, how exploratory decomposition and checkpointing have been realized, how memory exhaustion is prevented, and the implementation of two optimizations – reuse of solver cores and task compression.

## 34.1 SAT Task Anatomy

A Satciety task consists of

1. the *gateway's peer identifier*. This information is used to deliver the result or any error conditions arising during the solving process.
2. a *reference to the input formula*, which is a *uniform resource locator* (URL) provided by the middle-tier as part of the provisioning process (see Section 36),
3. the set of *top-level assignments* $\phi_0$ which defines the subproblem to be solved. As large SAT instances have millions of variables this part constitutes the major fraction of a task. Thus, Satciety employs task compression as described below to minimize its size.
4. and additional *parameters* specified at job submission. This includes whether to suppress task decomposition in order to force the solver to operate in sequential mode, whether to perform lemma exchange, and a timeout to prevent the solver from running indefinitely long for very hard instances.

## 34.2 SAT Task Behavior

As depicted in Figure 34.1, a task's lifecycle consists of its creation on initial job submission or decomposition, possibly several migrations, execution and finally its completion. Tasks may get lost and eventually restored, or aborted as a consequence of user-initiated job cancellation or expiration of the timeout specified on job submission.

The workflow executed in the Running state is depicted in the lower part of Figure 34.1 and consists of acquiring a solver instance for the given formula, executing the solver on the arguments as specified by the task, optionally verifying the result, sending the result back to the gateway peer for delivery to the middle-tier of Satciety, and releasing the solver.

Several aspects (marked with an asterisk in Figure 34.1) of the default behavior of a Cohesion task have been customized. These are the operations `migrate`, `split`, and

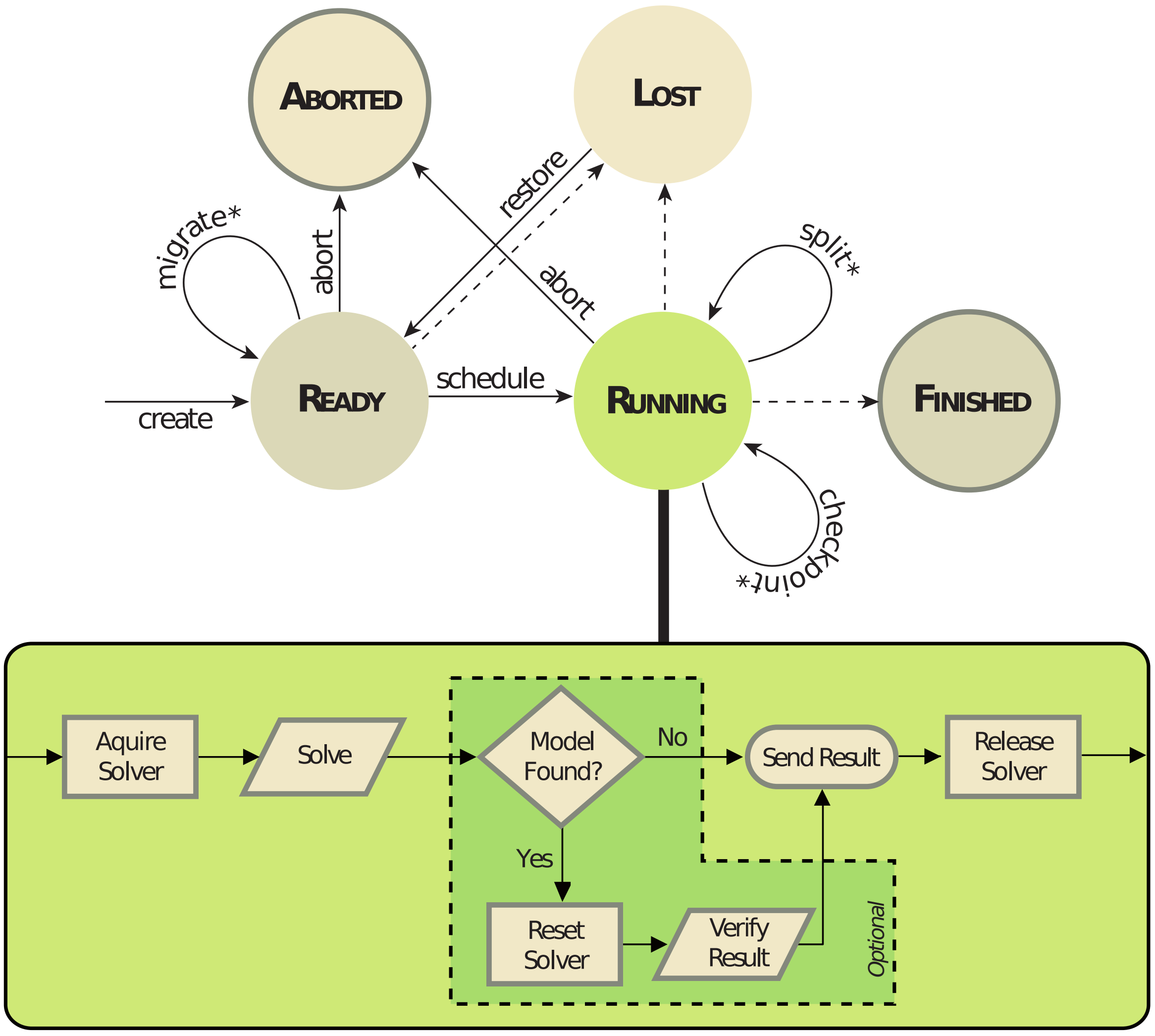


**Figure 34.1:** Lifecycle of a SATCIETY task

checkpoint. They are discussed right after a short description of solver recycling and the optional result verification step. Finally, task compression, which is performed as part of migrate operations, is explained.

### 34.2.1 Solver Reuse

Creating a new solver core instance for each task is disadvantageous for two reasons: First, parsing of formulae is a time-consuming task, especially for real-world problems that can be over a hundred megabyte in size [SATd]. Second, during processing of a task the solver accumulates valuable knowledge through dynamic learning that can be reused for pruning the search space when solving other tasks of the same job. For these reasons SATCIETY implements a solver pool from which solver instances are fetched as needed and to which solvers are released as soon as a task has been processed. To reclaim a released solver the pool performs a reset that basically rewinds the assignment stack.

### 34.2.2 Result Verification

SATCIETY can be configured to execute a verification step (the region surrounded by the dashed line in Figure 34.1) in case a solution is found. Although this step is functionally unnecessary, it was a valuable tool for identifying bugs in the highly complex distributed solver implementation[1]. To verify a supposed solution the previously created solver instance is reset and reused. In case the verification process fails, an error signal is sent to the gateway. Result verification in case no solution has been found is an open problem even for sequential SAT solving. Although, it is possible in principle, existing approaches are very inefficient and have not yet been generalized to work in the context of parallel SAT solving.

### 34.2.3 Task Splitting

The *split* operation is implemented using the guiding path decomposition technique described in Chapter 32. Technically, SATCIETY performs an assignment stack transformation (see Figure 34.2) that modifies the assignment stack of the original task and creates a new guiding path for the split off task. As motivated in Chapter 32, the variable assigned by decision on level 1 is selected for splitting. The assignment stack of the running solver is transformed on-the-fly, i.e., without restarting the solver, by moving the split literal and its implications to decision level 0. Hence, they become part of the top-level assignment and are fixed from then on. The top-level assignment for the split off task is constructed by appending the complement of the split literal to the top-level assignments of the original task.

As described in Chapter 26, COHESION supports different strategies for triggering task decomposition. We prefer on-demand splitting rather than eager problem decomposition. Thus, we can limit undesirable growth of the effective search space caused by unnecessary decomposition operations: As described in Section 31.1, a newly learned lemma prevents that the solver makes the same unprofitable work over and over again in other parts of the search space. By splitting tasks eagerly, the newly created task cannot profit from this extra knowledge if it is transferred to another node for execution. This effect can only be partially mitigated by distributed dynamic learning, as the vast number of locally deduced lemmas cannot be exported to remote clause databases efficiently. Consequently, part of the search space will be examined more often than necessary, resulting in the undesired growth of the effective search space.

### 34.2.4 Checkpointing

Checkpointing is implemented as a lightweight operation that copies the current top-level assignments from the solver core to the corresponding task object. As in the case of task splitting, a checkpoint can be created on the fly. While in principle the entire state of the solver core could have been included in a checkpoint, the sheer size of the clause database,

1 Achieving correctness is not easy even in the case of sequential SAT solving. This is reflected in the fact that several solvers returned wrong results in recent SAT competitions [satc].

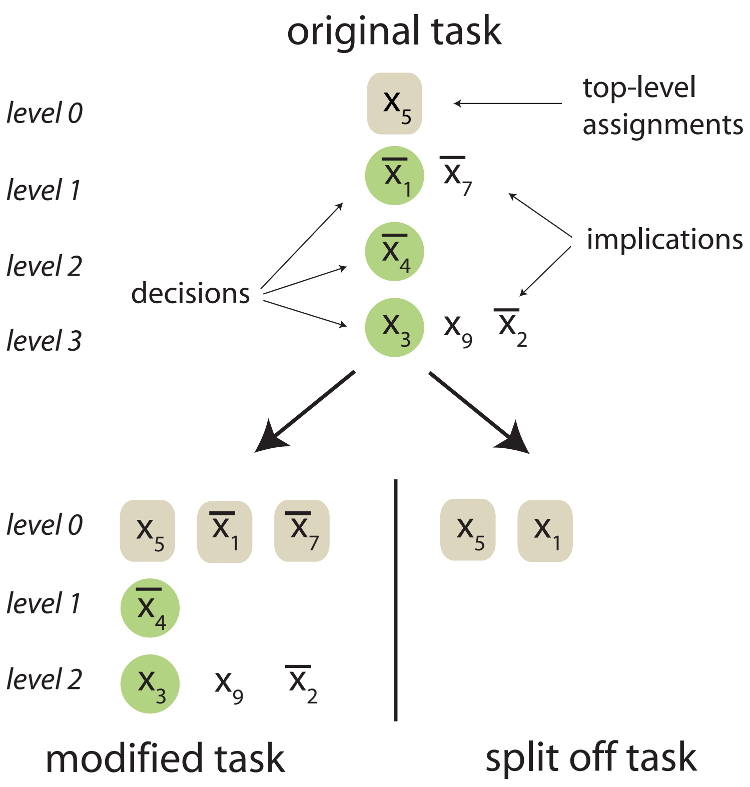


**Figure 34.2:** Assignment stack transformation for on-the-fly SAT task decomposition

which can contain millions of clauses, makes such an approach absolutely infeasible, especially in the context of Desktop Grids with comparatively low-bandwidth connections.

### 34.2.5 Differential Updates

COHESION's distributed task pool uses differential updates for task state and location bookkeeping to reduce bandwidth consumption on the superpeer. The conceptual description given in Chapter 27, is now complemented with specific information on how differential updates are implemented in the context of distributed SAT solving: The initial task, which has an empty set of top-level assignments, is submitted on the coordinator allowing TTCR-0 of Protocol 27.2 to be performed locally without external communication. When TTCR-1 (completed task), TTCR-3 (transfer from one peer to another), and TTCR-C-FPR-1 of Protocol 27.4 are executed, the task has been sent to the coordinator previously. Thus, it is sufficient to transfer a unique identifier attached to each task on creation that can be used by the coordinator to locate the associated task within the task location table. For the two remaining updates performed in TTCR-2, SATCIETY exploits the fact that the top-level assignments for both tasks resulting from a splitting operation can be composed from the top-level assignments of the original task, the assignments done up to the moment the split is performed, and the split literal (cp. Figure 34.2). As the coordinator already knows the top-level assignments of the original task, it is not necessary to include them in the `Update` messages. Together these measures significantly reduce the network load on the coordinator that would otherwise become a bottleneck.

### 34.2.6 Task Compression

As the solving process evolves, the top-level assignments quickly become the largest part of a task. Its size is in the order of the number of variables which can easily exceed a million for real-world instances. Simply using integer arrays to encode the top-level assignments results in tasks sizes in the order of megabytes. This is prohibitive in the context of Desktop Grid applications. Thus, SATCIETY employs a more sophisticated two-stage strategy: First, we look at a compact binary encoding of the top-level assignments. For the set of variables $V$, we need

$$|Lit|_2 = \lceil \log_2 |V| \rceil + 1 \tag{34.1}$$

bits to binary encode a literal and hence

$$|\phi_0|_{sparse} \approx |Lit|_2 \times |\phi_0| \tag{34.2}$$

bits to encode the top-level assignments by concatenating the encoded literals. We call this encoding *sparse* as only those variables are encoded that are assigned. In contrast a *dense* encoding is performed by encoding every literal between the lowest and the highest literal of the top-level assignments using 2 bits each. This results in a size of

$$|\phi_0|_{dense} \approx 2 \times (max\{i : lit_i \in \phi_0\} - min\{i : lit_i \in \phi_0\}). \tag{34.3}$$

When a task is to be migrated, SATCIETY's task encoder computes both $|\phi_0|_{sparse}$ and $|\phi_0|_{sparse}$ and encodes the top-level assignments using that encoding which yields the more compact representation. As a second step SATCIETY uses GZIP compression to further reduce the size of the task.

## 34.3 Memory Management

The CDCL algorithm adds a new clause to the clause database on every conflict. To avoid running out of memory, modern sequential SAT solvers use heuristics to predict the usefulness of clauses for the future solving process. Based on this prediction they decide which clauses are deleted on periodic database reductions. One popular heuristic is to delete those clauses that have not been used in conflict clause construction for a certain period of time. To guarantee termination CDCL solvers gradually increase the reduction interval. Hence, the solvers memory footprint is constantly growing, eventually resulting in the system running out of memory. In a Desktop Grid scenario this behavior is prohibitive for two reasons: First, with SATCIETY allocating a great deal of the overall system memory, other user processes are swapped out by the operating system, which drastically reduces responsiveness in case the user reclaims the system. Second, the JVM is killed when no more memory is available, preventing the host from further participation in the parallel solving process. To circumvent these undesirable scenarios, SATCIETY employs a three-stage memory management approach that enforces memory limits while preserving solver completeness. The stages are:

1. *Application Control*. SATCIETY leverages the fine-grained application lifecycle control of COHESION. SATCIETY is allowed to run only as long as the amount of free physical memory is above a given threshold $M_{Shutdown}$. In case free memory falls below this threshold, local tasks are off-loaded to other peers and the SATCIETY application is shutdown freeing all memory used by the solver.

2. *Stimulated Reduction*. When the amount of free physical memory drops below a threshold $M_{Stimulate} > M_{Shutdown}$ SATCIETY instructs the solver to reduce the clause database. This reduction is equivalent to reductions triggered by the clause removal heuristic described above and executed as part of the regular solving process.

3. *Forced Reduction*. Removing enough clauses to comply with memory constraints by stimulated reduction is not always possible, since clauses are locked when they are participating in the current backtracking branch by being the reason for a variable assignment [ES03]. In case free memory is below a threshold $M_{Forced}$ with

$$M_{Stimulate} > M_{Forced} > M_{Shutdown} \tag{34.4}$$

after a stimulated reduction has been performed, SATCIETY triggers a restart. This rewinds the assignment stack which unlocks additional clauses that may now be safely deleted by another stimulated reduction.

To guarantee termination SATCIETY splits off a new task after performing a stimulated or forced reduction. As described in Section 34.2 splitting fixes the decision variable on level 1 in the original task and adds the variable in the opposite phase to the assignment stack of the split off task. Thus, performing the split ensures progress and eventual termination.

# 35 Topology-Aware Distributed Dynamic Learning

Dynamic learning by conflict analysis has become standard for sequential SAT solvers and tremendously improves their performance. By exchanging lemmas between solver cores, a similar effect can be achieved for parallel and distributed SAT solvers. Although there is neither a thorough theoretical investigation nor there are dependable experimental results, the beneficial effect of sharing comparatively short lemmas is undisputed in the literature [BSK01, PKA$^+$06, HJS09b] as they carry high potential to help in pruning the search space while at the same time are cheap to exchange.

A feature specific to Desktop Grid environments is the fact that in general a peer is not able to directly exchange messages with all other peers. While COHESION removes this limitation by message relaying performed by superpeers, restricted connectivity introduces inhomogeneous communication costs that have to be reflected in application level communication patterns to avoid inefficiency and superpeer overload. This kind of protocols is called *topology-aware*. In contrast to existing Grid-enabled SAT solvers [CW06a, HJN08], lemma exchange in SATCIETY is topology-aware. Neighbor classification is performed by leveraging the component detection algorithm provided by ORBWEB (see Chapter 19). As depicted in Figure 35.1, lemmas are exchanged at a high rate within and at a lower rate between

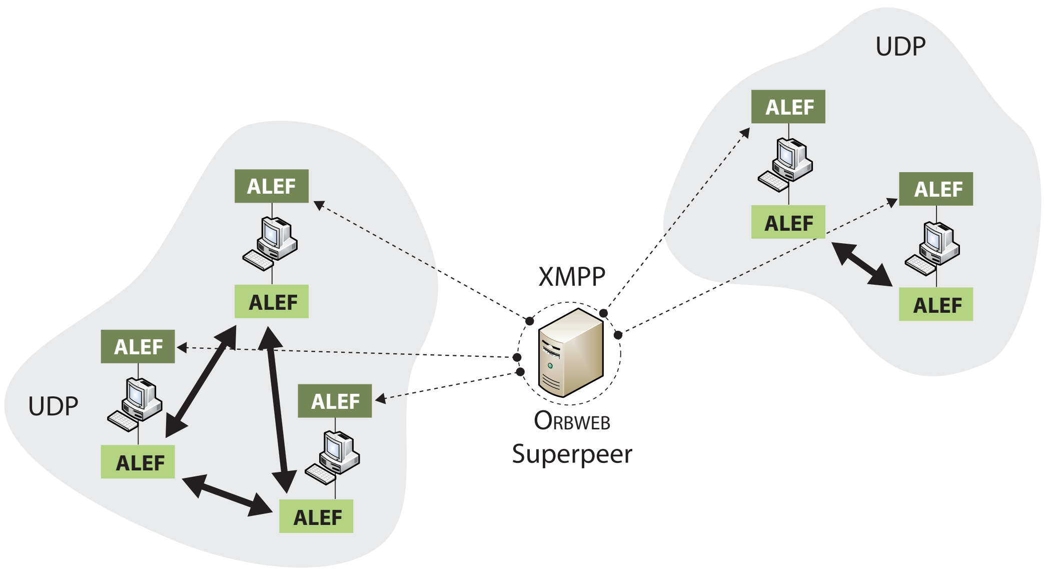


**Figure 35.1:** SATCIETY's topology-aware lemma exchange using two ALEFs (see Figure 35.2) for higher exchange rates within (light green) than between network components (dark green)

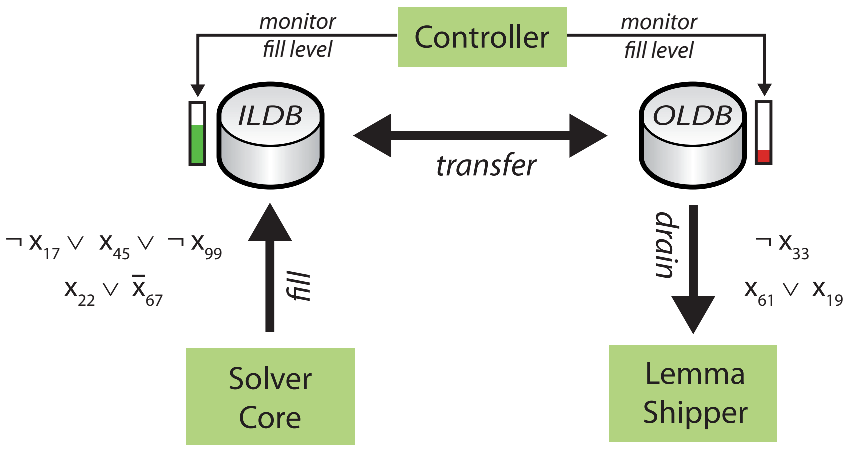


**Figure 35.2:** SATCIETY's ALEF implements a demand-driven strategy that maintains a steady lemma output rate by dynamically regulating the lemma output of the solver core.

components. While SATCIETY uses UDP for the former because of its small overhead, lemmas exchanged between peers within different components are transmitted in-band over XMPP connections with messages relayed by the ORBWEB superpeer.

The number and average size of deduced lemmas heavily depends on the concrete SAT instance. Straightforward solutions using hard-coded size limits thus yield unsatisfactory results. Hence, SATCIETY makes use of an adaptive approach to ensure that a predefined exchange rate is sustained but not exceeded. This is of particular importance in the Desktop Grid context, as the user experience may be impaired by excessive communication. Figure 35.2 illustrates the functionality of the *Adaptive Lemma Exchange Facility* (ALEF) which is instantiated twice by SATCIETY to implement topology-awareness as described above. The ALEF control logic as depicted in Figure 35.2 continuously monitors two lemma databases each with a fixed capacity. While the *inbound lemma database* (ILDB) is filled with lemmas produced by the solver core, the *outbound lemma database* (OLDB) is drained by the lemma shipper, which marshals extracted lemmas and sends them to the target peer. Every time the OLDB becomes empty or the ILDB becomes full, the controller swaps the content of the databases. Although the mechanism could have been implemented with a single DB filled and drained concurrently, the strategy using two databases decouples solver core and shipper, which is of particular importance as conflict clause generation is part of the main loop of CDCL solvers. The database swap does not involve copying of the lemmas but is implemented by simply swapping references. If both databases are constantly filled near their capacity limit, the lemma production rate is too high. In this case the controller decreases the length limit up to which lemmas are exported to the ILDB. Analogously, in case the databases are nearly empty for a certain period of time, the length threshold is increased. As proposed in [HJS09a], adjustment of the length threshold is done using the *additive increase/multiplicative-decrease* (AIMD) algorithm known from TCP congestion avoidance.

# 36 Instance Provisioning

SAT instances that encode real world problems can be very large, this is especially true for instances originating from formal verification. Figure 36.1 shows the histogram of file sizes of the DIMACS [Dim93] encoded instances of the *SAT Competition 2007* [Sata]. Depending on how they are created, instances are categorized as random, (hand-)crafted, or industrial. Most instances from the industrial category are larger than 512 KB with a cluster between 512 KB and 4 MB ranging up to over 40 MB. The size of instances from the *SAT Race 2008* [Satb] is even up to 145 MB. Although there is in general no correlation between the size of a formula and the time required to solve it, in practice, one can often observe such a relation in sequential SAT solving – at least for problems with similar structure. Hence, reducing the size of formulae by simplification is a promising approach to shorten solving times and thus is a subject of active research. In the Desktop Grid scenario, limited bandwidth is another important reason to keep instance encodings as compact as possible. For the largest formula from above (47 MB) and the grid comprised of 40 peers used in our performance evaluation in Section 37), a total of $40 \cdot 47$ MB $\approx 2$ GB of data has to be transmitted only to transfer the formula to the peers.

