Transaktionsorientierte
Kommunikationsmechanismen
im verteilten
Transaktionssystem
APRICOTS

Dieter Laube

26. März 1992

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis





# Kapitel 1

# Einführung

## 1.1 Vorbemerkung

Die vorliegende Arbeit wurde im Rahmen des Teilprojektes „Transaktions konzepte und Recovery-Verfahren für Objektbanken" innerhalb des DFG-Schwerpunktprogramms „Objektbanken für Experten" erstellt. Ihr Inhalt ist die Dokumentation des Entwurfs und der Implementierung eines Kommunikationsmechanismus, der für das ConTract[1]-verarbeitende System APRICOTS[2] entwickelt wurde, das in [23], [20], [21] und [22] ausführlich beschrieben ist.

Ein ConTract-verarbeitendes System dient der Abwicklung langlebiger, verteilter Datenbankanwendungen. Die Aufgaben des Kommunikationssystems betreffen die Namensvergabe, die Empfängerbestimmung und das Versenden bzw. Empfangen von Nachrichten. Diese Begriffe mit Bedeutung zu füllen und anschaulich zu machen, dazu dient dieses erste, einführende Kapitel. Zunächst wird begründet, warum es eines besonderen Verarbeitungsmodells bedarf, um Nonstandard-Datenbankanwendungen angemessen zu unterstützen. Daran anschließend werden das ConTract-Verarbeitungsmodell und APRICOTS als eine Realisierung vorgestellt. Parallel dazu werden aus den Eigenschaften eines Systems für **verteilte** Transaktionsverarbeitung, wie es APRICOTS bildet, die Anforderungen an ein Kommunikationssystem entwickelt.

Das zweite Kapitel beginnt mit der Analyse der Kommunikation in einem ConTract-verarbeitenden System und führt zu einem Entwurf, der die Grundlage dieser Arbeit bildet. Das Modell für das APRICOTS-Kommunikationssystem wird nach Aufbau und Eigenschaften beschrieben, ohne auf Implementierungsdetails einzugehen.

Das folgende Kapitel vollzieht anhand von Beispielen den Ablauf eines Dienstaufrufs in APRICOTS nach und möchte das Verständnis für das Konzept des **Resource Manager Calls** vertiefen, indem alle Aspekte der Kommunikation gemeinsam im Überblick erscheinen.

Die Phase der Implementierung wird hauptsächlich im nächsten Kapitel dokumentiert, wobei es mir ein Anliegen war, nicht nur die Realisierung selbst,

[1] Controlled activities, controlled transactions compound

[2] A PRototype Implementation of the COnTract System

sondern auch die verschiedenen Alternativen und letztendlichen Entscheidungskriterien festzuhalten.
Wenn das vierte Kapitel unter den Begriff „Algorithmen" fällt, so kann das fünfte Kapitel als Einblick in die verwendeten Datenstrukturen verstanden werden. Schwerpunkt ist dabei die Erläuterung der Adressierung in Hinblick auf die Adreßstruktur, die Adreßhaltung und -pufferung sowie das Nachrichtenprotokoll.
Das Kapitel 6 ist der Fehlerbehandlung gewidmet. Es beschreibt die Fehlerklassen bei der ConTract-Verarbeitung und enthält einen Entwurf für die Protokollierung von Nachrichten unter dem Gesichtspunkt der Recovery.
Das abschließende Kapitel faßt die Ergebnisse der bisher geleisteten Arbeit zusammen. Im Vergleich mit dem Remote Procedure Call wird das Konzept des Resource Manager Calls herausgearbeitet. Eine Leistungsmessung weist die Effizienz des Prototypen nach. Zuletzt wird ein Ausblick auf die nächsten Ausbaustufen des Kommunikationssystems für APRICOTS gegeben.
Im Anhang werden die Schnittstellen der Funktionen für das Kommunikationssystem ausführlich und nach Gruppen geordnet vorgestellt.

Als Einstieg in diese Arbeit ist es für Leser und Leserinnen, die nicht mit APRICOTS vertraut sind, unbedingt empfehlenswert, dieses erste einführende Kapitel zu lesen. Es enthält einen exemplarischen ConTract und eine Beispielkonfiguration, auf die ich mich im Verlauf dieser Arbeit immer wieder beziehen werde. Ich habe mich bemüht, den Einstieg zu erleichtern, indem ich häufig Querverweise auf vertiefende Kapitel bzw. Abschnitte zu Detailfragen eingeflochten habe.
Die einschlägige Literatur zur Interprozeßkommunikation, insbesondere alternative Vorschläge zur Realisierung von asynchronen Nachrichten, habe ich nicht von der Dokumentation getrennt, sondern sie in die jeweiligen Abschnitte integriert, in denen sie mit dem vorliegenden Modell verglichen werden können.
Die Konventionen in der Typenwahl sind schnell beschrieben: Funktionsnamen, Variablen- und Typbezeichner sind *kursiv*, Pseudocode ist in `Schreibmaschinenschrift` wiedergegeben.

## 1.2 Motivation für ConTracts

Das Transaktionsmodell, das in den Siebziger Jahren entwickelt wurde, begreift Transaktionen als atomare, geschützte Arbeitseinheiten, deren Ausführung mit den ACID[3]-Garantien versehen wohldefinierte, konsistenzerhaltende Zustandsänderungen in einer Datenbank bewirkt.
Das klassische Transaktionskonzept bietet jedoch nur eine gute Performance, wenn die Transaktionen kleine Arbeitseinheiten sind, die nur wenige Datensätze berühren, wenn sie also nur kurze Zeit im System verbringen. Auch sieht das Modell keine Strukturierung z.B. in Form einer Kontrollflußbeschreibung zwischen mehreren Transaktionen vor. Im Fehlerfall muß die gesamte Transaktion

[3] **A**tomicity, **C**onsistency, **Isolation**, **D**urability



durch Roll-Back zurückgesetzt werden.
Diese Eigenschaften sind für „langlebige" Verarbeitungseinheiten wie z.B. bei CAD oder bei komplexen Fertigungssteuerungen ungeeignet [21]. Die ACID-Eigenschaften für Transaktionen müssen modifiziert werden, damit die folgenden Bedingungen erfüllt sind:
Bei einer Migration einer Verarbeitung von einem Rechner auf einen anderen oder einem Systemabsturz muß die Verarbeitung ohne globales Rücksetzen fortgesetzt werden können.
Sperren auf berührten Objekten müssen nicht der Serialisierbarkeit genügen, sondern es reicht, anwendungsdefinierte Isolationsbedingungen — sogenannte Invarianten — zu erfüllen, um die Konsistenz der Zustandsübergänge im System zu wahren.
Es müssen Konstrukte angeboten werden, die es erlauben, logische Beziehungen zwischen den einzelnen atomaren Arbeitseinheiten zu definieren.
Auf dieser Basis wurde das ConTract-Modell entwickelt ([21]). Es umfaßt die folgenden Merkmale (nach [23]):

- mehrstufiges Programmiermodell für verteilte Anwendungen
- integrierte Fehlerbehandlung, Betriebsmittel-Verwaltung und Kontrollflußabwicklung
- modulare Architektur
- flexible Kommunikationsmechanismen für heterogene Systeme
- Forward Recovery (*continue* statt *undo*)
- Kompensationsmechanismen

Details zur Semantik von ConTracts und zum Programmiermodell finden sich in [21] und [23].

## 1.3 ConTracts

ConTracts sind ein neuer Ansatz für verteilte, langlebige Datenbankanwendungen. Herkömmliche Anwendungen basieren auf in sich abgeschlossenen Transaktionen. Ein wesentliches Merkmal eines ConTract ist dagegen die Mehrstufigkeit der Programmierung. Ein ConTract ist eine komplexe „Transaktion", die sich aus elementaren Arbeitsschritten, die Transaktionen entsprechen, zusammensetzt.
Dieser Abschnitt zeigt an einem einfachen Beispiel, wie diese Mehrstufigkeit realisiert wird, und führt die Terminologie für die ConTract-Verarbeitung ein.

### 1.3.1 Beispiel für einen ConTract

Eine typische verteilte Datenbankanwendung ist die Verknüpfung von Aktionen auf mehreren Datenbanken. So wird es in Zeiten wachsender Mobilität bei

```
BEGIN_CONTEXT
  start, ziel, abflugzeit, ankunftszeit,
  anzahl_tage, ......
END_CONTEXT

BEGIN_SCRIPT
  S1: flugdateneingabe
  PARBEGIN
    S2: flugbuchung
    S3: hotelreservierung
  PAREND
  S4: buchungsdaten_ausgeben
END_SCRIPT

BEGIN_TRANSACTIONS
  T1 (S2)
  T2 (S3)
  DEPENDENCY (T1 abort ↦ abort T2)
  DEPENDENCY (T2 abort ↦ abort T1)
END_TRANSACTIONS
```

Abbildung 1.1: Beispiel für ein Skript

gleichzeitig stetig zunehmender Bequemlichkeit für professionelle Reiseunternehmen unabdingbar, Kombinationen von Reservierungen anzubieten: Flugbuchung plus Hotelreservierung, Hotelreservierung mit Leihwagen oder gleich alles zusammen. Ich stelle hier ein Beispiel vor, das eine Flugbuchung mit einer Hotelreservierung kombiniert und den Kunden bzw. die Kundin mit gedruckten Bestätigungen dieser Vorgänge versorgt.
Für die Flugbuchung werden zwei Alternativen angeboten: Tibetan Oneways verlangt jedoch einen Reinkarnationsnachweis mittlerer Güte, sodaß diesseitig ausgerichtete Mitteleuropäer meist die Lusthansa bevorzugen.[4] Unsere Kunden und Kundinnen sind bei der Hotelreservierung darauf angewiesen, in Etablissements der Geigenberger-Kette abzusteigen. Der gesamte Vorgang ist nur dann erfolgreich, wenn beide Buchungen durchgeführt werden können.
Die formale Beschreibung des Kontrollflusses erfolgt in Form eines sogenannten Skriptes, das in Abb. 1.1 dargestellt ist.
Ein Skript gliedert sich in drei Blöcke: Kontextdefinitionen (*BEGIN_KONTEXT ... END_CONTEXT*), Kontrollflußdefinitionen (*BEGIN_SCRIPT ... END_SCRIPT*) und Definitionen zur Transaktionsverwaltung (*BEGIN_TRANSACTIONS ... END_TRANSACTIONS*).
Ich beginne mit der Kontrollflußdefinition, die die elementaren Arbeitsschritte des ConTracts, die **Steps**, festlegt. Der Vorgang beginnt mit der Online-Erhebung der Flugdaten im Step $S_1$ mit dem Namen *flugdateneingabe*. Daran schließt sich eine parallel ausführbare Sequenz an, die durch *PARBEGIN* einge-

[4]Kunden und Kundinnen der Tibetan Oneways unterliegen weiteren Restriktionen: sie können nur Einfachtickets in den Himalaya erwerben.