For these reasons SATCIETY employs an extensible preprocessing pipeline to ensure that formulae are compactified as much as possible before the actual distributed solving process is started. Extensibility ensures that new techniques can be incorporated easily. The pipeline

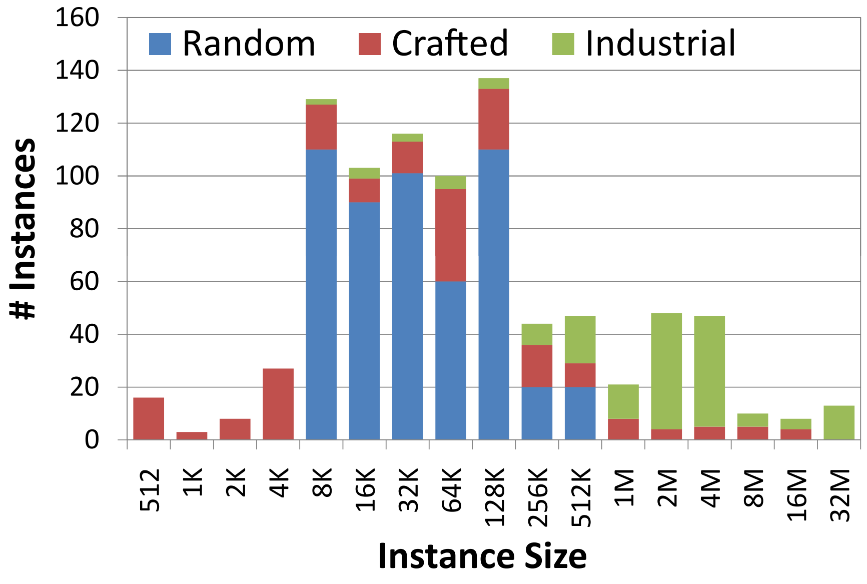


**Figure 36.1:** Histogram of DIMACS encoded file sizes for 877 instances of the SAT Competition 2007

is located in the middle-tier of Satciety (see Chapter 33). By default, it consists of five stages: decompression, simplification, transcoding, compression and P2P provisioning. Each stage maintains a job queue to allow for *pipelining*, i.e., the stages can be executed in parallel for different instances. The decompression and compression stages are quite self-explanatory: They use the GZIP algorithm to decompress the incoming formula and to compress the transcoded formula respectively. The remaining three stages are explained subsequently.

## 36.1 Formula Simplification

Preprocessing of SAT instances is a field of active research. Numerous approaches have been proposed that use various simplification techniques including hyper-resolution [BW03], clause distribution [SP04], vivification [PHS08], blocked clause elimination [JBH10], or a combination of several techniques [AS06]. How they actually work is not relevant in the context of this work. The application of a preprocessor to a SAT formula typically results in a reduction of the number of literals, variables, and clauses, which yields smaller file sizes and often significantly reduces the runtime of the SAT solver. Satciety uses *SatELite* by Eén and Biere [EB05] in the simplification stage of the preprocessing pipeline. As substantiated by the results of our evaluation presented in Chapter 37, *SatELite* is a lightweight preprocessor and does not add significant overhead to the solving process, which is crucial as preprocessing contributes to the sequential fraction of the solving process and thus limits the achievable speedup according to Amdahl's Law (see Appendix A). Thus, parallel simplification techniques will be required to exploit the full potential of parallel SAT solving. However, the invention of such techniques is beyond the scope of this thesis.

```
 c SAT07-Contest Parameters:
 c unif2p p=9 nbc2=228 nbc3=2052
 c v=630 seed=702278147
 p cnf 630 2280
-464 -204 0
-134 384 0
-456 446 0
 ...
-39 45 -225 0
 295 516 337 0
```

**Figure 36.2:** An example formula in DIMACS format from the random category of the *SAT Competition 2007* consisting of a preamble (comments and problem line giving the number of clauses and variables) and the clauses in CNF. Negative literals are denoted with a minus character. Clauses are terminated by a '0'.

## 36.2 Transcoding to Binary CNF

SAT problems are usually encoded in the DIMACS format [Dim93]. DIMACS files are plain ASCII files structured as depicted in Figure 36.2. While textual data encodings do not suffer from differences in endianness and thus are in principle well suited as a file format, their space efficiency is low. For example a literal -3.456.789 produces an eight Byte ASCII text encoding, while a binary representation consumes only 23 Bit or 3 Byte if byte alignment is enforced. Thus, SATCIETY transcodes DIMACS encoded formulae to a structure preserving[1] binary encoding we call *Bit-Packed Binary CNF* (BCNF). The BCNF encoding is illustrated in Figure 36.3.

SAT instances are very different with respect to the number of variables and the length of clauses: While a hard random instance may have only a couple of variables and rather long clauses, formulae resulting from real-world problems often have millions of variables and many very short clauses. Thus, our BCNF encoder first looks at the number of variables (given in the DIMACS preamble) to compute the number of bits $|Lit|_{BCNF}$ necessary to encode a single literal, which is

$$|Lit|_{BCNF} = \lceil \log_2 |V| \rceil + 1, \tag{36.1}$$

where $V$ is the set of variables. While clauses are terminated by a '0' in DIMACS, BCNF writes a variable length header in front of each clause indicating the number of literals the clause consists of. This way memory can be allocated before reading the clause which helps avoiding unnecessary copying. The first two bits of the header are used to indicate the length of the header resulting in a possible maximum clause length of $2^{8n-2}$ for an n-byte header. BCNF could have been designed to be even denser, if structural modifications like variable or clause reordering would have been applied. However, SATCIETY makes no use of such optimizations for two reasons: First, downstream GZIP compression would have partially leveled out potential savings. Second, reordering heavily influences the runtime of SAT instances making performance comparisons unnecessarily difficult.

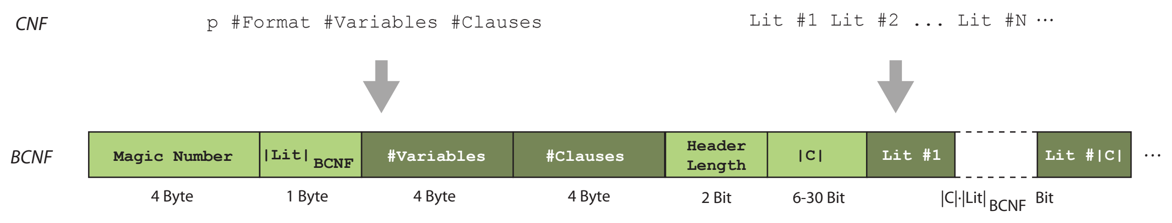


**Figure 36.3:** The transcoding of a generic DIMACS encoded formula into SATCIETY's Bit-Packed Binary CNF format

1 The *structure* of a formula is the order of the clauses within the file and the order of the literals within the clauses.

## 36.3 Peer-to-Peer Provisioning

After simplification and transcoding to BCNF the formula has to be delivered to the peers of the Desktop Grid. As illustrated by the estimation above, this involves the transmission of huge amounts of data for large real-world instances. Even when SATCIETY's preprocessing efforts result in files half as large as the original, the application server in the middle-tier of SATCIETY still would be busy for minutes serving the formula, which actually stalls the whole computation as all peers were served concurrently throttling each and every transfer. This phenomenon is commonly known as a *flash crowd* [AHM+03]: a resource catches the attention of a large number of parties, and subsequently gets an overloading surge of traffic.

To circumvent this problem, SATCIETY leverages *BitTorrent* [bit] for distributing SAT formulae. *BitTorrent* is a peer-to-peer file sharing protocol which distributes the onus of uploading over all participants. A large body of research concerning all aspects of the protocol has been conducted. For a thorough analysis of its performance in heterogeneous systems – like Desktop Grids – see for example Liao *et al.* [LPP07]. As demonstrated in Chapter 37 using *BitTorrent* significantly reduces provisioning time for large instances and/or large Grids.

# 37 Performance Evaluation

In this chapter we present the results of a performance analysis of SATCIETY. The testbed is the same as that used to evaluate COHESION's distributed task pool in Chapter 30.

## 37.1 Peer-to-Peer Provisioning

As discussed above, SAT instances can be very large. Distributing the file from a single server inevitably becomes a limiting factor for achieving high efficiency when the system is scaled beyond a few dozen peers. To substantiate this claim, we performed a comparison between provisioning over HTTP by the application server on the middle-tier of SATCIETY and P2P provisioning over *BitTorrent*. Our implementation is based on the *libtorrent* v0.14.9 library from *Rasterbar Software* [Ras] attached over JNI. Evaluation runs were repeated ten times and the confidence intervals are based on a 95 % confidence level.

Figure 37.1 shows the time until a formula is available on all peers in a 40 peer setup for varying formula sizes and both protocols. For formulae up to a size of 4 MB the overhead

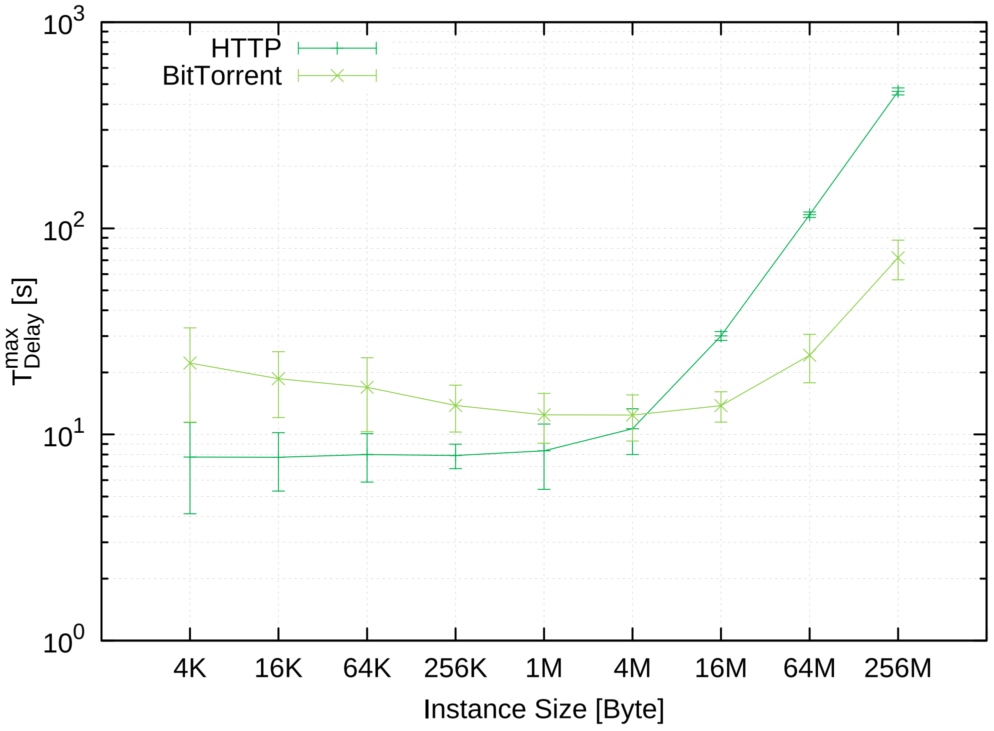


**Figure 37.1:** Comparison of elapsed time $T_{Delay}^{max}$ until a SAT formula of given size is available on all peers

of scraping[1] and peer-to-peer connection negotiation levels out the achievable speedup of multi-sourced download resulting in relative differences between $\approx 185\%$ for 4 KB and $\approx 16\%$ for 4 MB instances in favor of server-based provisioning. For 16 MB formulae and beyond P2P provisioning clearly outperforms server-based provisioning by increasing factors up to $\approx 6.4$ for 256 MB instances. The absolute difference for the latter amounts to 389 s, which is significant when compared to the overall runtimes given in the following section. The overall amount of data transmitted in this setup is 10 GB.

As the absolute difference between the two mechanisms is small for instance sizes up to 4 MB, the potential of an adaptive strategy, that employs the best mechanism for a given instance size, is limited. Furthermore, the transition point where P2P provisioning first outperforms server-based provisioning would shift towards lower instance sizes for larger networks. Thus, SATCIETY always uses P2P provisioning regardless of instance size.

## 37.2 SAT Solving

SATCIETY provides an interface to plugin any CDCL solver that provides the ability to split of new tasks on demand (as described in Chapter 34) and that emits and is able to consume lemmas on the fly. For our performance analysis, we used *Minisat* v1.14 [ES03] attached over JNI. *Minisat* attained top rankings in SAT competitive events of the last years [satc].

Motivated by the scalability results for COHESION's distributed task pool (see Chapter 30), the benchmark suite used to evaluate SATCIETY's SAT solving performance consists of long-running benchmarks from all three categories of the *SAT Competition 2007*. We call an instance *long-running* when it has been solved by *Minisat* within the allowed time (10,000 s for industrial and 5,000 s for crafted and random instances) but took at least 1,200 s. This criterion is met by 15 random, 23 crafted, and 17 industrial instances.

Note that we restrict our evaluation to unsatisfiable instances. For parallel heuristic search methods like SAT solving, speedups obtained for satisfiable instances are often super-linear (see Appendix A). Hence, performance results obtained for satisfiable instances are not suitable to investigate on the effectiveness of a specific parallel method.

Sequential runtimes were measured on the fastest machine among the testbed hosts (Type IV in Table 9.1). On each host a single COHESION peer was deployed. The coordinator peer performing fault-tolerance and termination detection was also located on the fastest host for the parallel runs. All other peers were configured to use the real-world session time distribution described in Chapter 30 and an error probability of $P_{Error} = 1\%$. Although recent research [Bie08] indicates that using (at least rapid) restarts may not be beneficial for unsatisfiable instances, disabling them would bias the results towards unsatisfiable instances. We thus use restarts for both the sequential and the parallel setups.

The limits for SATCIETY's memory management strategy (see Section 34.3) were set to $M_{Stimulate} = 90$ %, $M_{Forced} = 95$ %, and $M_{Shutdown} = 98$ % of the total physical system memory and/or the maximum supported process memory. Evaluation runs were repeated three times. The presented confidence intervals are based on a 95 % confidence level.

1 *Scraping* is the process of generating file chunks and associated metadata for distribution over *BitTorrent*.

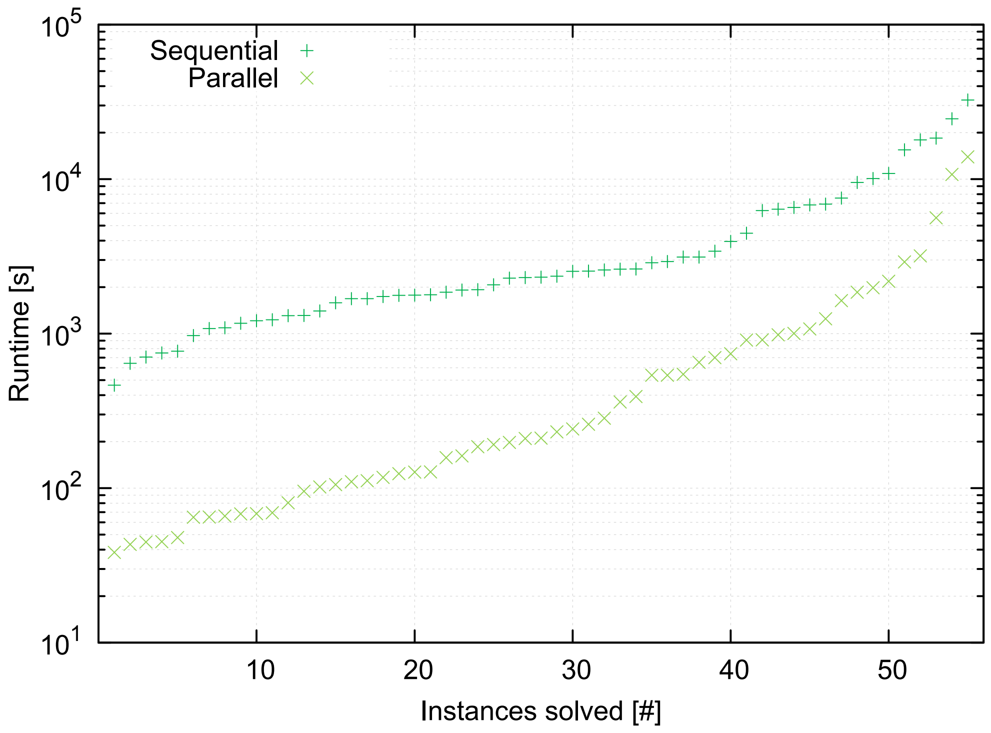


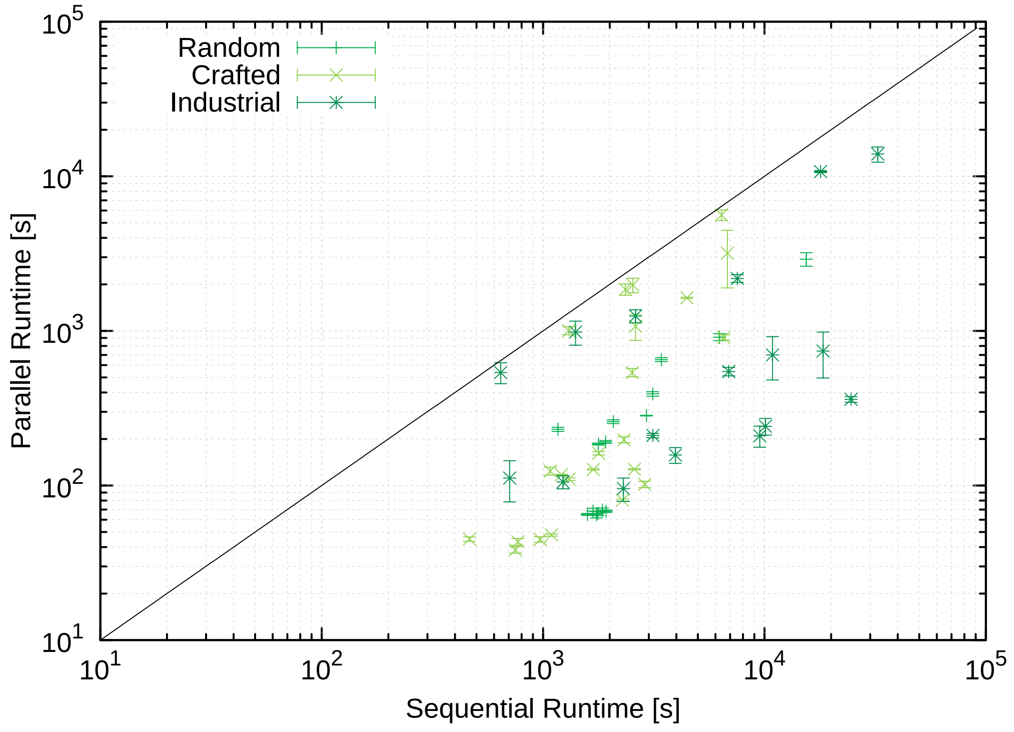
**Figure 37.2:** Cactus Plot[1] comparing the performance of the sequential *Minisat* and SATCIETY

Table 37.1 shows the results of our evaluation. With the exception of four benchmarks, preprocessing through *SatELite* and transcoding to BCNF resulted in significant file size reduction of up to 81.4 % and 30.3 % on average. Preprocessing times ($T_{pre}$) are – with a single exception (`uts-l06-ipc5-h33-unknown`) – negligible as compared to both sequential ($T_{seq}$) and parallel ($T_{par}$) solver runtimes. Thus, they do not significantly affect speedups.

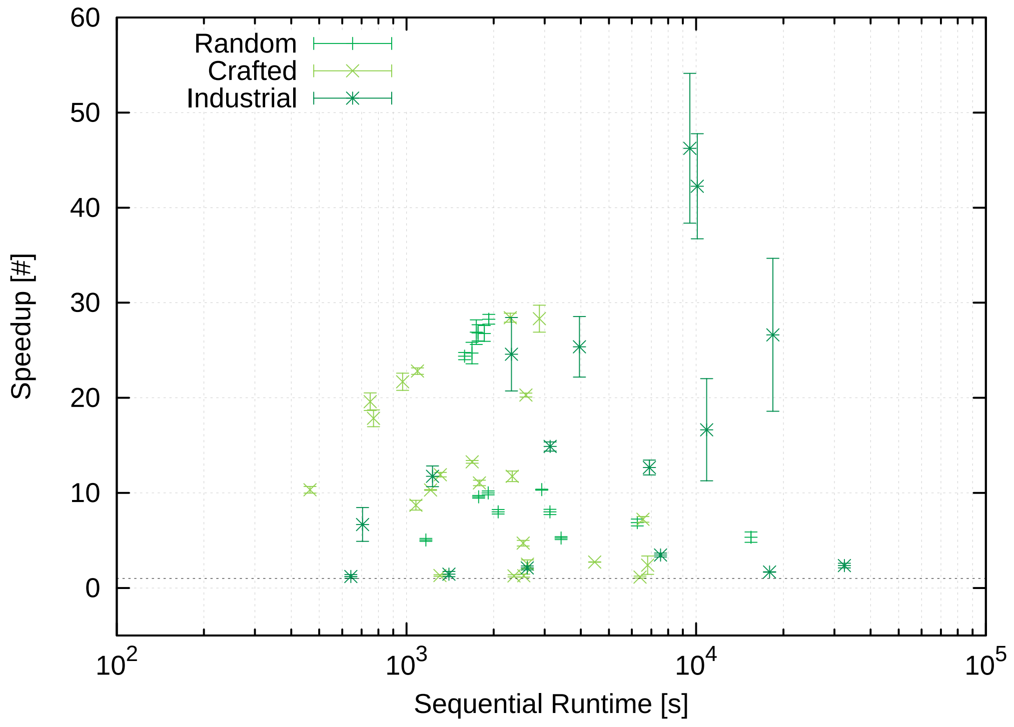
As can be seen from Figure 37.2 and Figure 37.3a, SATCIETY clearly outperforms the sequential solver in all categories realizing significant average speedups of $15.1 \pm 0.5$ for random, $11.3 \pm 0.4$ for crafted, and $18.1 \pm 2.5$ for industrial instances (see Figure 37.3b). The average speedup over all categories is $14.5 \pm 1.1$. Note that the given speedup values are based on the sequential runtimes on the fastest machine with a RAPI (as introduced in Chapter 22 on page 235) value of 1.0 (cf. Table 9.1) while the mean RAPI value of the hosts in our testbed is 0.44. In this light, the performance of SATCIETY is even more pronounced.

The variability of speedups, both across instances and across runs for the same instance, however is very pronounced. This decreased robustness is caused by *work-anomalies* [Blo06], i.e., the total amount of work carried out by the parallel execution is different and in some cases considerably larger than for the sequential execution. This behavior shows that it is difficult to keep the effectiveness of the sequential heuristics in the corresponding parallel

1 *Cactus Plots* are traditionally used in SAT competitive events to compare SAT solver performance. It is a cumulative plot showing how many instances (x-axis) have been solved in time below or equal to a given runtime (y-axis).