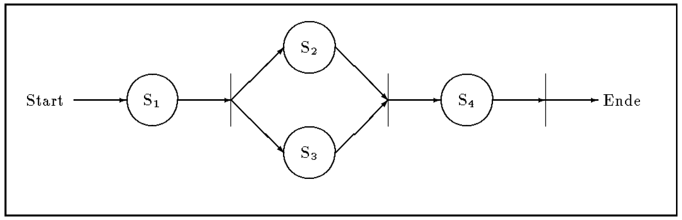


Abbildung 1.2: Graphische Darstellung für das Skript aus Abb. 1.1

leitet wird. Sie enthält die Steps $S_2$ und $S_3$, die für die Aktionen *flugbuchung* sowie *hotelreservierung* stehen.[5] Nach dem Ende dieser parallelen Sequenz (*PAREND*) erfolgt die Ausgabe der Buchungsdaten auf dem Bildschirm und auf dem Drucker durch den Step $S_4$ (*buchungsdaten_ausgeben*).
Die Kontextdefinition beschreibt die Daten, die zwischen den einzelnen Verarbeitungsschritten ausgetauscht werden.[6] *start* und *ziel* beispielsweise werden in *flugdateneingabe* erfragt und werden vom Step *flugbuchung* benötigt, um den richtigen Flug zu berechnen.
Die Transaktionsverwaltung definiert die im ConTract enthaltenen Transaktionen $T_1$ und $T_2$, die jeweils einen Step umfassen. Außerdem werden die Abhängigkeiten zwischen verschiedenen Transaktionen festgelegt. So muß die Transaktion $T_2$, die die Hotelreservierung vornimmt, abgebrochen werden, wenn die Transaktion mit der Flugbuchung ($T_1$) scheitert und umgekehrt.
Mithilfe von an Petri-Netze angelehnten Symbolen läßt sich ein Skript visualisieren. Die Stellen des Netzes entsprechen elementaren Verarbeitungsschritten. Der Kontrollfluß wird durch die Pfeile und Transitionen festgelegt. In Abb. 1.2 wird diese Darstellungsweise für das Skript aus Abb. 1.1 gezeigt. $S_1$ ist die Erfassung der für die Buchungen benötigten Eingangsdaten. $S_2$ entspricht dem Vorgang Flugbuchung, $S_3$ dem Vorgang Hotelreservierung. $S_4$ versinnbildlicht den Ausdruck der Formulare für beide Vorgänge.
Der erfolgreiche Abschluß von $S_1$ „triggert" die Steps $S_2$ und $S_3$, deren parallele Ausführbarkeit in Abb. 1.2 deutlich sichtbar ist. Nur wenn beide Steps erfolgreich ausgeführt sind, enthalten sie „Marken" und können damit die Ausführung von $S_4$ auslösen.
Oberhalb der Ebene der einzelnen Transaktion existieren also noch Kontrollmechanismen, die die Abhängigkeiten der Transaktionen untereinander und vom Verlauf des ConTracts selbst regeln. Die Entscheidung über das Gelingen der beiden Transaktionen liegt konsequenterweise bei einem Modul, das nicht selbst

[5]Das ist natürlich eine Vereinfachung, da die Flugzeiten je nach Anschluß mehr oder minder stark differieren können.
[6]In Abb. 1.1 sind die Kontextdaten nur auszugsweise und ohne ihre Datentypen angegeben.

einen Step ausführt.[7] Neben den Modulen mit den eigentlichen Elementaraktionen existieren Module wie ConTract Manager oder Transaction Manager, die Kontrollaktionen realisieren, wie z. B. ein *prepare_to_commit* für $T_1$ und $T_2$, die Generierung eines neuen Transaktionsbezeichners (*get_gtrid*) oder die Eintragung einer Transaktionsbeziehung in ein Transaktionsbeziehungsnetz wie *create_ta_dependency*.
Bedeutsam ist auch die Adressierungsweise. So werden Elementaraktionen wie $S_1$ oder $S_2$ nicht über einen Server-Namen, d. h. den Namen der ausführenden Instanz, adressiert, sondern über ihren Step-Namen (z. B. *flugdateneingabe, flugbuchung*). Diese Adressierung wird bei der Abwicklung des ConTracts durch die APRICOTS-Komponenten beibehalten.
Ein weiterer wichtiger Punkt ist die Orientierung an dem Event-Trigger-Mechanismus für die ConTract-Verarbeitung. Sie fordert von der Kommunikation einen Nachrichtenmechanismus, der diese Ereignisorientierung angemessen unterstützt.

### 1.3.2 Grundbegriffe der ConTract-Verarbeitung

Auf einer formaleren Ebene ist ein ConTract

> die fehlergeschützte Ausführung einer Folge vordefinierter Verarbeitungsschritte gemäß einer vorgegebenen Kontrollflußstruktur.[21]

Die Ablauforganisation eines ConTract erfolgt durch ein in einer Metasprache formuliertes **Skript**, dessen Darstellung in Abb. 1.1 zu sehen ist. Im Skript wird mithilfe von **Kontrollkonstrukten** festgelegt, wie die oben erwähnten elementaren Verarbeitungsschritte, **Steps** genannt, koordiniert werden. Ein Step kann aus einer Transaktion, z. B. einer Flugbuchung oder Hotelreservierung, bestehen. Kontrollkonstrukte entsprechen Operationen, die den Kontrollfluß steuern wie *PAR_BEGIN* oder Transaktionsverwaltungsaktionen wie *get_gtrid*, *abort_ta* oder *create_ta_dependency*.
Steps und Kontrollkonstrukte werden unter dem Begriff **Dienst**[8] zusammengefaßt.
Dienste werden wiederum nach Funktionalität zusammengefaßt. So bilden alle Dienste, die Operationen auf einer bestimmten Datenbank ausführen, z. B. die Flugbuchung und -stornierung für die Fluggesellschaft Lusthansa, eine Dienstklasse, die den Namen *LUSTHANSA* erhält. Auch die Dienste der Transaktionsverwaltung wie *get_gtrid* oder *create_ta_dependency* bilden eine Dienstklasse, die mit dem Namen *TM* versehen wird.[9]
Die Definition einer Dienstklasse wird **Resource Manager Klasse** genannt. Eine Instanz einer Resource Manager Klasse wird als **Resource Manager** bezeichnet.

[7] Unter der Voraussetzung, daß Modularität als Trennung funktionaler Einheiten verstanden wird.
[8] Im Text werden die Begriffe **Dienst** und **Service** wechselweise benutzt und bezeichnen dasselbe.
[9] *TM* steht für Transaction Manager.



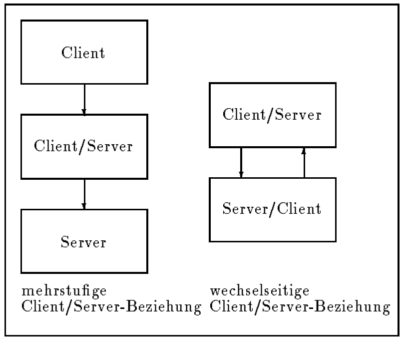


Abbildung 1.3: Client-Server-Beziehungen bei der ConTract-Verarbeitung

Auf der Ebene der Kontrollflußbeschreibung sind der Dienstname und der Name der Resource Manager Klasse ausreichende Identifikationsmerkmale zur Kennzeichnung der Ablaufschritte.
Das ConTract-Verarbeitungsmodell basiert — als Folge des mehrstufigen Programmiermodells und des modularen Aufbaus — auf der Client-Server-Beziehung zwischen Resource Managern. Ein Dienst $A$ ruft einen Dienst $B$ auf, der wiederum seinerseits weitere Dienste aufrufen kann, sodaß eine über mehrere Stufen reichende Client-Server-Beziehung entstehen kann (siehe Abb. 1.3). Abb. 1.3 zeigt ein weiteres Merkmal für die Beziehungen zwischen Resource Managern: zwei Resource Manager, respektive ihre Dienste, können sich wechselseitig Client und Server sein.
Daraus ergibt sich als eine Forderung an die Kommunikation für ein ConTract-verarbeitendes System, daß eine Komponente im Nachrichtenverkehr gleichzeitig sowohl Client als auch Server sein kann und das auch gegenüber derselben Partnerkomponente.

## 1.4 APRICOTS

Dieser Abschnitt stellt den Prototypen eines ConTract-verarbeitenden Systems vor, für den das in dieser Arbeit vorgestellte Kommunikationssystem entwickelt wurde. Zunächst führe ich die Begrifflichkeit ein. Danach wird das Prozeßsystem in APRICOTS erläutert und an einer kleinen Beispielkonfiguration anschaulich gemacht.

### 1.4.1 Resource Manager Klassen eines ConTract-Systems

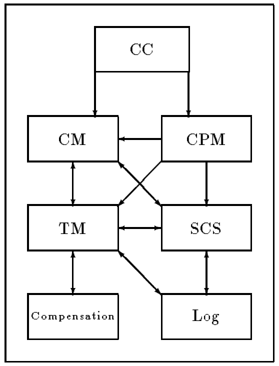


Abbildung 1.4: Module eines APRICOTS-Systems

Jede der im folgenden aufgeführten Module, Resource Manager Klasse genannt, bietet den anderen Modulen Dienste an, die durch einen **Dienstaufruf** aktiviert werden können. Mehrere Module können zu einem ablauffähigen Programm zusammengebunden werden, das als **Komponente** bezeichnet wird. Die im folgenden aufgeführten Resource Manager Klassen werden als Instanzen mit ihren Aufrufbeziehungen in Abb. 1.4 gezeigt.
Das **Kommunikationssystem (CS)** bietet selbst keine Dienste an, sondern vermittelt lediglich Dienstaufrufe und ermöglicht die Aufrufbeziehungen, die in Abb. 1.4 dargestellt sind. Es führt den Aufbau, die Erhaltung und den Abbau von Kommunikationskanälen durch, verfügt über eine integrierte Fehlerbehandlung und führt die Empfänger- und Adreßbestimmung durch.

Der **ConTract Client (CC)** ist eine Anwendung, die einen ConTract anstößt und dessen Verarbeitung durch Aufrufe an den ConTract-Manager beeinflußt.

Der **ConTract Processing Monitor (CPM)** initialisiert das ConTract-verarbeitende System, verwaltet die Prozeßkonfiguration, überwacht die Kommunikationskanäle und verwaltet die globalen Adreßtabellen.

Der **ConTract Manager (CM)** interpretiert den Kontrollfluß und wickelt den ConTract ab, bildet Steps auf Transaktionen ab und nimmt ConTract-Client-Befehle zur Ablaufsteuerung entgegen.

Der **Transaction Manager (TM)** verwaltet Transaktionen, überwacht die Transaktionsbeziehungen in einem ConTract und koordiniert das Commit-Protokoll für verteilte Transaktionen.

Der **Step Computation Server (SCS)** führt einen Step aus, der unter Transaktionsschutz steht.



weitere Module wie **Compensation Manager** und **Log Manager** verwalten die notwendigen Informationen für Recovery, Rücksetzen und Migration.

An dieser Stelle muß hinzugefügt werden, daß die Resource Manager Klasse **Step Computation Server** ein Oberbegriff für alle Resource Manager Klassen ist, die Elementaraktionen (Steps) anbieten.
So ist die Resource Manager Klasse *lusthansa* eine Zusammenfassung von Diensten, die Buchungen oder Stornierungen von Flügen der Lusthansa-Fluggesellschaft abwickeln. Die Resource Manager Klasse *tibetan_oneways* bietet oberflächlich gesehen dieselben Dienste an, doch sind die Ausführungsdetails sehr verschieden von denen der Lusthansa.[10]

### 1.4.2 TP-Monitor und ConTract Processing Monitor

An dieser Stelle bietet es sich an, auch die Rolle des ConTract Processing Monitor mit dem Konzept des TP-Monitors zu vergleichen, nachdem der Unterschied zwischen Transaktionen und ConTracts nun schon erläutert ist.
Die klassische Definition des TP-Monitors (vgl. [15]) geht von herkömmlichen Transaktionen mit wenigen DB-Operationen sowie einer Trennung in die Transaktionskomponenten DB-System und DC-System aus.
Diese Sichtweise läßt sich bei der ConTract-Verarbeitung nur noch beschränkt anwenden. So ist zum Beispiel die 1:1-Zuordnung von DB-Transaktionen und DC-Transaktionen (Nachrichten), ebenso wie die 1:1-Zuordnung von Dialogschritten und Transaktionen in der ConTract-Verarbeitung aufgehoben, da eine Transaktion mehrere über das Kommunikationssystem vermittelte Aufträge umfassen kann.[11]
Eine Transaktion kann in mehrere Steps (Dienstaufrufe) zerlegt werden, die Unteraufträge, d. h. weitere Dienstaufrufe an andere Komponenten, nach sich ziehen können. In dieser Hinsicht können einer Transaktion $m$ Nachrichten zugeordnet werden.
Kontrollkonstrukte wie *get_gtrid* und *create_ta_dependency* werden ebenfalls als Nachrichten verschickt.
Diese Eigenschaft von APRICOTS verlangt, daß jeder Prozeß über die Fähigkeit verfügen muß, Kommunikationsmechanismen anzustoßen. Daher ist zu jeder Komponente der Code des Kommunikationssystems dazugebunden. Die Folge ist, daß die Kontrolle über den Nachrichtenverkehr nicht mehr bei einer Komponente allein liegt.
Andere Aufgaben des klassischen TP-Monitors wie die Initialisierung und Konfigurierung des Systems sowie die Überwachung der Systemstruktur führt der ConTract Processing Monitor durch.
Dadurch ergibt sich eine Aufteilung des Aufgabenbereichs eines TP-Monitors in mehrere Module: Kommunikationssubsystem und ConTract Processing Mo-

[10] Tibetan Oneways bietet keine Rückflugtickets an und führt intern Plausibilitätsprüfungen für den Reinkarnationsnachweis durch.
[11] So kann für eine Flugbuchung der Vorgang in mehrere Anfragen an verschiedene Datenbanken unterschiedlicher Luftfahrtunternehmen zerlegt werden.

nitor. Der CPM ist eine eigene Instanz in der Prozeßkonfiguration wie der TP-Monitor, das Kommunikationssystem ein Modul jeder Instanz innerhalb von APRICOTS. Jeder Prozeß in APRICOTS kann selber Nachrichten versenden.