**(a)** Head-to-head solver comparison with respect to runtime



**(b)** Speedup by sequential runtime

**Figure 37.3:** Performance comparison of sequential *Minisat* and SATCIETY

version at the same level for a wide range of different SAT instances. Work anomalies tend to become more pronounced for larger number of processors which represents a significant challenge for future research. Possible countermeasures are hybrid solving strategies that combine both search space splitting and portfolio parallelism concurrently. Schulz *et al.* have recently published first results on such a hybrid solver based on SATCIETY [SB10a].

**Table 37.1:** SATCIETY performance for long-running problems from the *SAT Competition 2007* in a volatile Desktop Grid ($L_I \mathrel{\hat{=}}$ initial instance size, $L_F \mathrel{\hat{=}}$ final instance size after preprocessing, $\delta_L = 1 - (L_F/L_I) \mathrel{\hat{=}}$ instance size reduction, $T_{pre} \mathrel{\hat{=}}$ time spent on preprocessing, $T_{seq} \mathrel{\hat{=}}$ runtime of the sequential solver, $T_{par} \mathrel{\hat{=}}$ runtime of the SATCIETY solver). The index of dispersion is the (sample) standard deviation.

| **Instance** | **Preprocessing** | | | | **Solving Runtime** | | **Speedup** |
|---|---|---|---|---|---|---|---|
| | $L_I$ [Byte] | $L_F$ [Byte] | $\delta_L$ [%] | $T_{pre}$ [s] | $T_{seq}$ [s] | $T_{par}$ [s] | |
| **Random Category** | | | | | | | |
| unif2p-p0.7-v4500-c12015-S1349608788-18 | 78892 | 59607 | 24 | 0.74 | 15454.42 | 2912.38±295.22 | 5.34±0.54 |
| unif2p-p0.8-v1665-c5178-S883553387-12 | 32368 | 25425 | 21 | 0.46 | 3418.74 | 652.04±19.11 | 5.24±0.15 |
| unif2p-p0.8-v2035-c6328-S1022605561-16 | 40127 | 30140 | 25 | 0.50 | 1775.25 | 184.80±2.74 | 9.58±0.14 |
| unif2p-p0.9-v630-c2280-S1071799860-07 | 13560 | 11103 | 18 | 0.35 | 1166.76 | 230.75±6.90 | 5.05±0.15 |
| unif2p-p0.9-v810-c2932-S1275186626-09 | 17766 | 14178 | 20 | 0.37 | 6263.49 | 911.66±48.60 | 6.88±0.36 |
| unif-k3-r4.26-v400-c1704-S1013535775-14 | 10007 | 7777 | 22 | 0.33 | 3127.13 | 391.11±13.09 | 8.00±0.27 |
| unif-k3-r4.26-v400-c1704-S105499989-20 | 9982 | 7784 | 22 | 0.33 | 1914.97 | 191.20±3.89 | 10.00±0.20 |
| unif-k3-r4.26-v400-c1704-S1671397883-11 | 10050 | 7800 | 22 | 0.33 | 2073.05 | 258.62±7.33 | 8.01±0.23 |
| unif-k3-r4.26-v400-c1704-S1925680230-06 | 10013 | 7746 | 23 | 0.33 | 2928.86 | 282.92±1.40 | 10.34±0.05 |
| unif-k5-r21.3-v90-c1917-S1380126410-13 | 14035 | 9930 | 29 | 0.37 | 1740.44 | 64.42±3.17 | 26.91±1.29 |
| unif-k5-r21.3-v90-c1917-S1412274662-11 | 14042 | 9938 | 29 | 0.37 | 1585.32 | 64.67±0.99 | 24.39±0.37 |
| unif-k5-r21.3-v90-c1917-S1414833579-15 | 13988 | 9925 | 29 | 0.37 | 1682.77 | 67.85±3.03 | 24.71±1.12 |
| unif-k7-r89-v55-c4895-S1155123565-17 | 42501 | 34066 | 20 | 0.45 | 1923.19 | 67.62±1.23 | 28.26±0.51 |
| unif-k7-r89-v55-c4895-S1215610276-02 | 42451 | 34125 | 20 | 0.44 | 1766.97 | 65.48±2.13 | 26.83±0.85 |
| unif-k7-r89-v55-c4895-S145251364-05 | 42466 | 34120 | 20 | 0.44 | 1854.85 | 68.91±2.13 | 26.77±0.83 |
| **Crafted Category** | | | | | | | |
| 999999000001nw.sat05-447 | 130996 | 70639 | 46 | 0.95 | 748.70 | 37.38±1.83 | 19.59±0.92 |
| connm-ue-csp-sat-n800-d0.02-s925928766.sat05-538 | 48848 | 46435 | 5 | 0.73 | 1077.33 | 123.32±7.24 | 8.71±0.52 |
| contest03-hwb-n26-01-S1957858365.sat05-500 | 4713 | 3088 | 34 | 0.27 | 464.08 | 44.71±1.49 | 10.33±0.35 |
| contest03-SGI_30_50_30_20_1-dir.sat05-439 | 274484 | 312259 | -14 | 6.88 | 1679.13 | 120.21±1.37 | 13.27±0.14 |
| contest03-SGI_30_50_30_20_3-dir.sat05-440 | 279307 | 311213 | -11 | 4.71 | 2872.18 | 97.03±4.95 | 28.32±1.42 |
| contest04-lksat-n1000-m6860-k4-l4-s1935114289 | 52960 | 39473 | 25 | 0.46 | 2319.05 | 197.33±9.12 | 11.74±0.56 |
| hwb-n26-03-S540351185.sat05-490 | 4727 | 3109 | 34 | 0.27 | 769.86 | 42.97±2.19 | 17.84±0.89 |
| hwb-n28-01-S136611085.sat05-491 | 5127 | 4112 | 20 | 0.28 | 1309.34 | 109.67±2.16 | 11.91±0.23 |
| hwb-n28-02-S818962541.sat05-492 | 5088 | 4095 | 20 | 0.28 | 2584.33 | 127.11±1.23 | 20.29±0.20 |
| linvrinv5.sat05-564 | 7502 | 6120 | 18 | 0.32 | 1090.34 | 47.50±0.72 | 22.81±0.34 |
| mod2c-3cage-10-2.sat05-2567 | 3624 | 2641 | 27 | 0.23 | 2527.79 | 538.19±33.68 | 4.71±0.31 |

Table 37.1 – Continued

| **Instance** | **Preprocessing** $L_I$ [Byte] | $L_F$ [Byte] | $\delta_L$ [%] | $T_{pre}$ [s] | **Solving Runtime** $T_{seq}$ [s] | $T_{par}$ [s] | **Speedup** |
|---|---|---|---|---|---|---|---|
| mod2c-3cage-10-3.sat05-2568 | 3644 | 2620 | 28 | 0.22 | 2616.97 | 1073.21±204.98 | 2.50±0.46 |
| phnf-size10-exclusive-luckySeven.used-as.sat04-990 | 6558136 | 6468952 | 1 | 50.47 | 6751.40 | 3134.40±1289.16 | 2.39±0.97 |
| pmg-12.sat05-3940 | 3370 | 2709 | 20 | 0.23 | 2282.49 | 80.11±1.32 | 28.42±0.47 |
| pyhala-braun-40-4-02.sat05-459 | 211234 | 71182 | 66 | 0.89 | 969.55 | 43.90±1.85 | 21.69±0.90 |
| QG7a-gensys-icl001.sat05-3822 | 228575 | 168157 | 26 | 1.00 | 6395.19 | 5619.02±443.29 | 1.14±0.09 |
| QG7-dead-dnd001.sat05-3419 | 102470 | 52390 | 49 | 0.58 | 1304.73 | 1002.32±58.88 | 1.30±0.07 |
| QG7-dead-dnd002.sat05-3108 | 141168 | 64408 | 54 | 0.72 | 6550.16 | 908.05±35.96 | 7.22±0.29 |
| QG7-gensys-icl100.sat05-3226 | 162219 | 100981 | 38 | 0.78 | 2539.24 | 1978.89±213.54 | 1.29±0.15 |
| QG7-gensys-ukn003.sat05-3346 | 135088 | 84653 | 37 | 0.69 | 4464.59 | 1635.71±15.29 | 2.73±0.03 |
| s101-100 | 812 | 562 | 31 | 0.17 | 1784.41 | 161.49±4.14 | 11.04±0.28 |
| s97-100 | 777 | 546 | 30 | 0.17 | 1210.75 | 117.36±0.43 | 10.30±0.04 |
| unsat-set-b-fclqcolor-10-07-09.sat05-1282 | 19664 | 21806 | -11 | 0.46 | 2350.76 | 1854.89±157.90 | 1.27±0.11 |
| **Industrial Category** | | | | | | | |
| AProVE07-08 | 68759 | 66808 | 3 | 0.90 | 1228.54 | 104.46±10.29 | 11.74±1.09 |
| AProVE07-09 | 696516 | 693378 | 0 | 14.30 | 7522.14 | 2169.99±127.73 | 3.46±0.21 |
| AProVE07-16 | 814813 | 378435 | 54 | 6.05 | 699.84 | 105.56±33.29 | 6.68±1.78 |
| AProVE07-27 | 118729 | 100651 | 15 | 1.90 | 24591.93 | 358.88±15.61 | 68.25±2.98 |
| cube-11-h13 | 6238423 | 5259684 | 16 | 318.06 | 17609.81 | 10396.53±156.03 | 1.67±0.02 |
| dated-10-11-u | 2806193 | 637494 | 77 | 14.49 | 10857.07 | 685.39±218.52 | 16.63±5.37 |
| dated-10-13-u | 3672073 | 871186 | 76 | 20.06 | 9496.21 | 189.41±32.51 | 46.25±7.88 |
| dated-5-15-u | 3148506 | 635904 | 80 | 16.04 | 3939.23 | 141.47±18.50 | 25.36±3.19 |
| dated-5-17-u | 3902323 | 725932 | 81 | 20.17 | 10072.31 | 221.30±29.78 | 42.25±5.53 |
| emptyroom-4-h21 | 949908 | 748866 | 21 | 11.44 | 32460.37 | 13904.39±1571.51 | 2.35±0.25 |
| eq.atree.braun.11 | 26538 | 20337 | 23 | 0.47 | 2302.96 | 94.93±16.38 | 24.59±3.86 |
| manol-pipe-f9b | 2495048 | 1069250 | 57 | 19.06 | 2595.25 | 1228.82±120.50 | 2.11±0.21 |
| manol-pipe-f9n | 2517249 | 1083411 | 57 | 19.36 | 1382.13 | 962.74±174.24 | 1.46±0.28 |
| manol-pipe-g10nid | 2995202 | 1254304 | 58 | 21.80 | 18389.17 | 718.16±243.59 | 26.62±8.04 |
| sortnet-6-ipc5-h11 | 405533 | 174448 | 57 | 8.91 | 3121.68 | 201.56±6.94 | 14.88±0.48 |
| total-10-13-u | 4661995 | 1144413 | 75 | 25.70 | 6872.45 | 520.27±34.48 | 12.67±0.79 |
| uts-l06-ipc5-h33-unknown | 3496371 | 3625799 | -4 | 209.05 | 433.79 | 329.46±83.39 | 1.21±0.19 |

# 38 Related Work

In the following discussion of related work, we concentrate on approaches to parallel SAT solving that are designed for distributed-memory parallel architectures. We further distinguish between solvers suitable for clusters and solvers that can operate within a Grid. The reader interested in parallel SAT solving on shared-memory parallel hardware is referred to [FDH04], [SV05], [LSB07], and [HJS09b].

## 38.1 Parallel SAT Solving on Distributed-Memory Architectures

One of the first parallel SAT solvers was presented by Böhm *et al.* [BS96]. Their work especially investigates on efficient load balancing techniques for a d-dimensional mesh network-topology of a transputer. The outstanding parallel performance of the solver must be attributed to the fact that sequential solvers at that time did not yet use sophisticated heuristics like today's state-of-the-art solvers do.

Other early approaches to parallel SAT solving are Zhang's *PSATO* [ZBH96] and *PSatz* by Jurkowiak *et al.* [JLU05]. *PSATO* is targeted at networks of workstations. It introduced the *guiding path* technique for exploratory problem decomposition. *PSATO* is based on external parallelization of the sequential solver *SATO*. *PSatz* by Jurkowiak *et al.* [JLU05] is a parallel variant of the sequential solver *Satz*. It employs a very similar approach to parallelization as *PSATO*, but uses work-stealing techniques for load-balancing. Both, *PSATO* and *PSatz*, do not establish a distributed learning process, which is crucial for exploiting the potential of modern SAT solving methods in parallel environments.

*PaMiraXT* and *PaSAT* are parallel SAT solvers that both establish a distributed learning process, but are based on contrary design principles: *PaMiraXT* by Schubert *et al.* [SLB09] is a parallel SAT solver designed for networks of shared-memory parallel computers. It is based on the centralized Master/Worker model, where the master is responsible for steering problem decomposition and load balancing. The master also serves as a hub for collecting and disseminating lemmas among the clients. Due to the completely centralized architecture, the scalability of this approach is limited. The authors present a limited performance evaluation for a distributed environment consisting of 3 nodes with 8 cores in total. Furthermore, the benchmark suite used to evaluate *PaMiraXT* consists of unsatisfiable and satisfiable instances. Only cumulative results are given, such that it is not possible to assess to which extent the overall speedup is caused by super-linear speedups on satisfiable instances. Unlike SATCIETY, *PaMiraXT* does not implement fault-tolerance.

The parallel SAT solver *PaSAT* by Blochinger *et al.* [BSK03a] is targeted for tightly-coupled distributed memory architectures, like HPC clusters. It is based on a fully distributed task pool execution model for parallelizing the search process. Additionally, *PaSAT* establishes a distributed parallel learning process based on mobile agents. While the fully distributed task pool approach ensures scalability, *PaSAT* is not fault-tolerant and thus is not appropriate for environments with volatile hosts. The performance of *PaSAT* has been evaluated in a distributed environment comprised of 24 nodes.

## 38.2 Parallel SAT Solving on Grids

*ZetaSAT* by Blochinger *et al.* [BWKW05] is a parallel SAT solver for Desktop Grids. It is built on top of the discontinued Master/Worker based Desktop Grid platform *ZetaGrid* [Wed]. Due to the limitations of this class of Desktop Grids, *ZetaSAT* uses a centralized task pool and does not communicate lemmas among the nodes.

*SATU* by Hyvärinen *et al.* [HJN06] introduces the concept of *scattering* (see Chapter 32) that allows only a limited form of lemma exchange. Whether the performance of scattering is on par with decomposition based on guiding paths is an open question. The solver is based on the Master/Worker execution model.

*SDSAT* and *CL-SDSAT* [Hyv09] are parallel portfolio solvers based on randomization. In contrast to *SDSAT* which implements no lemma exchange at all, *CL-SDSAT* incorporates a limited form of dynamic learning which is tailored for Grids comprised of batch controlled resources, where individual jobs are not able to communicate directly. For solver jobs which are terminated prematurely by the Grid scheduler because they exceeded their resource limits, the lemmas deduced so far are communicated to the master node and stored in a central clause database. When additional solver jobs for the same instance are submitted some of these lemmas are selected by a heuristic and added to the initial clause databases. Unfortunately, the evaluation results for both solvers are of limited use, as speedups are either computed compared to the parallel solver using less compute nodes instead of the sequential solver or determined within a non-dedicated Grid. Nevertheless, with lemma exchange enabled the speedups are moderate ranging from $2.9$ to $15.5$ for $64$ processors.

*GridSAT* [CW06b, CW06a] by Chrabakh *et al.* is a parallel SAT Solver especially designed for *Globus* based Service Grids. The basic parallel procedure employs exploratory problem decomposition controlled by a dedicated master node. More precisely, the master node acts as a scheduler based on information delivered by external resource management services. It is also responsible for storing checkpoints. The maximum size of the lemmas which are selected for exchange is dynamically adjusted in order to adapt to the available network bandwidth. *GridSAT* is able to dynamically include batch controlled resources. When such resources become available, tasks are migrated from interactive nodes to these resources in order to exploit the additional computational power for the time allotted by the batch system.

While all discussed approaches are based on centralized control, SATCIETY is to the best of our knowledge the first parallel SAT solver for Grid environments that employs a decentralized execution model, which is able to provide good performance and scalability even under a high degree of volatility and heterogeneity.

# Part IX

# Conclusions

# 39 Summary

Desktop Grid Computing has become a valuable tool for scientific capacity computing at least on par with the world's largest supercomputers with respect to raw floating point computing power. The Master/Worker execution model has been the prevalent paradigm for computing on Desktop Grids. Its simplicity has helped the Desktop Grid approach to obtain a remarkable record of success.

However, going beyond embarrassingly parallel applications necessitates a paradigm shift from the Client/Server to a Peer-to-Peer interaction model. Doing so requires retrofitting the software stack of Desktop Grid Computing systems on all layers with novel methods and facilities to support Peer-to-Peer interaction and operation. The necessary enhancements are fundamental and affect both system architecture and distributed algorithms but allow for executing a class of advanced parallel applications formerly not suitable for Desktop Grids – task-parallel Irregularly Structured Problems.

Enabling Desktop Grid Computing systems to execute this kind of applications poses a number of challenges: First, the incorporation of Peer-to-Peer methods dramatically enlarges the design space of parallel applications which renders adoption of the functional exhaustiveness of existing platforms impossible. Instead, applications must be able to extend the Desktop Grid system with custom functionality optimized for the given use case. This extensibility comes at the price of increased platform complexity as security, compatibility, and accountability must be ensured in the resulting multi-authority modular environment. Management is a second cross-cutting aspect that becomes substantially more complicated in the context of a multi-authority large-scale distributed system. Third, the necessity of Peer-to-Peer interaction and the HPC characteristics of ISPs induce new requirements on the network substrate that are not satisfied by existing ones. Finally, to ensure scalability the centralized execution model of existing platforms must be replaced by a decentralized model. Unfortunately, distribution renders load balancing, fault-tolerance, and termination detection significantly more difficult.

To tackle the problems related to extensibility in multi-authority Desktop Grid environments, we proposed a novel isolating module management system based on the idea of providing configurable, shared, and ad-hoc isolation environments to deploy modules according to arbitrary manually contributed or automatically deduced isolation constraints. Solving the underlying Module Isolation Problem for systems of considerable size is despite its NP-hardness possible with approximation algorithms for graph coloring and with exact state-of-the-art methods for solving Pseudo Boolean problems. The resulting system is first of a kind and not only applicable for Desktop Grid Computing middleware but as well for other multi-authority modular systems like Cloud Computing infrastructures and Enterprise Mashups.

Managing Desktop Grid systems is difficult because of scattered authority, distribution, large scale, and volatile resources. We proposed a set of standards-based management services that addresses each of these issues: role-based management ensures that a party can only access its own managed objects, Peer-to-Peer management hides spatial distribution, agent cascading and bean clustering helps in handling the sheer size of Desktop Grids, and disconnected management can be used to manage peers that are temporarily offline. While these are only initial steps towards a comprehensive management solution, they considerably simplify operating Desktop Grids and other systems with similar properties.

The challenges on the network layer are met by ORBWEB, a hybrid Peer-to-Peer network substrate based on the mature *eXtensible Messaging and Presence Protocol*. The requirements for scalable and high performance peer groups, a fail-stop distributed system model, and efficient multicast communication are fulfilled by adopting existing standards and by complementing them with network component detection, Peer-to-Peer communication, virtual topologies, topology-aware probabilistic multicast, and efficient XML encoding. As substantiated by experimental results, ORBWEB scales to thousands of peers.

Support for many-to-one communication is provided by a novel capability-aware information aggregation system that turns the heterogeneity prevalent in Desktop Grids into an advantage by distributing the onus of aggregation according to the stability and performance of participating peers. The allocation quality of the resulting aggregation tree has been shown to be superior to that of existing systems.

To realize the required decentralized execution model, COHESION contributes a distributed task pool that leverages the properties of ORBWEB to implement an execution environment for task-parallel applications with unpredictable and evolving interaction patterns. Distributed termination detection is accomplished by a novel algorithm that tolerates task duplication which is an inevitable consequence of uncertainty in partially synchronous real-world systems. The task pool is shown to scale to hundreds of peers with high efficiency.

In conclusion, the resulting next generation middleware takes the concept of Desktop Grid Computing one step further by opening up a new class of problems with many important applications for efficient and scalable execution on one of today's most demanding parallel execution environments. To substantiate that COHESION in fact can enable this kind of applications, we created SATCIETY, a state-of-the-art distributed SAT solver that is – with respect to system aspects – the most sophisticated available at the time of this writing.

# 40 Relevance to other Areas of Research

While the results of this work are first of all relevant to research on Desktop Grids, the underlying methods to transparently handle volatility, heterogeneity, and non-uniform communication costs are of interest in other important areas of research. This includes multi- and manycore systems, Cloud Computing, and Exascale Computing.

## 40.1 Multi-/Manycore Systems

We haven't seen any significant increase in CPU clock rates over the last years. Going far beyond the 4 GHz barrier proves to be difficult[1] as power consumption and heat dissipation grow inevitably with clock speed. However, *Moore's Law* [Moo75] – saying that the number of transistors on an integrated circuit doubles approximately every two years – persists as the number of transistors per chip can still be increased at a high rate, now by increasing the number of cores per chip. Today, there are systems with six cores commercially available and Intel has already finished a design for an 80-core processor [VHR+08]. However, sequential programs no longer automatically benefit from increased CPU performance as the per-core performance is not increasing significantly any more. Thus, one of the presumably most important areas of computer science of the next decade will be the parallelization of software for multicore processors [Sut05]. Since, the availability of parallelism within programs is unlikely to grow at the same speed as the number of cores will, future processors will be heterogeneous with special purpose cores and will have the ability to suspend and resume cores on demand to save energy. A prominent example for this new kind of processor is the already available and widely used *Cell* processor [KDH+05]. Another major obstacle for exploiting parallelism at a larger scale is the limited bandwidth of the interconnect between cores. Hence, future systems will have *non-uniform memory access* (NUMA) characteristics.

For manycore architectures consisting of thousands of cores the expected changes are even more fundamental. Time-slicing, for example, is deemed to become obsolete as CPU time is no longer the most precious resource and context switching makes inefficient use of the now precious resources energy, on-chip memory, and off-chip bandwidth [ACC+09]. Instead, scheduling will be done by spatial partitioning and on-demand reallocation of the CPU cores. Applications must be able to cope with this core volatility to use the underlying hardware most efficiently.

In summary, many essential system properties of Desktop Grids and multi-/manycore processors will converge.

1 Although experimental systems with liquid nitrogen cooling can be operated at considerably higher clock rates.

## 40.2 Cloud Computing

*Cloud Computing* [nis11] deals with on-demand provisioning of infrastructure (hardware resources, like CPU time, memory, network bandwidth, and storage capacity), platforms (software, like operating systems, middleware, and application servers), and applications (like webmail, word or spreadsheet processors, and collaboration tools) with minimal management effort for the customer. The provided resource is taken from a shared pool and is made accessible as a service over a network. The three related service models are called *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS). Typically, PaaS provisioning includes IaaS provisioning, and SaaS provisioning includes PaaS and IaaS provisioning.