### 1.4.3 Prozesse in APRICOTS

Auf der Ebene der Implementierung wird der Semantik eines Dienstes ein Funktionscode zugeordnet. Hier stellt sich nun die Frage nach der Zuordnung von Funktionen (Diensten) zu Programmen.
In APRICOTS werden die Dienste einer oder mehrerer Resource Manager Klassen zu einer **Komponente** zusammengebunden. Dabei entsteht eine *n:m*-Zuordnung zwischen Resource Manager Klassen und Komponenten. Dies ist in Abb. 1.5 zu sehen.
Zu wird jedem Resource Manager, der an der Transaktionsverarbeitung beteiligt ist,[12] ein Resource Manager zur Transaktionsverwaltung, *TM* genannt, dazugebunden. Ein obligatorisches Modul jeder Komponente in APRICOTS ist das **Kommunikationssystem** (*CS*), das die Kommunikation zwischen verschiedenen Komponenten in APRICOTS und innerhalb einer Komponente realisiert.
Eine aktivierte Komponente stellt für das Betriebssystem eine Einheit der Betriebsmittelvergabe (Ein- und Ausgabekanäle) und damit für das Kommunikationssystem auf der Ebene der Kommunikationskanäle eine Einheit für die Adressierung dar.
Jede aktivierte Komponente wird von dem ConTract Processing Monitor, der auf dem gleichen Rechner wie sie selbst läuft, überwacht und erhält von diesem einen eindeutigen **Komponentennamen**.[13]
Die Zuordnung der Dienste und Resource Manager zu Komponentennamen ist in globale nTabellen festgehalten.[14]
Die Synchronisation der ConTract-Verarbeitung erfolgt durch wechselseitige Dienstaufrufe oder Resource Manager Calls. Dienstaufrufe werden vom Kommunikationssystem in Nachrichten umgesetzt, die innerhalb einer Komponente oder zwischen verschiedenen Komponenten versendet werden.

### 1.4.4 Beispielkonfiguration

Abb. 1.5 zeigt eine einfache Konfiguration für APRICOTS, das den Beispiel-ConTract aus Abb. 1.1 abwickeln kann.
Die Verarbeitung erfolgt auf zwei Rechnern: *agamemnon* und *kirke*. Für jeden Rechner überwacht ein ConTract Processing Monitor (CPM) alle Prozesse, die an der ConTract-Verarbeitung beteiligt sind. Auf *agamemnon* ist es der ConTract Processing Monitor *cpm_agamemnon*, auf *kirke* der CPM *cpm_kirke*.
Jede Komponente, die an APRICOTS teilnimmt, einschließlich des ConTract Client (CC), hat das Kommunikationssystem (CS) in den eigenen Code eingebunden. Die Resource Manager, die direkt an der Transaktionsverarbeitung be-

[12] In Abb. 1.5 sind das der ConTract Manager und die Step Computation Server.
[13] Zur Bildung und Vergabe des Komponentennamens vgl. Kap. 5.
[14] Der Name Service wird in Kap. 4 und in Kap. 5 beschrieben.



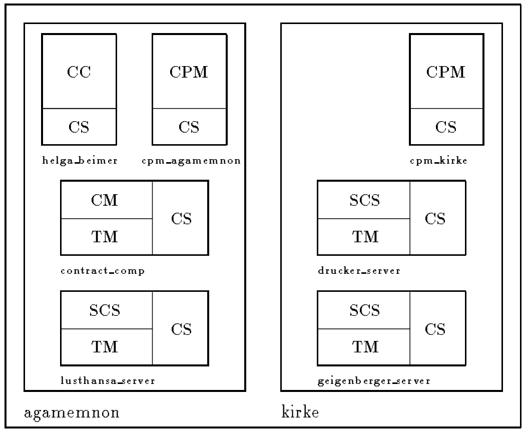


Abbildung 1.5: Beispielkonfiguration für APRICOTS

teiligt sind wie ConTract Manager (CM) oder Step Computation Server (SCS), werden zusätzlich mit dem Code des Transaction Manager (TM) zusammengebunden.
Die Komponente *lusthansa_server* bietet im Resource Manager *lusthansa* die Dienste *flugdateneingabe* und *flugbuchung* an. Der *geigenberger_server* bietet den Dienst *hotelreservierung* innerhalb des Resource Managers *geigenberger* an. Der *drucker_server* ist in der Lage, den Dienst *buchungsdaten_ausgeben* auszuführen.[15]
Der ConTract aus Abb. 1.1 wird folgendermaßen ausgeführt: *helga_beimer*, unsere ConTract-Klientin, möchte eine kombinierte Buchung abwickeln. Sie schickt ein *activate_contract* an die Komponente *contract_comp*. *contract_comp* ruft den Dienst *flugdateneingabe* auf, der von der Komponente *lusthansa_server* zur Verfügung gestellt wird. Nach Beendigung der Eingabe sendet *contract_comp* weitere Dienstaufrufe an die Step Computation Server der Konfiguration: der Aufruf *flugbuchung* wird an die Komponente *lusthansa_server* geschickt, der Aufruf *hotelreservierung* an *geigenberger_server*. Nach erfolgreichem Ende beider werden die Formulare von der Komponente *drucker_server* ausgedruckt. Damit ist der ConTract vollständig abgewickelt.

## 1.5 Anforderungen an das Kommunikationssystem

Aus den vorgestellten Beispielen lassen sich schon grundsätzliche Anforderungen an das Kommunikationssystem ableiten.

[15] Die Komponente *lusthansa_server* ist in Abb. 4.1 detailliert dargestellt.

- Eine Komponente muß im Client-Server-Modell gleichzeitig beide Rollen einnehmen können, auch gegenüber derselben Komponente.
- Die Ereignisorientierung der ConTract-Verarbeitung insbesondere bei der Step-Ausführung muß im Nachrichtenmechanismus berücksichtigt werden.
- Nachrichten werden zwischen verschiedenen Diensten innerhalb eines Prozesses, zwischen zwei Prozessen auf demselben Rechner oder auch zwischen Prozessen auf verschiedenen Rechnern versendet. Das Kommunikationssystem sollte eine einheitliche Nachrichtenschnittstelle für alle drei Fälle bereitstellen, damit die Transparenz für die aufrufenden Komponenten gewahrt bleibt.
- Die Adressierung von Steps erfolgt über den Step-Namen und nicht über die Server-Instanz.
- Auslagerungen von Komponenten, d. h. Migrationen von einem Rechner auf einen anderen, dürfen die Verarbeitung eines ConTract nicht beeinträchtigen. Eine vor der Migration empfangene Nachricht muß von der migrierten Komponente weiterverarbeitet werden.
- Die Möglichkeit der Verteilung von Transaktionen auf mehrere Komponenten mithilfe beliebiger Dienstaufrufe erfordert einen flexiblen Mechanismus zur Verwaltung verteilter Transaktionen.