An essential characteristic of a Cloud Computing solution is the ability to allocate and release additional resources on demand. This feature is called *elasticity* in Cloud Computing jargon. A well-known IaaS provider is *Amazon* with the *Elastic Compute Cloud* (EC2) [Ama]. Customers can rent hosts with different capabilities (concerning CPU speed, amount of available main memory, etc.) and are billed on a per-usage basis. So called *spot instances* are hosts that are not assigned permanently. Instead, they are assigned temporarily to the customer with the momentarily highest bid and are deprived as soon as he is outbid by another customer. Hence, distributed applications deployed on an ensemble of EC2 spot instances must be able to cope with both volatility and heterogeneity of resources. Moreover, if the physical hosts are not collocated at the same site or several small instances are running as virtual machines on the same physical host, the inter-instance communication costs are non-uniform.

Volatility, heterogeneity, and non-uniform communication costs are not the only similarities between Cloud and P2P Desktop Grid Computing. Middleware for both is concerned with running applications from multiple issuers on the same set of resources concurrently. Thus, the I-OSGI module isolation container of COHESION qualifies as an efficient PaaS container for Cloud Computing middleware.

## 40.3 Exascale Computing

A *petascale system* is a computer system – consisting of hardware, operating system, and application – that is capable of delivering performance in excess of one petaFLOPS ($10^{15}$ FLOPS). Several petascale systems exist at the time of this writing one of them being the Chinese *Tianhe-1A* known from Chapter 1, which leads the TOP500 list [TOP10] of the worlds fastest[1] supercomputers with 2.57 petaFLOPS.

According to an outlook from the *International Exascale Software Project* (IESP) [ies], the first *exascale systems* will emerge in the 2018-2020 timeframe. With a performance of at least one exaFLOPS ($10^{18}$ FLOPS) they will be three orders of magnitude faster than *Tianhe-1A*. However, as outlined in the IESP roadmap [DBM+11], there are still many open research questions concerning critical aspects on all layers of the system that have

1 The TOP500 table lists only systems that are able to run the *Linpack* benchmark. There are special purpose architectures that are able to achieve petaFLOPS performance as well [Pet].

to be solved to implement the exascale vision. In particular, exascale-enabled runtime systems must be able to cope with very dynamic environments with unforeseen variability in application load and in availability and performance of resources. A major reason for the latter is dynamic power management that will be indispensable in exascale systems to reduce overall power consumption (which is expected to be in the range of 10-100 MW) and to avoid thermal damages in increasingly compact hardware designs. The MTTF (mean time to failure) of an entire exascale system is expected to be in the range of a few minutes [DBM+11]. Thus, the occurrence of failures during the execution of an exascale application will be the rule rather than the exception. In fact, exascale systems will need to be able to cope with a continuous stream of error conditions by constant reconfiguration on all levels of the hardware/software stack [CGG+09].

# 41 Future Perspectives

A legitimate question to ask is, whether Desktop Grid Computing will be of any importance 20 years from now? The answer to this question depends on whether the conditions that make it attractive today will persist. These conditions are an increasing demand for computing power and the huge compute capacity of billions of personal computers world-wide available at extremely low-cost.

The projected demand for science-related compute capacity in Germany grew rapidly from roughly 100 teraFLOPS for the time period from 2005-2007, approximately one petaFLOPS for the time period from 2007-2009, to about eight petaFLOPS per year in 2010 [BHL05]. This tremendous increase surpasses the performance improvements as predicted by David House in a corollary to Moore's Law saying that the performance of integrated circuits doubles every 18 months. Thus, there will likely be a constant shortage of compute capacity. The situation is no different in other countries and will not change in the future as many grand-challenge problems from many different areas of science cannot be tackled with today's supercomputers. This stimulates the development of exascale systems as described in Chapter 40.

We believe that the demand for Desktop Grid Computing capacity will increase for another reason: The suitability of applications for execution in a Desktop Grid depends critically on the *compute intensity*, i.e., the ratio of time spent on performing computational operations to that of communicating. The lower bound on the compute intensity for applications suitable for execution in a Desktop Grid is determined by the ratio of processor performance and network bandwidth/latency. This ratio will probably decrease with broad application of two new networking technologies: The first are optical fiber networks. According to *Nielsen's Law* [Nie98] the bandwidth available to end users doubles every 21 months – slower than the number of transistors of a processor according to *Moore's Law* [Moo75]. However, the broad adoption of optical fiber network technology will probably overthrow this relation as *Butter's Law* [Teh00] states that the bandwidth of an optical fiber is doubling every nine months. Thus, bandwidth-related inefficiencies of applications with lower compute intensity are likely to become less dominant.

The second technology trend that may result in more available bandwidth and lower latency is not new but has reached a degree of maturity and pervasiveness that allows for new kinds of applications – wireless networking. By applying the P2P paradigm to the wireless networking infrastructure in metropolitan areas, an ad-hoc multi-hop high-speed P2P network that covers entire cities can be created by interconnecting private and/or public wireless local area networks. This way large amounts of data can be exchanged efficiently between computers within such a network bypassing the limited infrastructure of the Internet Service Provider.

The impressive growth in the number of personal computers deployed world-wide has been discussed in the introduction already. However, personal computers are not the only computing devices. There, are ever more smart devices with considerable computing power. The most important are smart phones, tablets, cars, set-top boxes, and game consoles. If these are exploited in the future in the same way as personal computers are today, the potential of Grid Computing on commodity hardware will increase by factors. *Grid Anywhere* [TSSE10] is a first step in this direction exploiting the computational resources of set-top boxes used to receive interactive digital television. The potential of the approach is huge as virtually every household has a television. The same is true for hand held devices like smart phones and tablets. While their usability is problematic while being on battery, this is not true when they are in a cradle for recharging at night.

However, making all this computing power accessible will be challenging. Volunteer Computing lives from enthusiasts donating their resources for projects with a public benefit. While leader-board competition among individuals and teams helps in attracting and retaining participants [ACK+02], additional incentive has to be provided to attract a broader audience and to tap the full potential of Desktop Grid Computing. Possible approaches are amongst others devices subsidized by parties requiring massive compute capacity, platforms realizing a computational exchange model [CM02, PHP+03], and commercial Cloud Computing solutions based on Desktop Grids, where resource donors are rewarded for providing their devices.

Considering these facts and trends, we think that Desktop Grid Computing *will* be important 20 years from now. Provided that the laws of Moore and Butter persist, we will see processors that are thousand times faster than today's fastest processors but networks that are million times faster than today's fastest networks (with respect to bandwidth). Most probably, we will see Desktop Grids executing applications that today are executable on supercomputers only.

# Appendix A

# Scalability Theory

*Scalability* is an important design goal and performance metric for distributed systems. However, depending on which aspect of a system is considered, there are different notions of this term. To provide a sound foundation for statements concerning scalability made within this thesis, we give a formal definition of a *scalable system*. After that, we discuss the term in the context of parallel computing.

## A.1 Formal Scalability Framework

Let $k$ be a *scale parameter* of a system $\mathcal{S}$ and $m$ be a *metric* quantifying a quality property of the system. For example the size of a system could be the scale parameter and the load per processor could be the metric. If $m(\mathcal{S};k)$ denotes the value of the metric $m$ for the system $\mathcal{S}$ at scale $k$, then $M(\mathcal{S};k) = m(\mathcal{S};k)/m(\mathcal{S};1)$ denotes the associated *normalized metric*. With $C(k)$ being some *scalability criterion*, then scalability can be defined as follows:

**Definition A.1.1.** *A system $\mathcal{S}$ is called* absolutely scalable *w.r.t. $m$ and $C$, if and only if there is a $k'$ for which*

$$M(\mathcal{S};k) \le C(k) \tag{A.1}$$

*holds for all $k > k'$.*

Even if a system is not absolutely scalable, it can be more scalable than another one. Let $M(\mathcal{S})$ and $M(\mathcal{S}')$ be the normalized metrics for the same metric $m$ and two systems $\mathcal{S}$ and $\mathcal{S}'$.

**Definition A.1.2.** *A system $\mathcal{S}$ is called* relatively scalable *w.r.t. $\mathcal{S}'$ and $m$, if and only if*

$$\lim_{k\to\infty} \frac{M(\mathcal{S};k)}{M(\mathcal{S}';k)} = 0 \tag{A.2}$$

*holds.*

## A.2 Computational Scalability

The prime measures of scalability in parallel computing are *parallel speedup* and *parallel efficiency*. They are defined as follows:

**Definition A.2.1.** *Let* $T_1 := T(1)$ *be the* sequential execution time, *i.e., the time a single processor needs to complete a given task, and let* $T(p)$ *be the* parallel execution time, *i.e., the time required by a* $p$*-way multiprocessor for the same task. Then the* parallel speedup $S$ *is defined as*

$$S(p) = \frac{T_1}{T(p)}. \tag{A.3}$$

*The* parallel efficiency $E$ *is defined as*

$$E(p) = \frac{S(p)}{p} = \frac{T_1}{pT(p)}. \tag{A.4}$$

Typically parallelization results in sublinear speedups where an $n$-way parallel program yields an $m$-fold speedup with $1 < m < n$. However, as depicted in Figure A.1 there are other types of speedups: slowdowns (the parallel program is slower than the sequential one), linear speedups (an $n$-way parallel program yields an $n$-fold speedup), and superlinear speedups (am $n$-way parallel program yields am $m$-fold speedup with $n < m$).

Every sequential program consists of parts that are suitable for parallel execution and others that are not. The ratio between the time spent in the latter with respect to the whole execution time is called the *sequential fraction* $\sigma \in [0,1]$. Typical examples are parallel programs where each processor has to read the whole input of the sequential program. There are three laws – two of them with associated scalability definitions – that can be used to model the scalability of such programs, *Amdahl's Law*, *Gustafson's Law*, and Gunther's *Universal Scalability Law*. Figure A.2 shows the characteristics of these laws as a function

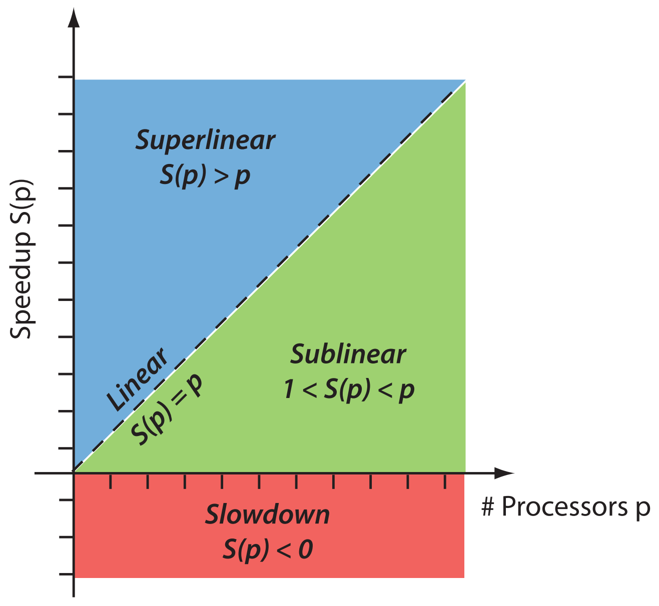


**Figure A.1:** Different types of parallel speedup are slowdowns (negative speedup), sublinear, linear, and superlinear speedups

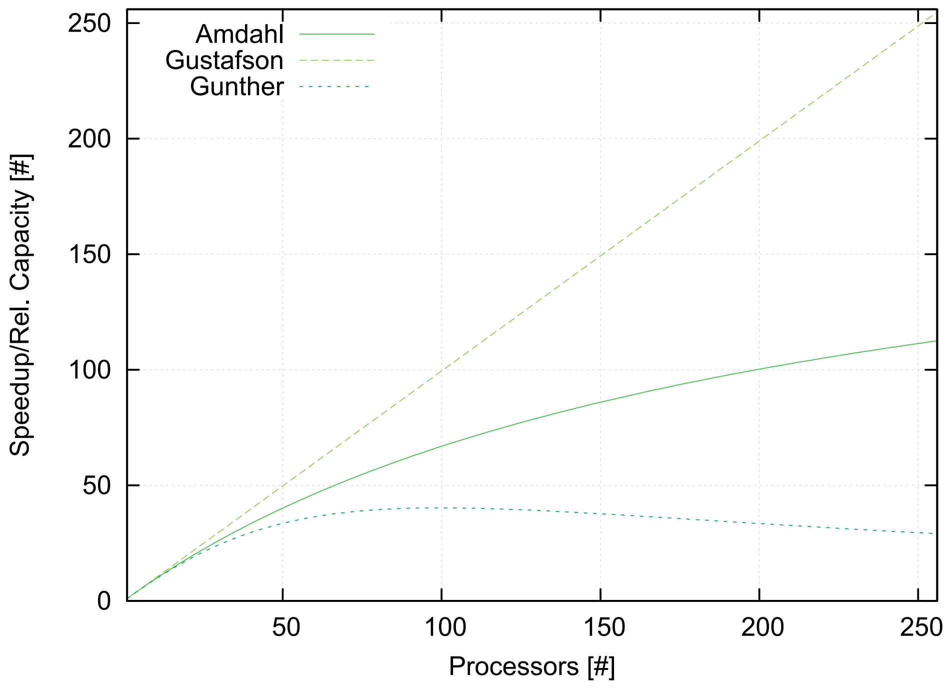


**Figure A.2:** Characteristics of the three different scalability laws for a sequential fraction $\sigma = 5 \times 10^{-3}$ and $\kappa = 1 \times 10^{-4}$

of the number of processors.

## A.2.1 Amdahl's Law

Let $T_1$ be the sequential execution time for a given program with an irreducible sequential fraction $\sigma$. *Amdahl's Law* [Amd67] says that if the task can be equipartitioned into $p$ subtasks (see Figure A.3a), the speedup by parallel execution on a $p$-way multiprocessor has an upper bound of

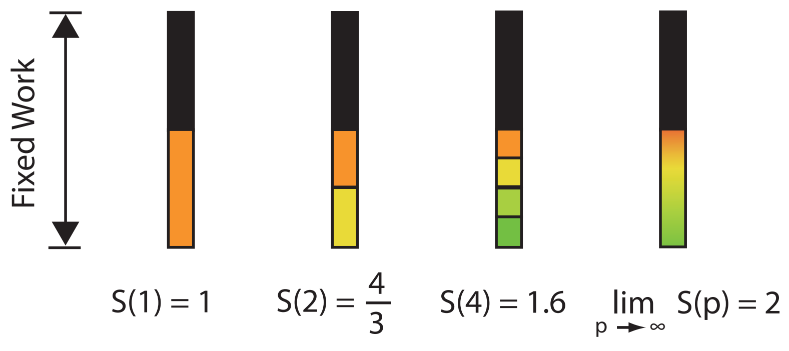
$$S^A_{max}(\sigma;p) = \frac{T_1}{\sigma T_1 + \left(\frac{1-\sigma}{p}\right) T_1} = \frac{1}{\sigma + \frac{1-\sigma}{p}}. \tag{A.5}$$

**Definition A.2.2.** *A system $\mathcal{S} = (\mathcal{P},p)$ of a program $\mathcal{P}$ and $p$ nodes is called* strongly scalable *w.r.t. some scalability criterion $C(p)$, if and only if it is absolutely scalable w.r.t. $m(\mathcal{S};p) = T(p)$ and $M(\mathcal{S};p) = S(p)^{-1}$ for a task $T$ with fixed size and $C(p)$.*

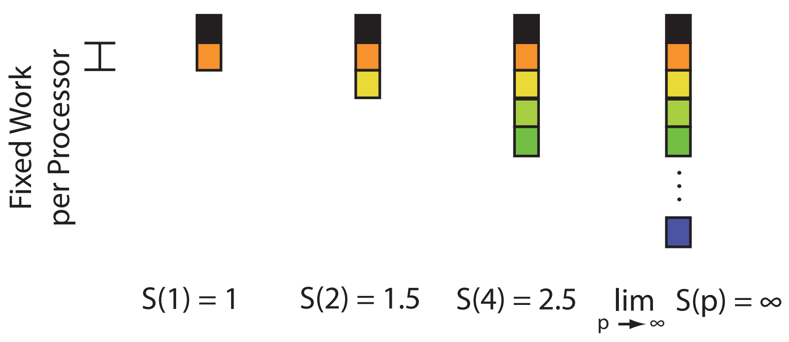
Note, that the sequential fraction limits the achievable speedup, as

$$\lim_{p\to\infty} S^A_{max}(\sigma,p) = \frac{1}{\sigma} \tag{A.6}$$

holds.



**(a)** Amdahl's Law



**(b)** Gustafson's Law

**Figure A.3:** Comparison of Amdahl's and Gustafson's Law for a program with a sequential fraction $\sigma = 0.5$

## A.2.2 Gustafson's Law

Amdahl's Law prohibits linear speedups for problems with a non-vanishing sequential fraction. Given the fact that real-world programs typically do have a non-vanishing sequential fraction, the prospect of large-scale parallel computing seems to be very limited. However, there are real-world problems that are efficiently solved using large-scale parallel processing — the key point is that these problems are large. Gustafson bore this point in mind when he reconsidered the definition of scalability in the context of large-scale parallel computing. His formulation is based on the idea of linearly scaling up the size of the input with the number of processors (see Figure A.3b). This co-scaling interpretation of scalability results in *Gustafson's Law* [Gus88]

$$S^G_{max}(\sigma,p) = \sigma + (1-\sigma)p. \tag{A.7}$$

Obviously, Gustafson's Law is a linear function in $p$ and thus allows for linear speedups. With $\tau_1$ being the size of the input for the sequential program, the related notion of *weak scalability* is defined as follows:

**Definition A.2.3.** *A system $\mathcal{S} = (\mathcal{P},p)$ of a program $\mathcal{P}$ and $p$ processors is called* weakly scalable *w.r.t. some scalability criterion $C(p)$, if and only if it is absolutely scalable w.r.t.*

*$m(\mathcal{S};p) = T(p)$ and $M(\mathcal{S};p) = S(p)^{-1}$ for a task $\tau(p) = p\tau_1$ and $C(p)$.*

### A.2.3 Universal Scalability Law

Amdahl's Law gives an upper bound on the achievable speedup through parallelization. This is due to the fact that Amdahl considered the parallel processing as totally independent. However, in most cases – embarrassingly parallel applications are an exception to this rule – this assumption does not hold, as parallel programs have additional cost associated with keeping shared data structures coherent. Gunther's *Universal Scalability Law* (USL) [Gun93] takes this into account by adding another parameter $\kappa$ to Amdahl's Law, which he appropriately calls *coherence*. His notion of scalability is based on the *relative capacity*

$$C_{rel} := \frac{X(p)}{X(1)}, \tag{A.8}$$

where $X(p)$ is the throughput on $p$ processors. Both notions of scalability – speedup and relative capacity – are equivalent. With this definition the USL can be written as

$$C_{rel}(\sigma,\kappa) = \frac{p}{1+\sigma(p-1)+\kappa p(p-1)} \tag{A.9}$$

The first term in the denominator can be attributed to the ideal concurrency with linear scalability. The second one models the limiting factor of contention due to serialization or queuing. The third term models the additional effort to maintain coherency. The USL is more realistic than Amdahl's law as coherency maintenance costs are quadratic in the number of processors and thus becomes dominant for large numbers of processors and eventually results in significant performance degradation. This behavior can be observed with real systems. Thus, the USL and the associated *super-serial model* are successfully used to evaluate and predict (by extrapolation) parallel system performance.

### A.2.4 A Note on (Super-)Linear Speedups

Every sensible parallel program must consolidate results somehow. Thus, the sequential fraction of a sensible program is actually never zero. Hence, according to Amdahl's Law no program exhibits strong linear scalability. However, in practice, one can observe such theoretically impossible linear and even superlinear speedups. A prominent example is parallel search on datasets with non-uniformly distributed matching elements. For a detailed explanation see [Sut08]. Another source for superlinearity are hidden increases in processing resources. A prime example is the size of memory caches that are typically private to processors and hence grow linearly with their number.

# Appendix B

# Notation

To describe the APIs and algorithms discussed in Parts V and Part VII, we adopt and adapt the asynchronous event-based component model and the associated notation of Cachin *et al.* [CRG11]. Note that we do not follow neither their model nor their notation strictly, but introduce modifications and extensions where necessary. In the following, we describe all major concepts and notations in order to enable the reader to understand the discussion of the aforementioned APIs and algorithms.

## B.1 Modules

Cachin's component model is centered around the concept of modules. A *module* is an API specification identified by a unique name and is characterized by a set of *properties* that must be guaranteed by implementations of the API. The API itself is specified based on the notion of *events*. An event is a tuple (*Module-Name*, Event-Type, $\{attr_1, \ldots, attr_n\}$) and is denoted as ⟨ *Module-Name*, Event-Type | $attr_1, \ldots, attr_n$ ⟩. *Module-Name* is the (abbreviated) name of the module that defines the event. *Event-Type* is a descriptive name for the event. An event optionally carries information by means of the *attributes* $attr_i, 0 < i \leq n$. In case no attributes are defined, the event is denoted as ⟨ *Module-Name*, Event-Type ⟩. In case we reference an event in running text where the defining module is unambiguously determined by the context, we simply write ⟨ Event-Type ⟩. There are two kinds of *events*: *Requests* are used to invoke a service of a module. They are the inputs to

**Algorithm B.1** Interface of the Sample module

```
Module:
    Name: Sample Module, instance sm.
Requests:
    ⟨ sm, R | x, y ⟩: Description of the request R that specifies the meaning of attributes x and y.
    ...
Indications:
    ⟨ sm, I | z ⟩: Description of the indication I that specifies the meaning of the attribute z.
    ...
Properties:
    SM1: Property-Name Description of the property.
    ...
```

a module. *Indications* are used to signal a condition to a module. They are the outputs of a module. A full module definition is shown in Figure B.1.

## B.2 Protocols

The implementation of a module is called a *protocol* (although it might perform local operations only and might never interact with another process). A sample protocol definition is shown in Figure B.2.

A protocol is defined by a set of named *event handlers* that are invoked in response to incoming events. The name of an event handler is a pair (*Module-Name*, $k$), where $k$ is a serial number starting from $0$, denoted as MN-$k$, where MN is an abbreviated and capitalized version of *Module-Name*. A handler is denoted as a clause **upon event** *event* **do** *body*, with *body* being a sequence of pseudo code instructions. We do not describe all pseudo code instructions in detail here. In case they are not self-explanatory, we give a short description in the running text.

Pseudo code instructions can access the attributes of the triggering event using their symbolic names, as shown in the first line of the body of (S-1). Furthermore, the body of an event handler may interact with other modules via two different instructions: **trigger** *event*; asynchronously performs a request on a module or raises an indication, i.e., enqueues the associated event in the process' event queue. **call** *event*;, in contrast, is valid for requests only and synchronously invokes associated handlers triggered by the given *event*. One can think of called event handlers as being inlined at the call site. For calls, event attributes are *by-reference* parameters and hence can be used to transfer information from a triggered event handler to the call site. The handler can be thought of as being inlined at the call site. For an example see the body of (S-2) in Figure B.2.

Sometimes an event handler should only be evaluated, if a condition holds. Such a *conditional event handler* is denoted by inserting a **such that** *condition* expression right after the *event* specification of the handler. We sometimes also use a condensed notation for conditional handlers that makes use of pattern matching on the event. This is denoted by replacing the symbolic names for event attributes with patterns that might in turn contain symbolic names. A fully specified plain event handler (S-1) and an event handler that makes use of pattern matching (S-2) are shown in Figure B.2. The latter one is invoked only, when the first attribute of an event of type Event$_a^2$ matches *some-value*.

The body of a handler may also contain *function calls*. These are tuples (*Function-Name*, $arg_1, \ldots, arg_n$) denoted as $\textsc{Function-Name}(arg_1, \ldots, arg_n)$. Specific functions used in event handlers are described in the running text.

To initialize a protocol, an optional $\langle$ Init | $attr_1, \ldots, attr_n$ $\rangle$ request is automatically invoked by the runtime, when an instance of the protocol is created. The given attributes $attr_i, 0 < i \leq n$ are initialization parameters and are described in the module header using a **with** clause. These attributes are implicitly declared as variables and accessible from all event handlers of the protocol.

**Algorithm B.2** Sample protocol

```
Implements:
    Sample, instance $S$ with parameter $p$.

Uses:
    Module-A, instance a.
    Protocol-B, instance b.

upon event ⟨ s, Init | $p$ ⟩ do                                                  ▷ S-0
    $state := \bot$;

upon event ⟨ b, $\text{Event}_b^2$ | $attr_1, \ldots, attr_n$ ⟩ such that Condition do     ▷ S-1
    $sum := attr_1 + \ldots + attr_n$;
    trigger ⟨ a, $\text{Event}_a^1$ | $sum$ ⟩;
    ...

upon event ⟨ a, $\text{Event}_a^2$ | some-value ⟩ do                              ▷ S-2
    call ⟨ b, $\text{Event}_b^1$ | $result$ ⟩;
    $state := result$;
    ...
```

### B.2.1 Composition

The functionality of a module or protocol can be used by two different means: aggregation and inheritance. *Aggregation* allows to specify a set $\{(name_1, entity_1), \ldots, (name_n, entity_n)\}$ of abstract module interfaces or concrete protocol instances $entity_i$ with associated names $name_i$ on which the declaring protocol depends on. Aggregation is denoted by means of a special **Uses** section in the protocol definition. Each dependency is denoted using a *Protocol-Or-Module-Name*, **instance** *Instance-Name* statement. A module can invoke all requests and consume all indications from its dependencies by setting the *Module-Name* of the respective event to *Instance-Name* like in both (S-1) and (S-2). Additionally, in case the dependency is a protocol rather than a module, access to all its variables is possible and denoted as Instance-Name.$v$ for variable $v$.

*Inheritance* imports all properties, handlers, variables, and dependencies from another protocol. Inheritance is denoted by replacing the standard module header of a module definition by an **Extends:** *Bequeathing-Module* **with** *Extension-Name* statement. In case an event handler from a protocol replaces one from the protocol it inherits from, this is explicitly stated in the running text.

## B.3 Event Handling

We assume that processes serially execute event handlers in a mutually exclusive and atomic way[1]. The execution order is *First-In-First-Out* (FIFO). Every triggered event is enqueued to the single event queue of a process and eventually executed as long as the hosting process

1 For calls this property holds for the handler's body after the called handlers are inlined.

is correct (in the sense described in Chapter 16). Events triggered during the execution of a handler are enqueued at the end of the queue. One can think of a protocol as a finite state machine whose transitions are triggered by the reception of events.

# Bibliography

[AB07] Nabil Abdennadher and Régis Boesch: *Towards a peer-to-peer platform for high performance computing*. In *Proceedings of the 2nd International Conference on Advances in Grid and Pervasive Computing (GPC '07)*, volume 4459 of *Lecture Notes in Computer Science*, pages 412–423, Paris, France, May 2007. Springer Berlin / Heidelberg. (Cited on pages 53 and 59)

[ABB+86] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young: *Mach: A new kernel foundation for unix development*. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Altanta, GA, USA, June 1986. Usenix Association. (Cited on pages 25 and 45)

[ACC+09] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Ko, J. Levesque, D. A. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling: *Exascale software study: Software challenges in extreme scale systems*. Technical report, DARPA IPTO, September 2009. (Cited on page 285)

[ACG10] Cosimo Anglano, Massimo Canonico, and Marco Guazzone: *The sharegrid peer-to-peer desktop grid: Infrastructure, applications, and performance evaluation*. Journal of Grid Computing, 8(4):543–570, December 2010. (Cited on page 62)

[ACJ09] Heithem Abbes, Christophe Cérin, and Mohamed Jemni: *Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids*. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, Anchorage, Alaska, USA, May 2009. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 67)

[ACJ10] Heithem Abbes, Christophe Cérin, and Mohamed Jemni: *A decentralized and fault-tolerant desktop grid system for distributed applications*. Concurrency and Computation: Practice & Experience - Advanced Scheduling Strategies and Grid Programming Environments, 22(3):261–277, March 2010. (Cited on pages 53 and 59)

[ACJD07] Gabriel Antoniu, Loic Cudennec, Mathieu Jan, and Mike Duigou: *Performance scalability of the jxta p2p framework*. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*, pages 109–118, Long Beach, CA, USA, March 2007. IEEE Computer Society Los Alamitos, CA, USA. (Cited on page 181)

[ACJM10] Heithem Abbes, Christophe Cérin, Mohamed Jemni, and Yazid Missaoui: *Fault-tolerance for pastrygrid middleware*. In *Workshop Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–8, Atlanta, GA, USA, April 2010. IEEE Computer Society. (Cited on page 59)

[ACJS10] Heithem Abbes, Christophe Cérin, Mohamed Jemni, and Walid Saad: *Fault tolerance based on the publish-subscribe paradigm for the bonjourgrid middleware*. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (Grid'10)*, pages 57–64, Brussels, Belgium, 2010. IEEE Computer Society. (Cited on page 67)

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer: *Seti@home: an experiment in public-resource computing*. Communications of the ACM, 45(11):56–61, November 2002. (Cited on pages 36 and 290)

[AD09] Heithem Abbes and Jean Christophe Dubacq: *Analysis of peer-to-peer protocols performance for establishing a decentralized desktop grid middleware*. In Eduardo César, Michael Alexander, Achim Streit, Jesper Larsson Träff, Christophe Cérin, Andreas Knüpfer, Dieter Kranzlmüller, and Shantenu Jha (editors): *Proceedings of the Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 235–246, Las Palmas de Gran Canaria, Spain, August 2009. Springer Berlin / Heidelberg. (Cited on page 67)

[Adoa] Adobe Systems Inc.: *AMF 3 Specification*. http://opensource.adobe.com/wiki/download/attachments/1114283/amf3_spec_05_05_08.pdf, Accessed: 3/2011. (Cited on page 257)

[Adob] Adobe Systems Inc.: *BlazeDS website*. http://opensource.adobe.com/wiki/display/blazeds/BlazeDS, Accessed: 3/2011. (Cited on page 257)

[Adoc] Adobe Systems Inc.: *Flex SDK website*. http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK, Accessed: 3/2011. (Cited on page 257)

[AHJN05] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet: *Performance evaluation of jxta communication layers*. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 251–258, Cardiff, Wales, UK, May 2005. IEEE Computer Society Washington, DC, USA. (Cited on pages 172 and 180)

[AHM+03] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long: *Managing flash crowds on the internet*. In *Proceedings of the 11th*

*IEEE/ACM International Symposium on Modeling Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 246–249, Orlando, FL, USA, October 2003. IEEE Computer Society Los Alamitos, CA, USA. (Cited on page 270)

[AM98] Lorenzo Alvisi and Keith Marzullo: *Message logging: Pessimistic, optimistic, causal, and optimal*. IEEE Transactions on Software Engineering, 24(2):149–159, February 1998. (Cited on page 58)

[Ama] Amazon.com, Inc: *Amazon Elastic Compute Cloud website*. http://aws.amazon.com/de/ec2/, Accessed: 3/2011. (Cited on page 286)

[Amd67] Gene M. Amdahl: *Validity of the single processor approach to achieving large scale computing capabilities*. In *Proceedings of the Spring Joint Computer Conference (AFIPS '67)*, pages 483–485, Atlantic City, NJ, USA, April 1967. ACM New York, NY, USA. (Cited on page 293)

[And04] David P. Anderson: *BOINC: a system for public-resource computing and storage*. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 365–372, Pittsburgh, USA, November 2004. IEEE Computer Society Washington, DC, USA. (Cited on pages 35, 43, and 53)

[APG+10] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle: *Multi-tenant soa middleware for cloud computing*. In *Proccedings of the 3rd International Conference on Cloud Computing (Cloud '10)*, pages 458–465, Miami, Florida, USA, July 2010. IEEE Computer Society Los Alamitos, CA, USA. (Cited on page 71)

[AS06] Anbulagan and John Slaney: *Multiple preprocessing for systematic SAT solvers*. In *Proceedings of the 6th International Workshop on the Implementation of Logics*, volume 212, Phnom Penh, Cambodia, November 2006. (Cited on page 268)

[BAV+07] Francisco Brasileiro, Eliane Araujo, William Voorsluys, Milena Oliveira, and Flavio Figueiredo: *Bridging the high performance computing gap: the ourgrid experience*. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, pages 817–822, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society Washington, DC, USA. (Cited on pages 61 and 62)

[Bay72] Rudolf Bayer: *Symmetric binary b-trees: Data structure and maintenance algorithms*. Acta Informatica, 1:290–306, 1972. (Cited on page 143)

[BBB96] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer: *Atlas: an infrastructure for global computing*. In *Proceedings of the 7th ACM SIGOPS*

*European Workshop: Systems support for worldwide applications*, pages 165–172, Connemara, Ireland, September 1996. ACM Press New York, NY, USA. (Cited on pages 53, 54, and 56)

[BBK02] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy: *Scalable application layer multicast*. ACM SIGCOMM Computer Communication Review, 32(4):205–217, October 2002. (Cited on page 180)

[BBK$^+$07] P. Oscar Boykin, Jesse S. A. Bridgewater, Joseph S. Kong, Kamen M. Lozev, Behnam Attaran Rezaei, and Vwani P. Roychowdhury: *A symphony conducted by brunet*. Computing Research Repository (CoRR), arXiv:0709.4048v1, 2007. (Cited on pages 179 and 180)

[BCC$^+$06] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E G. Talbi, and I. Touche: *Grid'5000: a large scale and highly reconfigurable experimental grid testbed.* International Journal of High Performance Computing Applications, 20(4):481–494, November 2006. (Cited on page 56)

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu: *Symbolic model checking without BDDs*. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer. (Cited on page 245)

[BDS06] Wolfgang Blochinger, Clemens Dangelmayr, and Sven Schulz: *Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform*. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'06)*, pages 49–56, Singapore, May 2006. IEEE Computer Society Washington. (Cited on pages 24 and 43)

[BEDSV04] Julien Bourgeois, Jean Baptiste Ernst-Desmulier, François Spies, and Jerome Verbeke: *Using similarity groups to increase performance of p2p computing*. In *Proceedings of the 10th International Euro-Par Conference (Europar'04)*, volume 3149 of *Lecture Notes in Computer Science*, pages 1056–1059, Pisa, Italy, August 2004. Springer. (Cited on pages 53 and 61)

[BGNT04] K. Burbeck, D. Garpe, and S. Nadjm-Tehrani: *Scale-up and performance studies of three agent platforms*. In *Proceedings of the 23rd IEEE International Conference on Performance, Computing, and Communications (IPCCC '04)*, pages 857–863, Phoenix, Arizona, April 2004. IEEE Computer Society. (Cited on page 181)

[BHL05] A. Bode, W. Hillebrandt, and Th. Lippert: *Petaflop-Computing mit Standort Deutschland im europäischen Forschungsraum, Bedarf und Perspektiven aus Sicht der computergestützten Natur- und Ingenieurwissenschaft*. Bonn, 2005. (Cited on pages 30 and 289)

[BHMW11] Roland Bless, Christian Hübsch, Christoph P. Mayer, and Oliver P. Waldhorst: *Spovnet: An architecture for easy creation and deployment of service overlays*. In Anand R. Prasad, John F. Buford, and Vijay K. Gurbani (editors): *Advances in Next Generation Services and Service Architectures*, volume 1, pages 23–47. River Publishers, 2011. (Cited on pages 105 and 180)

[BHO+99] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky: *Bimodal multicast*. ACM Transactions on Computer Systems, 17(2):41–88, May 1999. (Cited on pages 147, 149, 151, and 154)

[BHS09] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz: *Experiments with massively parallel constraint solving*. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence (IJCAI'09)*, pages 443–448, Pasadena, California, USA, July 2009. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. (Cited on page 252)

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh (editors): *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, 2009. (Cited on page 88)

[Bie08] Armin Biere: *Adaptive restart strategies for conflict driven SAT solvers*. In Hans Kleine Büning and Xishun Zhao (editors): *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33, Guangzhou, China, May 2008. Springer. (Cited on page 272)

[Bir93] Kenneth P. Birman: *The process group approach to reliable distributed computing*. Communications of the ACM, 36(12):37–53, December 1993. (Cited on page 123)

[bit] *Bittorrent*. http://www.bittorrent.org. Accessed: 3/2011. (Cited on pages 60, 172, and 270)

[BJ87] Kenneth P. Birman and Thomas A. Joseph: *Reliable communication in the presence of failures*. ACM Transactions on Computer Systems, 5(1):47–76, January 1987. (Cited on page 123)

[BJCCP05] Javier Bustos-Jimenez, Denis Caromel, Alexandre di Costanz, and Jose M. Piquer: *Balancing active objects on a peer to peer infrastructure*. In *Proceedings of the 25th International Conference on The Chilean Computer Science Society (SCCC'05)*, pages 109–118, Valdivia, Chile, November 2005. IEEE Computer Society Washington, DC, USA. (Cited on page 58)

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou: *Cilk: an efficient multithreaded runtime system*. ACM SIGPLAN Notices, 30(8):207–216, August 1995. (Cited on page 56)

[BK03] Wolfgang Blochinger and Wolfgang Küchlin: *The design of an API for strict multithreading in C++*. In Harald Kosch, László Böszörményi, and Hermann Hellwagner (editors): *Proceedings of 9th International Conference Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 722–731, Klagenfurt, Austria, August 2003. Springer Berlin / Heidelberg. (Cited on page 41)

[BKKW99] A. Baratloo, M. Karaul, Z.M. Kedem, and P. Wijckoff: *Charlotte: Metacomputing on the web*. Future Generation Computer Systems, 15(5–6):559–570, October 1999. (Cited on pages 53 and 55)

[BKWb98] Wolfgang Blochinger, Wolfgang Küchlin, and Andreas Weber: *The distributed object-oriented threads system DOTS*. In A. Ferreira, J. Rolim, H. Simon, and S. H. Teng (editors): *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, volume 1457 of *LNCS*, pages 206–217, Berkeley, CA, USA, August 1998. Springer-Verlag. (Cited on page 41)

[BL97] Robert D. Blumofe and Philip A. Lisiecki: *Adaptive and reliable parallel computing on networks of workstations*. In *Proceedings of the USENIX Annual Technical Conference*, pages 10–24, Anaheim, CA, USA, January 1997. USENIX Association Berkeley, CA, USA. (Cited on page 56)

[BL99] Robert D. Blumofe and Charles E. Leiserson: *Scheduling multithreaded computations by work stealing*. Journal of the ACM, 46(5):720–748, 1999. (Cited on page 205)

[Blo06] Wolfgang Blochinger: *Towards robustness in parallel SAT solving*. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata (editors): *Proceedings of the International Conference on Parallel Computing (ParCo '05): Current & Future Issues of High-End Computing*, volume 33 of *John von Neumann Institute for Computing Series*, Malaga, Spain, September 2006. Central Institute for Applied Mathematics, Jülich, Germany. (Cited on pages 236 and 273)

[BMS00] L. Baptista and J. P. Marques-Silva: *Using randomization and learning to solve hard real-world instances of satisfiability*. In Rina Dechter (editor): *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494, Singapore, September 2000. Springer Berlin / Heidelberg. (Cited on page 249)

[BMvN$^+$10] Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Roelof Kemp, Nick Palmer, Nick Palmer, Thilo Kielmann, Frank Seinstra, and Ceriel Jacobs: *Real-world distributed computing with ibis*. IEEE Computer, 43(8):54–62, August 2010. (Cited on pages 54 and 57)

[BS96] M. Boehm and E. Speckenmeyer: *A fast parallel SAT-solver – efficient workload balancing*. Annals of Mathematics and Artificial Intelligence, 17(3–4):381–400, 1996. (Cited on pages 251 and 279)

[BSK01] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin: *Distributed parallel SAT checking with dynamic learning using DOTS*. In T. Gonzales (editor): *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'01)*, pages 396–401, Anaheim, CA, USA, August 2001. ACTA Press. (Cited on pages 254 and 265)

[BSK03a] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin: *Parallel propositional satisfiability checking with distributed dynamic learning*. Parallel Computing, 29(7):969–994, 2003. (Cited on page 279)

[BSK03b] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin: *A universal parallel SAT checking kernel*. In Hamid R. Arabnia and Youngsong Mun (editors): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, volume 4, pages 1720–1725, Las Vegas, NV, USA, June 2003. CSREA Press. (Cited on page 253)

[BSV03] R. Bhagwan, S. Savage, and G. Voelker: *Understanding availability*. In Michal Feldman and Shelley Zhuang (editors): *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, volume 2735 of *Lecture Notes in Computer Science*, pages 256–267, Berkeley, CA, USA, February 2003. Springer Berlin / Heidelberg. (Cited on pages 23 and 36)

[BV01] Rajkumar Buyya and Sudharshan Vazhkudai: *Compute power market: Towards a market-oriented grid*. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 574–581, Brisbane, Australia, May 2001. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 67)

[BW03] Fahiem Bacchus and Jonathan Winter: *Effective preprocessing with hyper-resolution and equality reduction*. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355, Santa Margherita Ligure - Portofino, Italy, May 2003. Springer Berlin / Heidelberg. (Cited on page 268)

[BWKW05] Wolfgang Blochinger, Wolfgang Westje, Wolfgang Küchlin, and Sebastian Wedeniwski: *ZetaSAT – Boolean satisfiability solving on desktop grids*. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, pages 1079–1086, Cardiff, UK, May 2005. IEEE Computer Society. (Cited on pages 53, 252, and 280)

[BZH06] Ali R. Butt, Rongmei Zhang, and Y. Charlie Hu: *A self-organizing flock of condors*. Journal of Parallel and Distributed Computing, 66(1):145 –161, January 2006. (Cited on pages 53 and 63)

[CAG$^+$06] Edjozane Cavalcanti, Leonardo Assis, Matheus Gaudencio, Walfredo Cirne, and Francisco Brasileiro: *Sandboxing for a free-to-join grid with support for secure site-wide storage area*. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC'06)*, pages 11–18, Tampa, Florida, November 2006. IEEE Computer Society Washington, DC, USA. (Cited on page 62)

[CB94] James M. Crawford and Andrew B. Baker: *Experimental results on the application of satisfiability algorithms to scheduling problems*. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, volume 2, pages 1092–1097, Seattle, Washington, 1994. AAAI Press/MIT Press. (Cited on pages 29, 48, and 245)

[CBA$^+$06] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray: *Labs of the World, Unite!!!* Journal of Grid Computing, 4(3):225–246, 2006. (Cited on page 182)

[CBG$^+$06] Sungjin Choi, Maengsoon Baik, Joonmin Gil, Soonyoung Jung, and Chongsun Hwang: *Adaptive group scheduling mechanism using mobile agents in peer-to-peer grid computing environment*. Applied Intelligence, 25(2):199–221, October 2006. (Cited on pages 53 and 64)

[CBK$^+$08] Sungjin Choi, Rajkumar Buyya, Hongsoo Kim, Eunjoung Byun, Maengsoon Baik, Joonmin Gil, and Chanyeol Park: *A taxonomy of desktop grids and its mapping to state-of-the art systems*. Technical report GRIDS-TR-2008-3, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, February 2008. (Cited on pages 35 and 53)

[CBL05] A.J. Chakravarti, G. Baumgartner, and M. Lauria: *The organic grid: self-organizing computation on a peer-to-peer network*. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 35(3):373–384, 2005. (Cited on page 64)

[CBL07] A.J. Chakravarti, G. Baumgartner, and M. Lauria: *Self-organizing scheduling on the organic grid*. In Manish Parashar and Salim Hariri (editors): *Autonomic Computing: Concepts, Infrastructure, and Applications*, chapter 19, pages 389–411. CRC Press New York, USA, 2007. (Cited on pages 53 and 64)

[CCEB03] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia: *Entropia: architecture and performance of an enterprise desktop grid system*. Journal of Parallel and Distributed Computing, 63(5):597–610, May 2003. (Cited on pages 35 and 53)

[CCM07] Denis Caromel, Alexandre di Costanzo, and Clément Mathieu: *Peer-to-peer for computational grids: mixing clusters and desktop machines*. Parallel Computing, 33(4–5):275–288, May 2007. (Cited on page 59)

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman: *Planetlab: an overlay testbed for broad-coverage services*. ACM SIGCOMM Computer Communication Review, 33(3):3–12, July 2003. (Cited on pages 64 and 180)

[CDDCL06] Denis Caromel, Christian Delbe, Alexandre Di Costanzo, and Mario Leyton: *Proactive: an integrated platform for programming and running applications on grids and p2p systems*. Computational Methods in Science and Technology, 12(1):69–77, 2006. (Cited on pages 54, 58, and 59)

[CDF+05] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky: *Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid*. Future Generation Computer Systems, 21(3):417–437, 2005. (Cited on pages 53 and 59)

[CDK+04] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris: *Practical, distributed network coordinates*. ACM SIGCOMM Computer Communication Review, 34(1):113–118, January 2004. (Cited on page 180)

[CDKR02] Miguel Castro, Peter Druschel, Anne Marie Kermarrec, and Antony Rowstron: *Scribe: A large-scale and decentralized application-level multicast infrastructure*. IEEE Journal on Selected Areas in Communication (JSAC), 20(8):1489–1499, October 2002. (Cited on page 64)

[CDT03] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer: *A multi-user virtual machine*. In *Proceedings of the USENIX Annual Technical Conference*, pages 7–20, San Antonio, Texas, USA, June 2003. USENIX Association Berkeley, CA, USA. (Cited on pages 74 and 87)

[CGG+09] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir: *Toward exascale resilience*. International Journal of High Performance Computing Applications, 23(4):374–388, November 2009. (Cited on page 287)