Für Entwurf und Implementierung des Kommunikationssystems ist nach Vorgabe ein strikt serieller Ablauf maßgeblich. Multi-Threading erweist sich in der Programmierung des Kontrollflusses als zu komplex:

- Von mehreren Threads innerhalb eines Prozesses gemeinsam genutzte Resourcen benötigen zusätzliche Synchronisationsmechanismen: für die Verwaltung eines von mehreren Prozessen genützten Speicherbereiches benötigt eine Multi-Threading-Implementierung etwa zehn Semaphore gegenüber dreien bei der Single-Threading-Implementierung.
- Verwaltungsstrukturen des Kommunikationssystems für eine gesamte Komponente sind im Gegensatz zum Single-Threading nicht mehr überschaubar zu halten: Die Verwaltung der gerade arbeitenden Dienste kann bei der Single-Threading-Lösung aufgrund der Aufrufhierarchie durch einen Stack realisiert werden. Beim Multi-Threading kann jedoch weder ein gemeinsamer Stack aller Threads noch ein für jeden Thread privater Stack eine konsistente Verwaltung realisieren.
- Threads müssen explizit die Kontrolle abgeben, wodurch es notwendig wird, bei manchen Kommunikationsmechanismen wie Remote Procedure Calls den Sourcecode zu ändern.

Die Beschleunigung gegenüber Single-Threading ist dagegen gering, da nur eine quasiparallele Ausführung mehrerer Threads möglich ist. Aus diesen Gründen



erweist sich eine Single-Threading-Implementierung als die praktikablere Variante zur Realisierung eines komplexen Systems, wie es APRICOTS darstellt.

Eine weitere Vorgabe war, vorhandene Kommunikationsmechanismen so weit wie möglich zu verwenden. Es ist nicht Ziel dieser Arbeit, einen Remote Procedure Call neu zu implementieren, sondern, wo es dienlich erscheint, realisierte Konzepte in den Kommunikationsmechanismus von APRICOTS zu integrieren. Da diese Arbeit auf einer Sun-Workstation implementiert wurde, wurden Kommunikationsmechanismen, die das Betriebssystem SunOS anbietet, verwendet.

# Kapitel 2

# Modell für das Kommunikationssystem

Dieses Kapitel stellt den Entwurf für das Kommunikationssystem vor, der dieser Arbeit zugrundeliegt.
Die Analyse der Kommunikation in APRICOTS wird eingeführt mit zwei Beispielen, die prototypisch für einen Großteil des Nachrichtenaufkommens in APRICOTS sind: Kontrolloperationen zwischen ConTract Manager und Transaction Manager sowie Step-Aktivierungen.
Die Analyse verallgemeinert die Merkmale dieser Beispiele und berücksichtigt dabei die Erkenntnisse, die im letzten Kapitel über den Ablauf der ConTract-Verarbeitung und die Systemkonfiguration von APRICOTS gewonnen wurden.
Das anschließend vorgestellte Modell ist bewußt anschaulich gehalten und dient als erste Orientierung über die Eigenschaften und den Aufbau des Kommunikationssystems, die in den folgenden Abschnitten ausführlicher dargestellt werden.

## 2.1 Beispiele für Dienstaufrufe in APRICOTS

Wenn ein Dienst in APRICOTS einen anderen Dienst mit einer Aufgabe für die ConTract-Verarbeitung beauftragt, so wird das als **Dienstaufruf** bezeichnet. Dienstaufrufe werden vom Kommunikationssystem in **Nachrichten** umgesetzt, die vom aufrufenden an den aufgerufenen Dienst gesendet und — für die Antwort — vom aufgerufenen an den aufrufenden Dienst zurückgeschickt werden. Da die Adressierung eines Dienstes ortstransparent sein soll, werden alle Dienstaufrufe an der Schnittstelle zur Anwendungsebene gleich realisiert. Der jeweils implementierte Nachrichtenübertragungsmechanismus ist an die Entfernung zwischen Aufrufer und Aufgerufenem angepaßt.
Der Empfang einer Nachricht entspricht der **Aktivierung** des adressierten Dienstes. Wenn im folgenden von der Nachricht <*service_name*> die Rede ist, so bezeichnet <*service_name*> den Dienst, also eine Funktion, die ausgeführt werden soll.
Ich stelle hier zwei Dienstaufrufe vor, die typisch für das Nachrichtenaufkommen





bei der ConTract-Verarbeitung sind.

- *get_gtrid* ist ein Dienstaufruf des Transaction Managers durch den ConTract Manager mit der Aufforderung, einen neuen Transaktionsbezeichner zu generieren und zurückzusenden, um eine neue Transaktion innerhalb eines ConTract starten zu können.
- *flugbuchung* ist ein Dienstaufruf eines Step Computation Servers durch den ConTract Manager mit der Aufforderung, den Step *flugbuchung* zur Ausführung zu bringen. Die Antwort des Steps enthält die Ergebnisse der Step-Ausführung.

### 2.1.1 Beispiel *get_gtrid*

Der Aufruf *get_gtrid* gehört zu einer großen Gruppe von Dienstaufrufen, mit denen sich ConTract Manager und Transaction Manager gegenseitig aktivieren. Da das Nachrichtenaufkommen zwischen diesen beiden Teilkomponenten sehr groß ist, werden sie mit dem Kommunikationssystem in einer Komponente zusammengefaßt (vgl. Abb. 1.5). Der Dienstaufruf ist also prozeßlokal.
Ein spezifisches Kriterium für *get_gtrid* ist, daß der ConTract Manager nicht ohne die Antwort — hier: den neuen Transaktionsbezeichner — weiterarbeiten kann. Es stehen Steps zur Verarbeitung an, die mit einer Transaktionsnummer versehen werden müssen, um unter Transaktionsschutz ablaufen zu können. Erst wenn der ConTract Manager die Transaktionsnummer erhalten hat, kann ein Step, z. B. *flugbuchung*, aktiviert werden. Der Aufruf *get_gtrid* ist demnach synchroner Natur.
Der übertragene Datentyp ist ein Transaktionsbezeichner, also eine einfache Datenstruktur.