[CGR11] Nuno Carvalho, Rachid Guerraoui, and Luis Rodrigues: *Revised Hands-On Sections*. Springer, Berlin Heidelberg, March 2011. http://distributedprogramming.net/docs/handson-2011.pdf, Accessed 10/12. (Cited on page 127)

[CHS+03] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce: *A resource management interface for the java platform*. Technical report, Sun Microsystems, Mountain View, 2003. http://research.sun.com/techrep/2003/abstract-124.htm, Accessed: 2/2011. (Cited on page 82)

[CKB+07] SungJin Choi, HongSoo Kim, EunJoung Byun, MaengSoon Baik, SungSuk Kim, ChanYeol Park, and ChongSun Hwang: *Characterizing and classifying desktop grids*. In *Proceedings of the 7th IEEE International Symposium on*

*Cluster Computing and the Grid (CCGRID'07)*, pages 743–748, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society. (Cited on page 53)

[CL82] Shimon Cohen and Daniel Lehmann: *Dynamic systems and their distributed termination*. In *Proceedings of the 1st ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC '82)*, pages 29–33, Ottawa, Ontario, Canada, August 1982. ACM New York, NY, USA. (Cited on page 226)

[CL85] K. Mani Chandy and Leslie Lamport: *Distributed snapshots: determining global states of distributed systems*. ACM Transactions on Computer Systems, 3(1):63–75, February 1985. (Cited on page 58)

[CL96] J. C. Culberson and F. Luo: *Exploring the k-colorable landscape with iterated greedy*. In *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 245–284. American Mathematical Society, Rhode Island, 1996. (Cited on page 87)

[CM02] Peter Cappello and Dimitrios Mourloukos: *Cx: A scalable, robust network for parallel computing*. Scientific Programming, 10(2):159–171, April 2002. (Cited on pages 53, 67, and 290)

[CME04] Andrew A. Chien, Shawn Marlin, and Stephen T. Elbert: *Resource management in the entropia system*. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz (editors): *Grid resource management*, pages 431–450. Kluwer Academic Publishers, Norwell, MA, USA, 2004. (Cited on page 53)

[CNJ+07] Xingchen Chu, Krishna Nadiminti, Chao Jin, Srikumar Venugopal, and Rajkumar Buyya: *Aneka: Next-generation enterprise grid platform for e-science and e-business applications*. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing*, pages 151–159, Bangalore, India, December 2007. IEEE Computer Society Washington, DC, USA. (Cited on page 59)

[coh] *Cohesion Platform website*. `http://www.cohesion.de`, Accessed: 7/2011. (Cited on pages 24 and 43)

[Cona] Conseil Européen pour la Recherche Nucléaire: *EGEE-Enabling Grids for E-sciencE*. `http://www.eu-egee.org/`, Accessed: 2/2011. (Cited on page 34)

[Conb] Conseil Européen pour la Recherche Nucléaire: *gLite - Lightweight Middleware for Grid Computing*. `http://glite.cern.ch/`, Accessed: 2/2011. (Cited on page 34)

[conc] DEGISCO consortium: *Website of the* DEGISCO *project*. `http://degisco.eu/`, Accessed: 5/2011. (Cited on page 36)

[Coo71] S. A. Cook: *The complexity of theorem proving procedures*. In *Proceedings of the 3rd Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, USA, May 1971. ACM. (Cited on pages 29, 48, and 245)

[CRG11] Christian Cachin, Luis Rodrigues, and Rachid Guerraoui: *Reliable and Secure Distributed Programming*. Springer Berlin / Heidelberg, 2nd edition, 2011. (Cited on pages 49, 115, 128, 129, 149, and 297)

[CSKT08] Fernando Costa, Luis Silva, Ian Kelley, and Ian Taylor: *Peer-to-peer techniques for data distribution in desktop grid computing platforms*. In Marco Danelutto, Paraskevi Fragopoulou, and Vladimir Getov (editors): *Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments - Making Grids Work*, pages 377–391, Heraklion, Crete, Greece, June 2008. Springer Berlin / Heidelberg. (Cited on pages 53 and 60)

[CT96] Tushar Deepak Chandra and Sam Toueg: *Unreliable failure detectors for reliable distributed systems*. Journal of the ACM, 43(2):225–267, 1996. (Cited on page 129)

[CW03] Wahid Chrabakh and Rich Wolski: *GridSAT: A Chaff-based distributed SAT solver for the grid*. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC'03)*, pages 37–49, Phoenix, Arizona, USA, November 2003. ACM New York, NY, USA. (Cited on page 252)

[CW06a] Wahid Chrabakh and Rich Wolski: *Gridsat: a system for solving satisfiability problems using a computational grid*. Parallel Computing, 32(9):660–687, 2006. (Cited on pages 265 and 280)

[CW06b] Wahid Chrabakh and Rich Wolski: *Gridsat: Design and implementation of a computational grid application*. Journal of Grid Computing, 5(2):177–193, 2006. (Cited on page 280)

[CZ85] David R. Cheriton and Willy Zwaenepoel: *Distributed process groups in the v kernel*. ACM Transactions on Computer Systems, 3(2):77–107, May 1985. (Cited on page 123)

[D-G] D-Grid GmbH: *D-Grid Initiative*. `http://www.d-grid.de/index.php?id=1&L=1`, Accessed: 2/2011. (Cited on page 34)

[DBM$^+$11] Jack Dongarra, Peter H. Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean Claude Andre, David Barkai, Jean Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara M. Chapman, Xuebin Chi, Alok N. Choudhary, Sudip S. Dosanjh, Thom H. Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert J. Harrison, Mark Hereld, Michael A. Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David E. Keyes, Bill Kramer, Jesús Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi

Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Müller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E. Papka, Daniel A. Reed, Mitsuhisa Sato, Edward Seidel, John Shalf, David Skinner, Marc Snir, Thomas L. Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne E. Trefethen, Mateo Valero, Aad van der Steen, Jeffrey S. Vetter, Peg Williams, Robert Wisniewski, and Katherine A. Yelick: *The international exascale software project roadmap*. International Journal of High Performance Computer Applications, 25(1):3–60, 2011. (Cited on pages 36, 286, and 287)

[DeB05] Erik P. DeBenedictis: *Reversible logic for supercomputing*. In *Proceedings of the 2nd Conference on Computing frontiers (Computing Frontiers '05)*, pages 391–402, Ischia, Italy, May 2005. ACM New York, NY, USA. (Cited on page 33)

[Deu06] Deutsche Forschungsgemeinschaft: *Parallele Verfahren und Systeme für das SAT-Solving*, 2006. http://gepris.dfg.de/gepris/OCTOPUS/?module=gepris&task=showDetail&context=projekt&id=24060795, Accessed: 6/2011. (Cited on page 4)

[DFvG86] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren: *Derivation of a termination detection algorithm for distributed computations*. In *Proceedings of the NATO Advanced Study Institute on Control flow and data flow: concepts of distributed programming*, pages 507–512, Marktoberdorf, Germany, December 1986. Springer New York, NY, USA. (Cited on page 144)

[Dim93] *Satisfiability - suggested format.*, May 1993. http://www.satlib.org/Benchmarks/SAT/satformat.ps, Accessed: 3/2011. (Cited on pages 255, 267, and 269)

[DIR97] D. M. Dhamdhere, Sridhar R. Iyer, and E. Kishore Kumar Reddy: *Distributed termination detection for dynamic systems*. Parallel Computing, 22(14):2025–2045, 1997. (Cited on page 225)

[Dis] Distributed Computing Technologies, Inc.: *distributed.net*. http://www.distributed.net/, Accessed: 5/2011. (Cited on pages 24, 36, and 43)

[DJW+03] Wen Dou, Yan Jia, Huai Ming Wang, Wen Qiang Song, and Peng Zou: *A p2p approach for global computing*. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*, pages 248–253, Nice, France, April 2003. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 63)

[DLL62] M. Davis, G. Logemann, and D. Loveland: *A machine program for theorem-proving*. Communications of the ACM, 5(7):394–397, 1962. (Cited on pages 245 and 247)

[DP60] M. Davis and H. Putnam: *A computing procedure for quantification theory*. Journal of the ACM, 7(3):201–215, 1960. (Cited on pages 245 and 247)

[DS80] Edsger W. Dijkstra and C. S. Scholten: *Termination detection for diffusing computations*. Information Processing Letters, 11(1):1–4, August 1980. (Cited on pages 225 and 226)

[dSCB03] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro: *Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids*. In Harald Kosch, László Böszörményi, and Hermann Hellwagner (editors): *Proceedings of the 9th International Euro-Par Conference on Parallel and Distributed Computing (Euro-Par '03)*, volume 2790 of *Lecture Notes in Computer Science*, pages 169–180, Klagenfurt, Austria, August 2003. Springer Berlin / Heidelberg. (Cited on page 62)

[DTE07] Ronald F. DeMara, Yili Tseng, and Abdel Ejnioui: *Tiered algorithm for distributed process quiescence and termination detection*. IEEE Transactions on Parallel and Distributed Systems, 18(11):1529–1538, 2007. (Cited on page 226)

[DWH09] Sheng Di, Cho Li Wang, and Dexter H. Hu: *Gossip-based dynamic load balancing in a self-organized desktop grid*. In *Proceedings of the 10th International Conference on High-Performance Computing in Asia-Pacific Region (HPC Asia '09)*, pages 85–92, Kaohsiung, Taiwan, March 2009. (Cited on pages 53 and 63)

[EB05] Niklas Eén and Armin Biere: *Effective preprocessing in sat through variable and clause elimination*. In Fahiem Bacchus and Toby Walsh (editors): *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, St. Andrews, Scottland, June 2005. Springer Berlin / Heidelberg. (Cited on page 268)

[Ebn96] *Information technology - syntactic metalanguage - extended bnf*, December 1996. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`. (Cited on page 219)

[Ecl] Eclipse Foundation, Inc.: *Eclipse - An Open Development Platform*. `http://www.eclipse.org`, Accessed: 7/2011. (Cited on pages 45 and 100)

[ELvD$^+$96] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne: *A worldwide flock of condors: load sharing among workstation clusters*. Future Generation Computing Systems, 12(1):53–65, May 1996. (Cited on pages 53 and 62)

[ES03] Niklas Eén and Niklas Sörensson: *An extensible SAT-solver*. In Enrico Giunchiglia and Armando Tacchella (editors): *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer. (Cited on pages 264 and 272)

[FDH04] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna: *Parallel multithreaded satisfiability solver: Design and implementation*. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC '04)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 75–90, London, U.K., September 2004. Elsevier B.V. (Cited on page 279)

[fel] *Apache Felix Project Website*. http://felix.apache.org/site/index.html, Accessed: 3/2011. (Cited on page 87)

[FFM06] Thomas Fischer, Stephan Fudeus, and Peter Merz: *A middleware for job distribution in peer-to-peer networks*. In Bo Kagström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski (editors): *Proceeedings of the 8th International Workshop on Parallel Computing (Para '07): State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1147–1157, Umea, Sweden, June 2006. Springer Berlin / Heidelberg. (Cited on pages 53 and 66)

[FHC08] Gilles Fedak, Haiwu He, and Franck Cappello: *Bitdew: a programmable environment for large-scale data management and distribution*. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '08)*, pages 45:1–45:12, Austin, Texas, November 2008. IEEE Press Piscataway, NJ, USA. (Cited on pages 53 and 60)

[FHL+08] Gilles Fedak, Haiwu He, Oleg Lodygensky, Zoltan Balaton, Zoltan Farkas, Gabor Gombas, Peter Kacsuk, Robert Lovas, Attila Csaba Marosi, Ian Kelley, Ian Taylor, Gabor Terstyanszky, Tamas Kiss, Miguel Cardenas-Montes, Ad Emmen, and Filipe Araujo: *Edges: A bridge between desktop grids and service grids*. In *Proceedings of the 3rd China Grid Annual Conference (China Grid '08)*, pages 3–9, Dunhuang, Gansu, P.R.China, August 2008. IEEE Computer Society. (Cited on pages 37, 53, and 67)

[fi] *Fast Infoset Project*. http://fi.dev.java.net/, Accessed: 3/2011. (Cited on pages 114 and 157)

[FJL+97] Sally Floyd, Van Jacobson, Ching Gung Liu, Steven McCanne, and Lixia Zhang: *A reliable multicast framework for light-weight sessions and application level framing*. IEEE/ACM Transactions on Networking, 5(6):784–803, 1997. (Cited on page 149)

[FKBG10] Z. Farkas, P. Kacsuk, Z. Balaton, and G. Gombás: *Interoperability of boinc and egee*. Future Generation Computer Systems, 26(8):1092–1103, October 2010. (Cited on pages 37, 53, and 67)

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke: *The physiology of the grid: An open grid services architecture for distributed systems integration*. Technical report, Globus Alliance, 2002. http://www.globus.org/alliance/publications/papers/ogsa.pdf, Accessed: 7/2011. (Cited on page 34)

[FKS+05] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam J. Treadwell, and J. Von Reich: *The open grid services architecture*, January 2005. http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf, Accessed: 7/2011. (Cited on page 34)

[FKS10] Oliver Flauzac, Michael Krajecki, and Luiz Angelo Steffenel: *Confiit: a middleware for peer-to-peer computing*. The Journal of Supercomputing, 53(1):86–102, 2010. (Cited on pages 53, 54, and 56)

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke: *The anatomy of the grid: Enabling scalable virtual organizations*. International Journal of High Performance Computing Applications, 15(3):200–222, August 2001. (Cited on page 33)

[Fly72] Michael J. Flynn: *Some computer organizations and their effectiveness*. IEEE Transactions on Computers, 21(9):948–960, September 1972. (Cited on page 197)

[Fos02] Ian Foster: *What is the Grid? - a three point checklist*. GRIDtoday, 1(6), July 2002. http://dlib.cs.odu.edu/WhatIsTheGrid.pdf, Accessed: 2/2011. (Cited on page 34)

[Fos06] Ian Foster: *Globus toolkit version 4: Software for service-oriented systems*. Journal of Computer Science and Technology, 21(4):513–520, 2006. (Cited on page 34)

[Fra80] Nissim Francez: *Distributed termination*. ACM Transactions on Programming Languages and Systems, 2(1):42–55, January 1980. (Cited on page 225)

[GA] P. Gomez and P. Aston: *The Grinder V3.0*. http://grinder.sourceforge.net, Accessed: 7/2011. (Cited on page 166)

[GD09] Kiev Gama and Didier Donsez: *Towards dynamic component isolation in a service oriented platform*. In Grace A. Lewis, Iman Poernomo, and Christine Hofmeister (editors): *Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE'09)*, volume 5582 of *Lecture Notes in Computer Science*, pages 104–120, East Stroudsburg, PA, USA, June 2009. Springer Berlin / Heidelberg. (Cited on page 95)

[GD10] Kiev Gama and Didier Donsez: *A self-healing component sandbox for untrustworthy third party code execution*. In Lars Grunske, Ralf Reussner, and Frantisek Plasil (editors): *Proceedings of the 13th International Symposium on Component-Based Software Engineering (CBSE'10)*, volume 6092 of *Lecture Notes in Computer Science*, pages 130–149, Prague, Czech Republic, June 2010. Springer Berlin / Heidelberg. (Cited on page 95)

[Gel85] David Gelernter: *Generative communication in linda*. ACM Transactions on Programming Languages and Systems, 7(1):80–112, January 1985. (Cited on page 63)

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar: *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003. (Cited on pages 38, 41, 197, 205, and 252)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns*. Addison-Wesley Professional, January 1995. (Cited on page 142)

[Gim] *Great Internet Mersenne Prime Search (GIMPS)*. http://www.mersenne.org/, Accessed: 7/2011. (Cited on page 36)

[GJR09] Stephane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka: *Fault management in p2p-mpi*. International Journal of Parallel Programming, 37(5):433–461, August 2009. (Cited on page 58)

[GKM01] Ayalvadi Ganesh, Anne Marie Kermarrec, and Laurent Massouli'e: *Scamp: Peer-to-peer lightweight membership service for large-scale group communication*. In Jon Crowcroft and Markus Hofmann (editors): *Proceedings of the 3rd International COST264 Workshop: Networked Group Communication (NGC '01)*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55, London, UK, November 2001. Springer Berlin / Heidelberg. (Cited on page 63)

[GKM03] Ayalvadi J. Ganesh, Anne Marie Kermarrec, and Laurent Massoulié: *Peer-to-Peer Membership Management for Gossip-based Protocols*. IEEE Transactions on Computers, 52(2):139–149, 2003. (Cited on page 141)

[GL02] Seth Gilbert and Nancy Lynch: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News, 33(2):51–59, June 2002, ISSN 0163-5700. http://doi.acm.org/10.1145/564585.564601. (Cited on page 133)

[GR07] Stephane Genaud and Choopan Rattanapoka: *P2p-mpi: A peer-to-peer framework for robust execution of message passing parallel programs on grids*. Journal of Grid Computing, 5(1):27–42, 2007. (Cited on pages 53, 58, and 61)

[GS01] Carla P. Gomes and Bart Selman: *Algorithm portfolios*. Artificial Intelligence, 126(1-2):43–62, 2001. (Cited on page 253)

[GSC97] Carla Gomes, Bart Selman, and Nuno Crato: *Heavy-tailed distributions in combinatorial search*. In Gert Smolka (editor): *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135, Linz, Austria, October 1997. Springer Berlin / Heidelberg. (Cited on page 249)

[GTFC08] Nicolas Geoffray, Gaël Thomas, Bertil Folliot, and Charles Clément: *Towards a new isolation abstraction for osgi*. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES '08)*, pages 41–45, Glasgow, Scotland, UK, April 2008. ACM New York, NY, USA. (Cited on page 95)

[GTL+10] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot: *VMKit: a substrate for managed runtime environments*. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'10)*, volume 45 of *ACM SIGPLAN Notices*, pages 51–62, Pittsburgh, PA, USA, March 2010. ACM New York, NY, USA. (Cited on page 95)

[GTM+09] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot: *I-JVM: a Java virtual machine for component isolation in OSGi*. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks (DSN'09)*, pages 544–553, Estoril, Lisbon, Portugal, June 2009. IEEE Computer Society Washington, DC, USA. (Cited on page 95)

[Gun93] N. J Gunther: *A simple capacity model for massively parallel transaction systems*. In *Proceedings of the Computer Measurement Group Conference (CMG'93)*, pages 1035–1044, San Diego, California, December 1993. Computer Measurement Group, Turnersville, NJ, USA. (Cited on page 295)

[Gus88] John L. Gustafson: *Reevaluating amdahl's law*. Communications of the ACM, 31(5):532–533, May 1988. (Cited on page 294)

[GY95] G. Goldszmidt and Y. Yemini: *Distributed management by delegation*. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS '95)*, pages 333–340, Vancouver, British Columbia, Canada, May 1995. IEEE Computer Society Washington, DC, USA. (Cited on page 104)

[HB08] Jarle Hulaas and Walter Binder: *Program transformations for light-weight cpu accounting and control in the java virtual machine*. Higher-Order and Symbolic Computation, 21(1–2):119–146, June 2008. (Cited on page 74)

[HD03] Emir Halepovic and Ralph Deters: *The costs of using jxta*. In Nahid Shahmehri, Ross Lee Graham, and Germano Caronni (editors): *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P'03)*, pages 160–167, Linköping, Sweden, September 2003. IEEE Computer Society Washington, DC, USA. (Cited on page 181)

[HJN06] Antti E. J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä: *A distribution method for solving sat in grids*. In Armin Biere and Carla P. Gomes (editors): *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 430–435, Seattle, WA, USA, August 2006. Springer Berlin / Heidelberg. (Cited on pages 252 and 280)

[HJN08] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä: *Incorporating learning in grid-based randomized SAT solving*. In Paolo Traverso Danail Dochev, Marco Pistore (editor): *Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA'08)*, volume 4183 of *Lecture Notes in Computer Science*, pages

247–261, Varna, Bulgaria, September 2008. Springer Berlin / Heidelberg. (Cited on page 265)

[HJS09a] Youssef Hamadi, Said Jabbour, and Lakhdar Sais: *Control-based clause sharing in parallel sat solving*. In Craig Boutilier (editor): *Proceedings of the 21st international joint conference on Artifical intelligence (IJCAI'09)*, pages 499–504, Pasadena, California, USA, July 2009. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. (Cited on pages 29, 48, 254, and 266)

[HJS09b] Youssef Hamadi, Said Jabbour, and Lakhdar Sais: *Manysat: a parallel sat solver*. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 6:245–262, 2009. (Cited on pages 253, 254, 265, and 279)

[HMM$^+$10] Christian Hübsch, Christoph P. Mayer, Sebastian Mies, Roland Bless, Oliver P. Waldhorst, and Martina Zitterbart: *Reconnecting the internet with ariba: self-organizing provisioning of end-to-end connectivity in heterogeneous networks*. ACM SIGCOMM - Computer Communication Review, 40(1):131–132, January 2010. (Cited on page 180)

[HMW10] Christian Hübsch, Christoph P. Mayer, and Oliver P. Waldhorst: *On runtime adaptation of application-layer multicast protocol parameters*. In Finn Arve Aagesen and Svein J. Knapskog (editors): *Proceedings of the 16th EUNICE/IFIP WG 6.6 conference on Networked services and applications: engineering, control and management (EUNICE'10)*, volume 6164 of *Lecture Notes in Computer Science*, pages 226–235, Trondheim, Norway, June 2010. Springer Berlin / Heidelberg. (Cited on page 180)

[HN] Sepp Hartung and Rolf Niedermeier: *Graph Coloring Program*. `http://theinf1.informatik.uni-jena.de/inc_cluster/README`, Accessed: 3/2011. (Cited on page 88)

[Hua07] Jinbo Huang: *The effect of restarts on the efficiency of clause learning*. In Manuela M. Veloso (editor): *Proceedings of the 20th international joint conference on Artifical intelligence (IJCAI'07)*, pages 2318–2323, Hyderabad, India, January 2007. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. (Cited on page 249)

[HV95] J. N. Hooker and V. Vinay: *Branching rules for satisfiability*. Journal of Automated Reasoning, 15(3):359–383, 1995. (Cited on page 248)

[Hyv09] Antti E.J. Hyvärinen: *Approaches to Grid-Based SAT Solving*. Licentiate's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2009. (Cited on pages 253 and 280)

[IBM] IBM Corporation: *IBM Tivoli Software*. `http://www-01.ibm.com/software/tivoli/`, Accessed: 7/2011. (Cited on page 104)

[ies] *The International Exascale Software Project website*. http://www.exascale.org, Accessed: 3/2011. (Cited on page 286)

[ios] *i-OSGi Project Website*. http://code.google.com/p/i-osgi/, Accessed: 3/2011. (Cited on pages 24, 43, and 87)

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule: *Blocked clause elimination*. In Javier Esparza and Rupak Majumdar (editors): *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144, Paphos, Cyprus, March 2010. Springer Berlin / Heidelberg. (Cited on page 268)

[jiv] *Ignite Realtime - A Jive Software Community*. http://www.igniterealtime.org, Accessed: 7/2011. (Cited on page 114)