### 2.1.2 Beispiel *flugbuchung*

Der Aufruf *flugbuchung* wird vom ConTract Manager zur Aktivierung eines ausführbereiten Step versendet.
Grundsätzlich können gleichzeitig mehrere Steps zur Ausführung gebracht werden, wie es im Beispiel-Skript von Abb. 1.1 bei den Steps *flugbuchung* und *hotelreservierung* der Fall ist. Dann ist es sinnvoll, den Aufruf *flugbuchung* an eine andere Komponente zu senden, die parallel zum ConTract Manager arbeiten kann, und damit die Serialisierung des Ablaufs zu vermeiden. Die Ausführung des Auftrags soll also nicht prozeßlokal stattfinden.[1]
Dann ist es auch nicht sinnvoll, daß der ConTract Manager synchron die Bearbeitung jedes einzelnen Schritts abwartet, da sonst die Bearbeitung parallel ausführbarer Steps serialisiert würde. *flugbuchung* ist somit ein asynchroner Auftrag.
Das Skript für einen ConTract ist als Event-Trigger-Ablauf definiert ähnlich

[1]Die Aussagen dieses Abschnittes gelten unter der Voraussetzung, daß kein Multithreading realisiert wird (vgl. Kap. 1 Abschnitt 1.5).

einem Petri-Netz.[2] Die erfolgreiche Ausführung des Steps *flugbuchung* ist ein Ereignis (Event), das eine Aktion des zuständigen ConTract Manager auslöst („triggert"). Dementsprechend wartet der ConTract Manager nicht nach Versenden aller gleichzeitig aktivierten Steps mithilfe eines *await_answer* auf deren Beendigung, sondern erhält die „Marke" für die Beendigung der Stepausführung in einem Dienstaufruf zugeschickt.
Die Antwort auf den asynchronen Dienstaufruf *flugbuchung* ist demnach folgerichtig ein Dienstaufruf von *step_computation_finished*, womit dem ConTract Manager das Ereignis „Beendigung eines Steps" mitgeteilt wird. Für die Bearbeitung der Antwort existiert also ein eigener Dienst, der vom ConTract Manager angeboten wird.
Die beim Aufruf eines Steps übertragenen Daten sind die jeweils vom adressierten Step benötigten Parameter. Sie können beliebigen Umfang annehmen und von beliebiger Komplexität sein.

## 2.2 Analyse der Kommunikation

**Ausführung**

Es gibt sowohl Aufträge wie *get_gtrid*, die synchron ausgeführt werden können als auch asynchrone Aufträge wie *step_computation_request*.
Der Entwurf für das Kommunikationssystem sieht vor, daß alle Aufrufe auf der Ebene der Basiskommunikationsmechanismen synchron sind in dem Sinne, daß der Sender stets ein Acknowledge des Kommunikationssystems abwartet. Dieses Acknowledge dient als „Garantie"-Karte des Kommunikationssystems für die korrekte Übertragung der Nachricht. Synchrone Aufrufe erhalten die Antwort mit dem Acknowledge zurück.

**Lokalität**

Beim Verschicken von Aufträgen innerhalb des ConTract-Systems lassen sich drei verschiedene Stufen der Lokalität unterscheiden:

**prozeßlokal:** Sender und Empfänger laufen im selben Prozeß, z. B. *get_gtrid* vom ConTract Manager an den Transaction Manager.

**rechnerlokal:** Sender und Empfänger laufen auf demselben Rechner, aber in verschiedenen Prozessen, z. B. eine Nachricht an einen Server auf derselben Sun-Workstation.

**netzübergreifend:** Sender und Empfänger laufen auf verschiedenen Rechnern mit unter Umständen unterschiedlichen Betriebssystemen (SunOS, VMS, GUARDIAN).

**Initiative**

Die Beziehungen zwischen Sendern und Empfängern können klassifiziert werden nach dem Schema:

[2] Dies ist im Beispiel-ConTract im Kap. 1 demonstriert (Abb. 1.2).



**Master/Slave:** Verbindung mit statisch festgelegter, einseitiger Initiative

**Server/Client:** Verbindung mit dynamisch veränderlicher, zu einem Zeitpunkt aber einseitiger Initiative

**Duplex-Verbindung:** Zu jedem Zeitpunkt ist die beidseitige Initiative möglich.

Die meisten Beziehungen zwischen APRICOTS-Komponenten innerhalb der ConTract-Verarbeitung sind Duplex-Verbindungen.
So können sich Resource Manager gegenseitig Nachrichten zusenden wie im Falle des ConTract-Manager, der einem Step Computation Server asynchron Aufträge zur Bearbeitung sendet und von diesem Server Antwortnachrichten mit den Bearbeitungsergebnissen erhält.

**Adressierung**

Hier kann die Empfängerkomponente unbekannt sein wie i. a. beim Aufruf eines Steps. So verfügt der CM in Abb. 1.5 über kein Wissen, von welchen Komponenten die Dienste *flugdateneingabe*, *hotelreservierung* oder *flugbuchung* angeboten werden.
Die Empfängerkomponente ist bekannt bei Aufrufen, die an schon vorher adressierte Resource Manager erfolgen, wie es beim Aufruf von *commit* der Fall ist. Dieser Aufruf ergeht an einen Step Computation Server, der eine Teil-Transaktion erfolgreich ausgeführt hat, für die er vorher einen entsprechenden Dienstaufruf erhalten hat. [3]
Wenn der Adressat unbekannt ist und unspezifiziert bleibt, dann kann es entweder dem Kommunikationssystem anheimgestellt werden, einen Empfänger auszusuchen, oder der Sender möchte einen Hinweis für das Scheduling angeben wie z. B. die Lokalitätsbedingung „an eine Teilkomponente im selben Prozeß" für den Aufruf von *get_gtrid*.

**Antwort**

Die Antwort auf eine Nachricht kann, muß aber nicht an den Sender zurückgeschickt werden. So kann das Voting beim *prepare_to_commit* an jeweils einen an einer Transaktion beteiligten Resource Manager auf jedem Rechner gehen, der wiederum alle anderen Resource Manager, die auf demselben Rechner an der Transaktion beteiligt waren, informiert. Die Antwort aller Resource Manager geht an den verantwortlichen Transaction Manager, der die Transaktion in Gang setzte.