[JLU05] Bernard Jurkowiak, Chu Min Li, and Gil Utard: *A parallelization scheme based on work stealing for a class of sat solvers*. Journal of Automated Reasoning, 34(1):73–101, 2005. (Cited on pages 252 and 279)

[jma] *jManage - Web and Command Line Based JMX Client*. http://www.jmanage.org, Accessed: 7/2011. (Cited on page 104)

[KCCF03] Barbara Kreaseck, Larry Carter, Henri Casanova, and Jeanne Ferrante: *Autonomous protocols for bandwidth-centric scheduling of independent-task applications*. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*, pages 26–35, Nice, France, April 2003. IEEE Computer Society Washington, DC, USA. (Cited on page 64)

[KDH$^+$05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy: *Introduction to the cell multiprocessor*. IBM Journal of Research and Development, 49(4–5):589–604, July 2005. (Cited on page 285)

[KF98] Carl Kesselman and Ian Foster: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998. (Cited on page 33)

[KHW03] Heather Kreger, Ward Harold, and Leigh Williamson: *Java™ and JMX: Building Manageable Systems*. Addison-Wesley Professional, January 2003. (Cited on page 257)

[KKF$^+$09] Peter Kacsuk, Jozsef Kovacs, Zoltan Farkas, Attila Marosi, Gabor Gombas, and Zoltan Balaton: *Sztaki desktop grid (szdg): A flexible and scalable desktop grid system*. Journal of Grid Computing, 7(4):439–461, 2009. (Cited on page 53)

[KKM$^+$07] Jik Soo Kim, Peter J. Keleher, Michael A. Marsh, Bobby Bhattacharjee, and Alan Sussman: *Using content-addressable networks for load balancing in* *desktop grids*. In *Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC '07)*, pages 189–198, Monterey, California, USA, June 2007. ACM New York, NY, USA. (Cited on page 66)

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin: *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web*. In *Proceedings of the 29th annual ACM symposium on Theory of Computing (STOC '97)*, pages 654–663, El Paso, Texas, USA, May 1997. ACM New York, NY, USA. (Cited on page 145)

[KNM+07] Jik Soo Kim, Beomseok Nam, Michael A. Marsh, Peter J. Keleher, Bobby Bhattacharjee, Derek Richardson, Dennis Wellnitz, and Alan Sussman: *Creating a robust desktop grid using peer-to-peer services*. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, pages 1–7, Long Beach, California, USA, March 2007. IEEE Computer Society. (Cited on pages 53 and 66)

[Knu08] Donald E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions (Art of Computer Programming)*. Addison-Wesley Professional, 1st edition, 2008. (Cited on page 56)

[KPS90] Z. M. Kedem, K. V. Palem, and P. G. Spirakis: *Efficient robust parallel computations*. In *Proceedings of the 22nd Annual ACM symposium on Theory of Computing (STOC '90)*, pages 138–148, Baltimore, MD, USA, May 1990. ACM New York, NY, USA. (Cited on page 56)

[KRS02] Wayne Kelly, Paul Roe, and Jiro Sumitomo: *G2: A grid middleware for cycle donation using .net*. In Hamid R. Arabnia (editor): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '02)*, volume 2, pages 699–705, Las Vegas, Nevada, USA, June 2002. CSREA Press. (Cited on page 66)

[KS92] Henry A. Kautz and Bart Selman: *Planning as satisfiability*. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, Vienna, Austria, August 1992. John Wiley and Sons, Chichester. (Cited on pages 29, 48, and 245)

[KSS05] Yoshio Tanaka Kazuyuki Shudo and Satoshi Sekiguchi: *P3: P2P-based middleware enabling transfer and aggregation of computational resources*. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05) - International Workshop on Global and Peer-to-Peer Computing*, pages 259–265, Cardiff, UK, May 2005. IEEE Computer Society. (Cited on pages 53 and 61)

[KTB+04] Derrick Kondo, Michela Taufer, Charles L. Brooks, Henri Casanova, and Andrew A. Chien: *Characterizing and evaluating desktop grids: An empirical* *study*. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 26–35, Sante Fe, New Mexico, April 2004. IEEE Computer Society. (Cited on pages 23, 37, 62, and 170)

[Kuh55] H. W. Kuhn: *The hungarian method for the assignment problem*. Naval Research Logistics Quarterly, 2(1-2):83–97, 1955. (Cited on page 87)

[LAGS09] Troy Leblanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok: *Volpexmpi: An mpi library for execution of parallel applications on volatile nodes*. In Matti Ropo, Jan Westerholm, and Jack Dongarra (editors): *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI '09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 124–133, Espoo, Finland, September 2009. Springer Berlin / Heidelberg. (Cited on pages 53 and 58)

[Lai86] Ten Hwang Lai: *Termination detection for dynamically distributed systems with non-first-in-first-out communication*. Journal of Parallel and Distributed Computing, 3(4):577–599, 1986. (Cited on page 226)

[Lam98] Leslie Lamport: *The part-time parliament*. ACM Transactions on Computer Systems, 16(2):133–169, May 1998, ISSN 0734-2071. http://doi.acm.org/10.1145/279227.279229. (Cited on page 28)

[LBP10] Daniel Le Berre and Anne Parrain: *The sat4j library, release 2.2, system description*. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 7:59–64, 2010. (Cited on page 88)

[LBRV05] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal: *Alchemi: A .net-based enterprise grid computing system*. In Hamid R. Arabnia and Rose Joshua (editors): *Proceedings of the International Conference on Internet Computing (ICOMP '05)*, pages 269–278, Las Vegas, Nevada, USA, June 2005. CSREA Press. (Cited on page 59)

[LCBF06] Aliandro Lima, Walfredo Cirne, Francisco Vilar Brasileiro, and Daniel Fireman: *A case for event-driven distributed objects*. In Robert Meersman and Zahir Tari (editors): *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1705–1721, Montpellier, France, October 2006. Springer Berlin / Heidelberg. (Cited on page 182)

[LCN90] Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld: *Process groups and group communications: Classifications and requirements*. Computer, 23(2):56–66, February 1990. (Cited on page 123)

[LG03] Mary Dageforde Li Gong, Gary Ellison: *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley Professional, 2003. (Cited on page 100)

[Lia99] Sheng Liang: *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley Longman, 1999. (Cited on page 81)

[Liu97] Ching Gung Liu: *Error recovery in scalable reliable multicast*. Ph.d. thesis, University of Southern California, Los Angeles, CA, USA, 1997. (Cited on page 149)

[LK00] Spyros Lalis and Alexandros Karipidis: *Jaws: An open market-based framework for distributed computing over the internet*. In Rajkumar Buyya and Mark Baker (editors): *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID '00)*, volume 1971 of *Lecture Notes in Computer Science*, pages 36–46, Bangalore, India, December 2000. Springer London, UK. (Cited on pages 53 and 67)

[LLM88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka: *Condor - a hunter of idle workstations*. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104–111, San Jose, California, USA, June 1988. IEEE Computer Society Press Los Alamitos, CA, USA. (Cited on page 62)

[Low01] Juval Lowy: *COM and .NET Component Services*. Nutshell Books. O'Reilly, 1st edition, September 2001. (Cited on page 95)

[LPP07] Wei Cherng Liao, Fragkiskos Papadopoulos, and Konstantinos Psounis: *Performance analysis of BitTorrent-like systems with heterogeneous users*. Performance Evaluation, 64(9-12):876–891, 2007. (Cited on pages 60 and 270)

[LS97] Emil C. Lupu and Morris Sloman: *Towards a role-based framework for distributed systems management*. Journal of Network and Systems Management, 5(1):5–30, 1997. (Cited on page 100)

[LS99] Michael Litzkow and Marvin Solomon: *Supporting checkpointing and process migration outside the unix kernel*. In *Mobility*, pages 154–162. ACM Press / Addison-Wesley Publishing, New York, NY, USA, 1999. (Cited on page 62)

[LSB07] M. Lewis, T. Schubert, and B. Becker: *Multithreaded SAT solving*. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASP-DAC '07)*, pages 926–931, Yokohama, Japan, January 2007. IEEE Computer Society Washington, DC, USA. (Cited on page 279)

[LSG10] Troy P. Leblanc, Jaspal Subhlok, and Edgar Gabriel: *A high-level interpreted mpi library for parallel computing in volunteer environments*. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*, pages 673–678, Melbourne, Victoria, Australia, May 2010. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 58)

[Luc98] Matthew Thomas Lucas: *Efficient data distribution in large-scale multicast networks*. Ph.d. thesis, Pennsylvania State University, 1998. (Cited on page 149)

[Lyn96] Nancy A. Lynch: *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1996. (Cited on page 128)

[LZZ+05] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao: *Cluster computing on the fly: P2p scheduling of idle cycles in the internet*. In Geoffrey Voelker and Scott Shenker (editors): *Peer-to-Peer Systems III*, volume 3279 of *Lecture Notes in Computer Science*, pages 227–236, La Jolla, CA, USA, February 2005. Springer Berlin / Heidelberg. (Cited on pages 53 and 65)

[mas] *Open Mashup Alliance (OMA) Website*. http://www.openmashup.org, Accessed: 3/2011. (Cited on page 71)

[Mat87] Friedemann Mattern: *Algorithms for distributed termination detection*. Distributed Computing, 2(3):161–175, 1987. (Cited on page 225)

[Mat89] Friedemann Mattern: *Global quiescence detection based on credit distribution and recovery*. Information Processing Letters, 30(4):195–200, 1989. (Cited on pages 29, 48, and 226)

[MC82] Jayadev Misra and K. M. Chandy: *Termination detection of diffusing computations in communicating sequential processes*. ACM Transactions on Programming Languages and Systems, 4(1):37–43, 1982. (Cited on page 226)

[MC98] Jeff Matocha and Tracy Camp: *A taxonomy of distributed termination detection algorithms*. Journal of Systems and Software, 43(3):207–221, November 1998. (Cited on page 226)

[MCC04] Matthew L. Massie, Brent N. Chun, and David E. Culler: *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*. Parallel Computing, 30(7):817–840, 2004. (Cited on page 192)

[MCT+09] Carlo Mastroianni, Pasquale Cozza, Domenico Talia, Ian Kelley, and Ian Taylor: *A scalable super-peer approach for public scientific computation*. Future Generation Computer Systems, 25(3):213–223, March 2009. (Cited on pages 53 and 62)

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong: *Tag: a tiny aggregation service for ad-hoc sensor networks*. In *Proceedings of the 5th Symposium on Operating systems Design and Implementation (OSDI '02)*, pages 131–146, Boston, Massachusetts, USA, December 2002. ACM Press New York, NY, USA. (Cited on page 193)

[Mic] Microsoft Corporation: *Windows Automated Installation Kit (AIK) for Windows 7*. http://www.microsoft.com, Accessed: 3/2011. (Cited on page 87)

[Mil] R. A. Milowski: *Xeerkat - A P2P computing framework over XMPP*. http://code.google.com/p/xeerkat/, Accessed: 7/2011. (Cited on page 182)

[Mil05] R. A. Milowski: *Computing for the mathematical sciences with xml web services and p2p*. In *XML 2005*, Atlanta, Georgia, U.S.A., November 2005. (Cited on page 182)

[min] *MINA*. http://mina.apache.org/, Accessed: 7/2011. (Cited on page 138)

[Mit05] David G. Mitchell: *A SAT solver primer*. EATCS Bulletin (The Logic in Computer Science Column), 85:112–133, February 2005. (Cited on page 248)

[MJB04] Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu: *Robust aggregation protocols for large-scale overlay networks*. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04)*, pages 19–28, Florence, Italy, July 2004. IEEE Computer Society Washington, DC, USA. (Cited on page 192)

[MK05] Richard Mason and Wayne Kelly: *G2-p2p: a fully decentralised fault-tolerant cycle-stealing framework*. In *Proceedings of the Australasian workshop on Grid computing and e-research*, volume 44 of *ACSW Frontiers '05*, pages 33–39, Darlinghurst, Australia, Australia, 2005. Australian Computer Society. (Cited on pages 53 and 65)

[MM00] F. Massacci and L. Marraro: *Logical cryptanalysis as a SAT problem*. Journal of Automated Reasoning, 24(1-2):165–203, February 2000. (Cited on pages 29, 48, and 245)

[MMB03] Alberto Montresor, Hein Meling, and Özalp Babaoglu: *Messor: Load-balancing through a swarm of autonomous agents*. In Gianluca Moro and Manolis Koubarakis (editors): *Agents and Peer-to-Peer Computing*, volume 2530 of *Lecture Notes in Computer Science*, pages 125–137, Melbourne, Australia, July 2003. Springer Berlin / Heidelberg. (Cited on pages 53 and 65)

[MMZ$^+$01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik: *Chaff: Engineering an efficient SAT solver*. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, Las Vegas, Nevada, USA, June 2001. ACM New York, NY, USA. (Cited on pages 248 and 252)

[Moo75] G.E. Moore: *Progress in digital integrated electronics*. In *Proceedings of the IEEE International Electron Devices Meeting (IEDM '75)*, pages 11–13, Washington, DC, USA, December 1975. IEEE Computer Society Washington, DC, USA. (Cited on pages 30, 285, and 289)

[MSS96] J. P. Marques-Silva and K. A. Sakallah: *Grasp - a new search algorithm for satisfiability*. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '96)*, pages 220–227, San Jose, California, USA, November 1996. IEEE Computer Society. (Cited on page 248)

[Mut92] Matt W. Mutka: *Estimating capacity for sharing in a privately owned workstation environment*. IEEE Transactions on Software Engineering, 18(4):319–328, April 1992. (Cited on pages 23 and 35)

[MVP04] N. Mittal, S. Venkatesan, and S. Peri: *Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies*. In Rachid Guerraoui (editor): *Proceedings of the 18th International Symposium on Distributed Computing (DISC '04) - Workshop on Distributed Algorithms (WDAG)*, volume 3274 of *Lecture Notes in Computer Science*, pages 290–304, Amsterdam, The Netherlands, October 2004. Springer Berlin / Heidelberg. (Cited on page 225)

[MWSP08] Peter Merz, Steffen Wolf, Dennis Schwerdel, and Matthias Priebe: *A self-organizing super-peer overlay with a chord core for desktop grids*. In Karin Anna Hummel and James P. G. Sterbenz (editors): *Proceedings of the 3rd International Workshop on Self-Organizing Systems (IWSOS '08)*, volume 5343 of *Lecture Notes in Computer Science*, pages 23–34, Vienna, Austria, December 2008. Springer Berlin / Heidelberg. (Cited on page 180)

[NC05] Michael O. Neary and Peter R. Cappello: *Advanced eager scheduling for java-based adaptive parallel computing*. Concurrency - Practice and Experience, 17(7–8):797–819, 2005. (Cited on pages 53, 54, and 57)

[NH09] Lei Ni and Aaron Harwood: *P2p-tuple: Towards a robust volunteer computing platform*. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pages 217–223, Higashi Hiroshima, Japan, December 2009. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 63)

[Nie98] Jakob Nielsen: *Nielsen's law of internet bandwidth*, April 1998. http://www.useit.com/alertbox/980405.html, Accessed: 7/2011. (Cited on page 289)

[nis11] *A NIST Definition of Cloud Computing*, January 2011. http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf, DRAFT, Accessed: 7/2011. (Cited on page 286)

[NKVB08] Ulrich Norbisrath, Keio Kraaner, Eero Vainikko, and Oleg Batrasev: *Friend-to-friend computing - instant messaging based spontaneous desktop grid*. In Abdelhamid Mellouk, Jun Bi, Guadalupe Ortiz, Dickson K. W. Chiu, and Manuela Popescu (editors): *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pages 245–256, Athens, Greece, June 2008. IEEE Computer Society Los Alamitos, CA, USA. (Cited on page 182)

[OOB00] Öznur Özkasap and Kenneth P. Birman: *Throughput stability of reliable multicast protocols*. In Tatyana M. Yakhno (editor): *Proceedings of the 1st International Conference on Advances in Information Systems (ADVIS '00)*, volume 1909 of *Lecture Notes in Computer Science*, pages 159–169, Izmir, Turkey, October 2000. Springer London, UK. (Cited on page 149)

[ORBP07] Angela Orebaugh, Gilbert Ramirez, Josh Burke, and Larry Pesce: *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source*

*Security)*. Syngress Publishing, 800 Hingham Street, Rockland, MA 02370, 2007. (Cited on page 159)

[OSG] OSGi Alliance: *OSGi™- The Dynamic Module System for Java™*. http://www.osgi.org, Accessed: 7/2011. (Cited on page 81)

[osg10] *OSGi service platform release 4 – core 4.3 early draft 1*, April 2010. http://www.osgi.org/download/osgi-core-4.3-early-draft1.pdf, Accessed: 2/2011. (Cited on page 82)

[Pac96] Peter S. Pacheco: *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. (Cited on page 58)

[Pac05] Charles Paclat: *Proposed JMX Technology Features by Expert Group Members*. Presentation at the JavaOne Conference, July 2005. java.net/attachments/lists/jdk/editor/2005-07/17/JavaOne_JMX_BOF.pdf, Accessed: 7/2011. (Cited on page 102)

[Pal06] Krzysztof Palacz: *Application isolation api specification*, June 2006. http://jcp.org/aboutJava/communityprocess/final/jsr121/index.html, Accessed: 2/2011. (Cited on page 82)

[Par72] D. L. Parnas: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12):1053–1058, December 1972. (Cited on page 71)

[PDH00] E. J. Walsh P. D. Hough, M. E. Goldsby: *Algorithm-dependent fault tolerance for distributed computing*. Technical Report SAND2000-8219, Sandia National Laboratories, February 2000. (Cited on page 56)

[Pea] Kirk Pearson: *distributedcomputing.info*. http://distributedcomputing.info, Accessed: 5/2011. (Cited on page 36)

[Pet] Peta Computing Institute, Ltd.: *MDGRAPE-3*. http://www.peta.co.jp/index-en.html, Accessed: 3/2011. (Cited on page 286)

[PHP+03] Pradeep Padala, Cyrus Harrison, Nicholas Pelfort, Erwin Jansen, and Michael P. Frank: *Ocean: the open computation exchange and arbitration network, a market approach to meta computing*. In *Proceedings of the 2nd International Conference on Parallel and Distributed Computing (ISPDC'03)*, pages 185–192, Ljubljana, Slovenia, October 2003. IEEE Computer Society, Washington, DC, USA. (Cited on page 290)

[PHS08] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs: *Vivifying propositional clausal formulae*. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris (editors): *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529, Patras, Greece, July 2008. IOS Press, Amsterdam, The Netherlands. (Cited on page 268)

[PKA+06] Stephen Plaza, Ian Kountanis, Zaher Andraus, Valeria Bertacco, and Trevor Mudge: *Advances and insights into parallel sat solving*. In *Proceedings of the 15th IEEE/ACM International Workshop on Logic and Synthesis (IWLS'06)*, pages 188–194, Vail, Colorado, USA, June 2006. IEEE Computer Society. (Cited on pages 254 and 265)

[Pos81] John Postel: *Transmission Control Protocol*, September 1981. `http://tools.ietf.org/html/rfc793`, Accessed: 7/2011. (Cited on page 130)

[PS97] Rico Piantoni and Constantin Stancescu: *Implementing the swiss exchange trading system*. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 309–313, Seattle, USA, 1997. IEEE Computer Society, Washington, DC, USA. (Cited on page 149)

[Rab89] Michael O. Rabin: *Efficient dispersal of information for security, load balancing, and fault tolerance*. Journal of the ACM, 36(2):335–348, April 1989. (Cited on page 64)

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe: *R-OSGi: distributed applications through software modularization*. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware (Middleware '07)*, pages 1–20, Newport Beach, CA, USA, 2007. Springer, New York, USA. (Cited on page 87)

[Ras] *Rasterbar Software website*. `http://www.rasterbar.com`, Accessed: 3/2010. (Cited on page 271)

[Rat08] Choopan Rattanapoka: *P2P-MPI: A Fault-tolerant Message Passing Interface Implementation for Grids*. Ph.d. thesis, University Louis Pasteur Strasbourg, April 2008. `http://icps.u-strasbg.fr/upload/icps-2008-208.pdf`, Accessed: 7/2011. (Cited on page 181)

[RBV03] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels: *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*. ACM Transactions on Computer Systems, 21(2):164–206, 2003. (Cited on page 192)

[RD01a] Antony Rowstron and Peter Druschel: *Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility*. ACM SIGOPS Operating Systems Review, 35(5):188–201, October 2001. (Cited on pages 59 and 64)

[RD01b] Antony I. T. Rowstron and Peter Druschel: *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*. In Rachid Guerraoui (editor): *Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350, Heidelberg, Germany, November 2001. Springer Berlin / Heidelberg. (Cited on pages 59, 63, 66, and 179)

[RDA09] Jan S. Rellermeyer, Michael Duller, and Gustavo Alonso: *Engineering the cloud from software modules*. In *Proceedings of the International Conference on Software Engineering (ICSE'09), Workshop on Software Engineering Challenges of Cloud Computing*, pages 32–37, Vancouver, BC, Canada, May 2009. IEEE Computer Society, Washington, DC, USA. (Cited on page 71)

[RFH⁺01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker: *A scalable content-addressable network*. ACM SIGCOMM - Computer Communication Review, 31(4):161–172, August 2001. (Cited on pages 65, 66, and 179)

[RFLM06] Yvan Royon, Stéphane Frénot, and Frédéric Le Mouël: *Virtualization of service gateways in multi-provider environments*. In Ian Gorton, George Heineman, Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau (editors): *Component-Based Software Engineering (CBSE '06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 385–392, Vasteras, Sweden, June 2006. Springer Berlin / Heidelberg. (Cited on page 95)

[RJ08] Benjamin Reed and Flavio P. Junqueira: *A simple totally ordered broadcast protocol*. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*, pages 2:1–2:6, Yorktown, New York, USA, 2008. ACM Press New York. (Cited on page 87)

[RLS00] Rajesh Raman, Miron Livny, and Marvin Solomon: *Resource management through multilateral matchmaking*. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC '00)*, pages 290–291, Pittsburgh, Pennsylvania, USA, August 2000. IEEE Computer Society Washington, DC, USA. (Cited on page 62)

[RM06] John Risson and Tim Moorsa: *Survey of research towards robust peer-to-peer networks: Search methods*. Computer Networks, 50(17):3485–3521, 2006. (Cited on pages 27, 47, and 185)

[RMSA06] Arathi Ramani, Igor L. Markov, Karem A. Sakallah, and Fadi A. Aloul: *Breaking instance-independent symmetries in exact graph coloring*. Journal of Artificial Intelligence Research (JAIR), 26(1):289–322, July 2006. (Cited on pages 88 and 90)

[RN00] Ori Regev and Noam Nisan: *The popcorn market. online markets for computational resources*. Decision Support Systems, 28(1–2):177–189, March 2000. (Cited on pages 53 and 67)

[SA04a] P. Saint-Andre: *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*. RFC 3923, October 2004. `http://www.ietf.org/rfc/rfc3923.txt`, Accessed: 7/2011. (Cited on page 114)

[SA04b] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 3920, October 2004. http://www.ietf.org/rfc/rfc3920.txt, Accessed: 7/2011. (Cited on pages 114 and 115)

[SA04c] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. RFC 3921, October 2004. http://www.ietf.org/rfc/rfc3921.txt, Accessed: 7/2011. (Cited on pages 114 and 115)

[SA13] Peter Saint-Andre: *Xep-0045: Multi-user chat*, 2013. http://xmpp.org/extensions/xep-0045.html. (Cited on page 119)

[SAA+04] Ion Stoica, Daniel Adkins, Shelley Adkins, Scott Shenker, and Sonesh Surana: *Internet indirection infrastructure*. IEEE/ACM Transactions on Networks, 12(2):205–218, 2004. (Cited on page 110)

[Sar98] Luis Sarmenta: *Bayanihan: Web-based volunteer computing using java*. In Yoshifumi Masunaga, Takuya Katayama, and Michiharu Tsukamoto (editors): *Proceedings of the International Conference on Worldwide Computing and its Applications (WWCA'98)*, volume 1368 of *Lecture Notes in Computer Science*, pages 444–461, Tsukuba, Japan, March 1998. Springer Berlin / Heidelberg. (Cited on page 55)

[Sar99] Luis F. G. Sarmenta: *An adaptive, fault-tolerant implementation of bsp for java-based volunteer computing systems*. In *Proceedings of the 11th Workshops Held in Conjunction with the 13th International Parallel Processing Symposium (IPPS '99) and 10th Symposium on Parallel and Distributed Processing (SPDP '99)*, pages 763–780, San Juan, Puerto Rico, 1999. Springer London, UK. (Cited on pages 53 and 55)

[Sar01] Luis F. G. Sarmenta: *Volunteer Computing*. Ph.d. thesis, Dept. of Electrical Engineering and Computer Science, MIT, March 2001. (Cited on page 53)

[Sata] *SAT Competition 2007 website*. http://www.satcompetition.org/2007/, Accessed: 3/2010. (Cited on page 267)

[Satb] *SAT Race 2008 website*. http://baldur.iti.uka.de/sat-race-2008/, Accessed: 3/2011. (Cited on page 267)

[satc] *SAT Competition website*. http://www.satcompetition.org, Accessed: 7/2011. (Cited on pages 261 and 272)

[SATd] *SAT Race 2010 website*. http://baldur.iti.uka.de/sat-race-2010/, Accessed: 7/2011. (Cited on page 260)

[Sat09] *SAT Competition 2009 website*, 2009. http://www.satcompetition.org/2007/, Accessed: 2/2011. (Cited on page 253)

[SB07] Sven Schulz and Wolfgang Blochinger: *An integrated approach for managing peer-to-peer desktop grid systems*. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 233–240, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society Washington, DC, USA. (Cited on pages 24, 26, 43, 45, 97, and 100)

[SB10a] Sven Schulz and Wolfgang Blochinger: *Cooperate and compete! a hybrid solving strategy for task-parallel sat solving on peer-to-peer desktop grids*. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2010), Workshop on Parallel Satisfiability Solving (WPSS 2010)*, pages 314–323, Caen, France, 2010. IEEE Computer Society. (Cited on pages 24, 29, 43, 48, 243, and 275)

[SB10b] Sven Schulz and Wolfgang Blochinger: *Parallel sat-solving on peer-to-peer desktop grids*. Journal of Grid Computing, 8(3):443–471, 2010. (Cited on pages 24, 28, 29, 43, 48, 195, and 243)

[SB11] Sven Schulz and Wolfgang Blochinger: *Adjustable module isolation for distributed computing infrastructures*. In *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (GRID '11)*, Lyon, France, September 2011. IEEE Computer Society / ACM Press. In press. (Cited on pages 24, 25, 43, 45, and 69)

[SBB+11] L.A. Steffenel, J C. Boisson, C. Barberot, S. Gerard, E. Henon, C. Jaillet, O. Flauzac, and M. Krajecki: *Deploying a fault-tolerant computing middleware over grid'5000: performance analysis of confiit and its integration with a quantum molecular docking application*. In *Proceedings of the 4th Grid'5000 Spring School*, Reims, France, April 2011. (Cited on pages 53 and 56)

[SBH09] Sven Schulz, Wolfgang Blochinger, and Hannes Hannak: *Capability-aware information aggregation in peer-to-peer grids – methods, architecture, and implementation*. Journal of Grid Computing, 7(2):135–167, 2009. (Cited on pages 24, 27, 43, 47, 143, 183, 185, 191, and 192)

[SBHD08] Sven Schulz, Wolfgang Blochinger, Markus Held, and Clemens Dangelmayr: *COHESION - A microkernel based desktop grid platform for irregular task-parallel applications*. Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications, 24(5):354–370, 2008. (Cited on pages 24, 25, 28, 43, 45, and 48)

[SBP09] Sven Schulz, Wolfgang Blochinger, and Mathias Poths: *A network substrate for peer-to-peer grid computing beyond embarrassingly parallel applications*. In *Proceedings of the WRI International Conference on Communications and Mobile Computing (CMC 2009)*, volume 3, pages 60–68, Kunming, Yunnan, China, January 2009. IEEE Computer Society Washington, DC, USA. (Cited on pages 24, 26, 43, 46, and 105)

[SBP10] Sven Schulz, Wolfgang Blochinger, and Matthias Poths: *Orbweb – a network substrate for peer-to-peer grid computing based on open standards*. Journal of Grid Computing, 8(1):77–107, 2010. (Cited on pages 24, 26, 43, 46, and 105)

[SBSV96] P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli: *Combinational test pattern generation using satisfiability*. IEEE Transactions on Computer-Aided Design, 15(9):1167–1176, 1996. (Cited on page 245)

[SC05] Daniel Steinberg and Stuart Cheshire: *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, 1st edition edition, 2005. (Cited on page 67)

[Sch90] Fred B. Schneider: *Implementing fault-tolerant services using the state machine approach: A tutorial*. ACM Computing Surveys (CSUR), 22(4):299–319, 1990. (Cited on page 58)

[Sch95] Douglas C. Schmidt: *Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching*. In James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*, pages 529–545. ACM Press / Addison-Wesley, New York, NY, USA, 1995. (Cited on page 157)

[Sch01] R. Schollmeier: *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P'01, pages 101–102, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on pages 41 and 113)

[SE10] Bernhard Schott and Ad Emmen: *Degisco: Green methodologies in desktop-grids*. In *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT '10)*, pages 671–676, Wisla, Poland, October 2010. (Cited on page 36)

[SEE$^+$03] O. Smirnova, P. Eerola, T. Ekelöf, M. Ellert, J. R. Hansen, A. Konstantinov, B. Kónya, J. L. Nielsen, F. Ould-Saada, and A. Wäänänen: *The nordugrid architecture and middleware for scientific applications*. In *Proceedings of the 1st International Conference on Computational Science (ICCS '03)*, Lecture Notes in Computer Science, pages 264–273, Melbourne, Australia, 2003. Springer Berlin / Heidelberg. (Cited on page 34)

[Sei] Jean Mark Seigneur: *JXTA Pipe Performance*. `http://bench.jxta.org`, Accessed: 7/2008. (Cited on page 181)

[SH99] Luis F.G. Sarmenta and Satoshi Hirano: *Bayanihan: building and studying web-based volunteer computing systems using java*. Future Generation Computer Systems, 15(5–6):675–686, 1999. (Cited on page 55)

[Shi08] George Shiffler: *Forecast: Pc installed base, worldwide, 2004-2012*, April 2008. `http://www.gartner.com/DisplayDocument?`

ref=g_search&id=644708&subref=simplesearch, Accessed: 7/2011. (Cited on pages 23, 33, and 36)

[SKH95] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson (editors): *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. (Cited on page 205)

[SLB09] Tobias Schubert, Matthew Lewis, and Bernd Becker: *PaMiraXT: Parallel SAT Solving with Threads and Message Passing*. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 6:203–222, 2009. (Cited on page 279)

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan: *Chord: A scalable peer-to-peer lookup service for internet applications*. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pages 149–160, San Diego, CA, USA, August 2001. ACM Press New York, NY, USA. (Cited on pages 145, 179, and 180)

[SNS03] David Stutz, Ted Neward, and Geoff Shilling: *Shared Source CLI Essentials*. Nutshell Books. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 1st edition, March 2003. (Cited on page 95)

[Sol07] David Solomon: *Data Compression: The Complete Reference*. Springer, 4th edition edition, January 2007. (Cited on page 166)

[SP04] Sathiamoorthy Subbarayan and Dhiraj K Pradhan: *Niver: Non increasing variable elimination resolution for preprocessing sat instances*. In Holger H. Hoos and David G. Mitchell (editors): *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*, volume 3542 of *Lecture Notes on Computer Science*, pages 276–291, St. Andrews, Scotland, June 2004. Springer Berlin / Heidelberg. (Cited on page 268)

[SS87] Dale Skeen and Michael Stonebraker: *A formal model of crash recovery in a distributed system*. IEEE Transactions on Software Engineering, 9(3):295–317, 1987. (Cited on page 144)

[Sta08] Standard Performance Evaluation Corporation: *SPEC CPU2006 results*, August 2008. http://www.spec.org/cpu2006/results/cpu2006.html, Accessed: 7/2011. (Cited on page 190)

[Sta10] Stanford University: *Folding@home website*, December 2010. http://folding.stanford.edu/, Accessed: 7/2011. (Cited on pages 23 and 35)

[Stu02] Gideon Stupp: *Stateless termination detection*. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*, pages 163–172, Toulouse, France, October 2002. Springer London, UK. (Cited on page 225)

[Suna] Sun Microsystems Inc.: *Java Management Extensions (JMX) Remote API (JSR-160)*. http://jcp.org/en/jsr/detail?id=160, Accessed: 7/2011. (Cited on page 99)

[Sunb] Sun Microsystems Inc.: *Java Management Extensions (JMX) Specification (JSR-3)*. http://jcp.org/en/jsr/detail?id=3, Accessed: 7/2011. (Cited on pages 26, 45, and 99)

[Sunc] Sun Microsystems Inc.: *JConsole*. http://download.oracle.com/javase/1.5.0/docs/guide/management/jconsole.html, Accessed: 7/2011. (Cited on page 104)

[Sut05] Herb Sutter: *A fundamental turn toward concurrency in software*. Dr. Dobb's Journal, 30(3), March 2005. http://drdobbs.com/architecture-and-design/184405990, Accessed: 7/2011. (Cited on page 285)

[Sut08] Herb Sutter: *Going superlinear*. Dr. Dobb's Journal, 33(3), January 2008. http://drdobbs.com/cpp/206100542, Accessed: 7/2011. (Cited on page 295)

[SV05] Daniel Singer and Alain Vagner: *Parallel resolution of the satisfiability problem (sat) with openmp and mpi*. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski (editors): *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM'05)*, volume 3911 of *Lecture Notes in Computer Science*, pages 380–388, Poznan, Poland, September 2005. Springer Berlin / Heidelberg. (Cited on page 279)

[SW03] Yudong Sun and Cho Li Wang: *Solving irregularly structured problems based on distributed object model*. Parallel Computing, 29(11–12):1539–1562, 2003. (Cited on pages 24 and 37)

[TAA$^+$03] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean Christophe Hugly, Eric Pouyoul, and Bill Yeager: *Project JXTA 2.0 Super-Peer Virtual Network*. Technical report, Sun Microsystems, May 2003. (Cited on pages 61, 105, 144, 180, and 181)

[TB05] Niklas Therning and Lars Bengtsson: *Jalapeno: Decentralized grid computing using peer-to-peer technology*. In *Proceedings of the 2nd Conference on Computing Frontiers (CF '05)*, pages 59–65, Ischia, Italy, May 2005. ACM Press New York, NY, USA. (Cited on pages 53 and 61)

[TB07] Bernhard Thomaszewski and Wolfgang Blochinger: *Physically based simulation of cloth on distributed memory architectures*. Parallel Computing, 33(6):377–390, 2007. (Cited on page 41)

[Teh00] Rich Tehrani: *Communications solutions publisher's outlook: As we may communicate*, January 2000. http://www.tmcnet.com/articles/comsol/0100/0100pubout.htm, Accessed: 7/2011. (Cited on pages 30 and 289)

[TGSR04] Andrei Tsaregorodtsev, Vincent Garonne, and Ian Stokes-Rees: *Dirac: A scalable lightweight architecture for high throughput computing*. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, pages 19–25, Pittsburgh, PA, USA, November 2004. IEEE Computer Society Washington, DC, USA. (Cited on page 182)

[tin] *Tiny Core Linux Project Website*. http://www.tinycorelinux.com/, Accessed: 3/2011. (Cited on page 87)

[TOP10] TOP500.Org: *Top500*. http://www.top500.org, December 2010. Accessed: 7/2011. (Cited on pages 33, 35, and 286)

[TS10] Pablo G. S. Tiburcio and Marco Aurelio Spohn: *Ad hoc grid: An adaptive and self-organizing peer-to-peer computing grid*. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology (CIT '10)*, pages 225–232, Bradford, UK, July 2010. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 62)

[TSSE10] Fabiano Costa Teixeira, Marcos Jose Santana, Regina Helena Carlucci Santana, and Julio Cezar Estrella: *Grid anywhere: An architecture for grid computing able to explore the computational resources of the set-top boxes*. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Anastasios Doulamis, Joe Mambretti, Ioannis Tomkos, and Theodora Varvarigou (editors): *Proceedings of the 4th International ICST Conference on Networks for Grid Applications (GridNets'10)*, volume 25 of *Lecture Notes of the Institute for Computer Sciences*, pages 79–88, Chicago, IL, USA, September 2010. Springer Berlin / Heidelberg. (Cited on page 290)

[UN 10] UN News Service: *Robust demand for mobile phone services will continue, un agency predicts*, February 2010. http://www.un.org/apps/news/story.asp?NewsID=33770&Cr=Telecom&Cr1=, Accessed: 2/2011. (Cited on page 33)

[Uni] University of California, Berkeley: *Seti@home*. http://setiathome.ssl.berkeley.edu/, Accessed: 2/2011. (Cited on pages 24 and 43)

[Uni10] University of California, Berkeley: *Boinc website*, December 2010. http://boinc.berkeley.edu/, Accessed: 7/2011. (Cited on pages 25 and 36)

[VB01] Miroslav N. Velev and Randal E. Bryant: *Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors*. In *Proceedings of the 38th Conference on Design Automation Conference (DAC'01)*, pages 226–231, Las Vegas, Nevada, USA, June 2001. ACM Press, New York, USA. (Cited on pages 29, 48, and 245)

[VC08] Monica Vladoiu and Zoran Constantinescu: *A taxonomy for desktop grids from users' perspective*. In *Proceedings of the International Conference of Parallel and Distributed Computing (ICPDC '08), a Conference of World Congress on Engineering 2008 (WCE '08)*, volume 1, pages 599–604, London, UK, July 2008. International Association of Engineers, Hung To Road, Hong Kong. (Cited on page 53)

[VD76] K. Vairavan and R.A. Demillo: *On the computational complexity of a generalized scheduling problem*. IEEE Transactions on Computers, 25(11):1067–1073, 1976. (Cited on page 205)

[VHR$^+$08] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar: *An 80-tile sub-100-w teraflops processor in 65-nm cmos*. Journal of Solid-State Circuits, 43(1):29–41, January 2008. (Cited on page 285)

[vir] *VirtualBox Project Website*. http://www.virtualbox.org/, Accessed: 3/2011. (Cited on page 87)

[VJvS03] Spyros Voulgaris, Márk Jelasity, and Maarten van Steen: *A robust and scalable peer-to-peer gossiping protocol*. In *Proceedings of the 2nd International Workshop on Agents and Peer-to-Peer Computing (AP2PC '03)*, volume 2872 of *Lecture Notes in Computer Science*, pages 47–58, Melbourne, Australia, July 2003. Springer Berlin / Heidelberg. (Cited on page 63)

[VN] Arun Viswanathan and B.C. Neuman: *A survey of isolation techniques*. http://www.arunviswanathan.com/survey_isolation_techniques.pdf, Accessed: 2/2011. (Cited on page 95)

[vNKB01] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal: *Efficient load balancing for wide-area divide-and-conquer applications*. SIGPLAN Notices, 36(7):34–43, June 2001. (Cited on page 57)

[VNRS02] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov: *Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment*. In Manish Parashar (editor): *Proceedings of the 3rd International Workshop on Grid Computing (GRID '02)*, volume 2536 of *Lecture Notes in Computer Science*, pages 1–12, Baltimore, MD, USA, November 2002. Springer London, UK. (Cited on pages 53 and 61)

[vNWJB10] Rob V. van Nieuwpoort, Gosia Wrzesińska, Ceriel J.H. Jacobs, and Henri E. Bal: *Satin: A high-level and efficient grid programming model*. ACM Transactions on Programming Languages and Systems, 32(3):1–39, 2010. (Cited on pages 53 and 57)

[vR03] Robbert van Renesse: *The importance of aggregation*. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao (editors): *Future Directions in Distributed Computing*, volume 2584 of *Lecture*

*Notes in Computer Science*, pages 87–92. Springer Berlin / Heidelberg, 2003. (Cited on pages 27, 47, and 185)

[Wan05] Ian Wang: *P2PS (Peer-to-Peer Simplified)*. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005. (Cited on page 181)

[Wed] Sebastian Wedeniwski: *ZetaGrid website*. `http://www.zetagrid.net/`, Accessed: 3/2011. (Cited on page 280)

[Weg09] Tobias Wegner: *A secure multi-provider OSGi platform enabling process-isolation by using distribution*. In Hamid R. Arabnia and Kevin Daimi (editors): *Proceedings of the International Conference on Security and Management (SAM '09)*, pages 340–345, Las Vegas, Nevada, USA, July 2009. CSREA Press. (Cited on page 95)

[WEK01] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann: *yfiles: Visualization and automatic layout of graphs*. In Petra Mutzel, Michael Jünger, and Sebastian Leipert (editors): *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 453–454, Vienna, Austria, September 2001. Springer. (Cited on page 160)

[WNB07] R. Wolski, D. Nurmi, and J. Brevik: *An analysis of availability distributions in Condor*. In *Proceedings of the IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–6, Long Beach, CA, USA, March 2007. IEEE Computer Society. (Cited on page 237)

[WSH99] Rich Wolski, Neil Spring, and Jim Hayes: *Predicting the cpu availability of time-shared unix systems on the computational grid*. In *Proceedings of the the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 105–112, Redondo Beach, California, USA, August 1999. IEEE Computer Society Washington, DC, USA. (Cited on pages 23 and 36)

[WTK07] Hong Wang, Hiroyuki Takizawa, and Hiroaki Kobayashi: *A dependable peer-to-peer computing platform*. Future Generation Computer Systems, 23(8):939–955, November 2007. (Cited on pages 53 and 59)

[WvNMB05] Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal: *Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid*. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, April 2005. IEEE Computer Society. (Cited on page 57)

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown: *Satzilla: portfolio-based algorithm selection for sat*. Journal of Artificial Intelligence Research, 32(1):565–606, 2008. (Cited on page 253)

[XL96] Chengzhong Xu and Francis C. M. Lau: *Efficient termination detection for loosely synchronous applications in multicomputers*. IEEE Transactions on Parallel and Distributed Systems, 7(5):537–544, 1996. (Cited on page 225)

[xsf] *The XMPP Software Foundation*. http://www.xmpp.org, Accessed: 3/2011. (Cited on pages 26, 46, 114, and 115)

[Xtr] XtremeJ Corp: *Xtremej Management Suite v3.0*. http://www.xtremej.com, Accessed: 7/2011. (Cited on page 104)

[YD04] Praveen Yalagandula and Michael Dahlin: *A scalable distributed information management system*. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall (editors): *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '03)*, pages 379–390, Karlsruhe, Germany, August 2004. ACM Press. (Cited on pages 187 and 192)

[ZBH96] H. Zhang, M. P. Bonacina, and J. Hsiang: *PSATO: A distributed propositional prover and its application to quasigroup problems*. Journal of Symbolic Computation, 21(4–6):543–560, 1996. (Cited on pages 251 and 279)

[ZHS+03] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz: *Tapestry: A global-scale overlay for rapid service deployment*. IEEE Journal on Selected Areas in Communications, 22(1):41–53, 2003. (Cited on pages 179 and 180)

[ZL06] Dayi Zhou and Virginia Lo: *Wavegrid: A scalable fast-turnaround heterogeneous peer-based desktop grid system*. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, pages 28–37, Rhodes Island, Greece, April 2006. IEEE Computer Society Washington, DC, USA. (Cited on pages 53 and 65)

[ZLL11] Han Zhao, Xinxin Liu, and Xiaolin Li: *A taxonomy of peer-to-peer desktop grid paradigms*. Cluster Computing, 14(2):129–144, June 2011. (Cited on page 53)

[ZM02] L. Zhang and S. Malik: *The quest for efficient boolean satisfiability solvers*. In Andrei Voronkov (editor): *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 313–331, Copenhagen, Denmark, July 2002. Springer Berlin / Heidelberg. (Cited on page 248)

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik: *Efficient conflict driven learning in a boolean satisfiability solver*. In *Proceedings of the IEEE/ACM international conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, San Jose, California, USA, November 2001. IEEE Press, Piscataway, New Jersey, USA. (Cited on page 248)

[ZYX09] Zhikun Zhao, Feng Yang, and Yinglei Xu: *Ppvc: A p2p volunteer computing system*. In *Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology*, pages 51–55, Beijing, China, August 2009. IEEE Computer Society Los Alamitos, CA, USA. (Cited on pages 53 and 65)

# Index

Sven Trieflinger

# High-Performance Peer-to-Peer Desktop Grid Computing

## Architecture, Methods, Applications

Although today's largest Desktop Grid harvests idle cycles from only 0.46‰ of the Personal Computers (PC) deployed world-wide, it is on par with the currently fastest supercomputer with respect to raw computing performance. If it were possible to attract roughly 7% of all PC owners to donate their resources, the resulting virtual supercomputer would *right now* punch through the exascale barrier expected to be broken by supercomputers not until around the year 2020.

However, with respect to application support the full potential of Desktop Grid Computing has not yet been unleashed. Due to their centralized interaction model, existing Desktop Grids are limited to embarrassingly parallel applications. By complementing the foundations of Desktop Grid Computing systems with Peer-to-Peer concepts and methods, their scope can be extended to a special class of parallel applications from the field of High-Performance Computing called *Irregularly Structured Problems* (ISP). Examples are parallel search problems, raytracing, and N-Body simulations.

*Cohesion*, the next generation Desktop Grid Computing platform described in this thesis, is an amalgamation of novel approaches designed to retrofit the Desktop Grid Computing approach to support efficient and scalable execution of ISPs on one of today's most demanding parallel execution environments .

Institute for Parallel and Distributed Systems (IPVS) • University of Stuttgart