## 2.3 Modell für das Kommunikationssystem

Die Veranschaulichung des Modells für das Kommunikationssystem verzichtet auf eine formale Darstellung und möchte dem Leser und der Leserin nur grundlegende Prinzipien der Nachrichtenübertragung und -verarbeitung nahebringen.

[3] Vgl. hierzu die Schnittstelle eines Dienstaufurfes in Abschnitt 4.1.4.

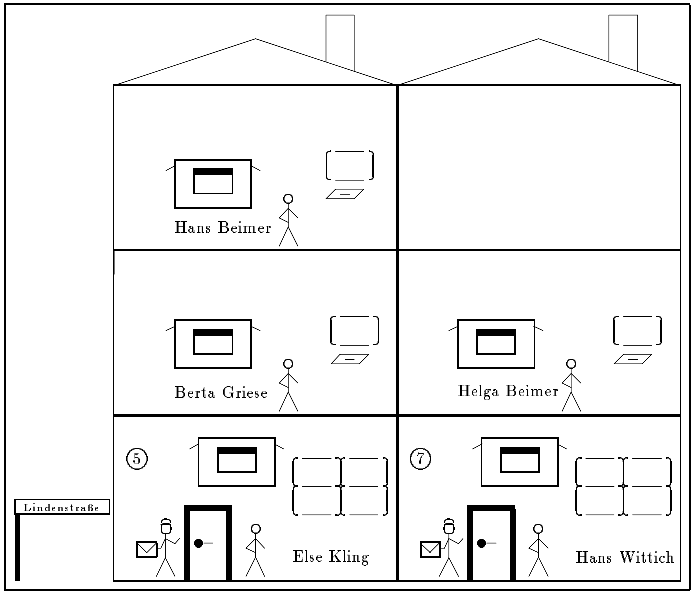


Abbildung 2.1: Postmodell für das Kommunikationssystem

Eigenschaften und Implementierungsaspekte sind in den folgenden Abschnitten sowie in Kap. 4 beschrieben. Beispielhaft und richtungsweisend für das Kommunikationssystem ist die Post. Beispielhaft, indem sie seit geraumer Zeit in punkto Zuverlässigkeit und Schnelligkeit unser Vetrauen zurückgewonnen hat. Richtungsweisend dadurch, daß die Zustellbarkeit von der Anwesenheit des Empfängers unabhängig ist.
Abb. 2.1 zeigt einen einfachen kleinen „Zustellbezirk" für Nachrichten respektive Briefe, der zwei Häuser umfaßt: Lindenstraße 5 und 7. In jedem Haus sitzt ein Briefträger, der keine Freizeit kennt. Rund um die Uhr wartet er auf Post, die aus dem jeweiligen Nachbarhaus gebracht wird. Im Haus Lindenstr. 5 sitzen neben der Hausmeisterin Else Kling mit ihrer Überwachungsanlage (Monitor) zwei arbeitende Menschen.[4] Im Haus Nr. 7 sitzt derzeit neben dem Hausmeister Hans Wittich nur eine Mitarbeiterin, die ebenfalls unter Aufsicht steht.
Fallen der Briefträger oder gar Hans Beimer bzw. Berta Griese durch Krankheit oder Todesfall aus, so sorgt Frau Kling sofort für Ersatz.[5]

[4]Auf der Zeichnung ist leider nicht die raffinierte Verbindung aller Briefkästen in einem Haus über eine Rohrpostanlage zu sehen.
[5]Das gleiche gilt für Hans Wittich (Onkel Hans) und die Beschäftigten im Haus Nr. 7.



Frau Griese hat, wie alle aufgeführten Personen, auf ihrem Schreibtisch ein Fach mit bisher unerledigte Aufgaben. Diese kann sie nach und nach abarbeiten, bis das Fach leer ist. Es können dabei weitere Aufgaben anfallen, die sie wieder in das Fach hineinlegt.
Die Büros in einem Haus sind über Rohrpost miteinander verbunden. Will z. B. Hans Beimer Berta Griese eine Nachricht zukommen lassen, so öffnet er sein Rohr und die Nachricht landet in Frau Grieses Briefkasten. Kann er ohne die Antwort nicht weiterarbeiten, so wartet er, bis Frau Griese ihrerseits die Rohrpost gebraucht, um ihre Antwort in den Briefkasten von Herrn Beimer zu befördern. Will Herr Beimer nicht auf die Antwort warten, dann nimmt er den nächsten Auftrag aus seinem Briefkasten, sobald er seinen Auftrag an Frau Griese in ihren Briefkasten hat plumpsen hören.
Will Herr Beimer seiner Ex-Gemahlin in Haus Nr. 7 eine Nachricht zukommen lassen, so muß er seine Beine bemühen, das Treppenhaus hinuntersteigen, bis zum Nachbarhaus laufen und Onkel Hans seinen Brief übergeben. Herr Wittich quittiert ihm noch den Empfang und Herr Beimer begibt sich wieder in sein Büro zurück.
Er hat nun zwei Möglichkeiten. Entweder greift er sich sofort den nächsten Brief aus seinem Briefkasten oder aber er wartet, ungeduldig auf die Schreibtischplatte trommelnd, auf die Antwort. In beiden Fällen muß Onkel Hans den Brief von Herrn Beimer in Helga Beimers Briefkasten befördern. Helga Beimer macht ihn auf, verfährt nach Anweisung und bringt die Antwort, atemlos zu Frau Kling eilend, in die Lindenstr. 5. Frau Kling quittiert und befördert die Antwort in den Briefkasten von Herrn Beimer, der nun im Wartefall erlöst ist.
Wie unschwer zu erkennen ist, haben Else Kling und Hans Wittich die Rolle des ConTract Processing Monitor zugewiesen bekommen. Herr Beimer ist wie im richtigen Leben Manager und zwar hier der ConTract Manager. Die Frauen haben die untergeordneten Rollen der Step Computation Server erhalten.
Das Kommunikationssystem selbst ist in seinen Grundzügen festgehalten:

- Mitarbeiter und Mitarbeiterinnen (Komponenten) können sich selbst mit Aufträgen versorgen (**bürolokales Aufgabenfach**).
- Die ständige Empfangsbereitschaft aller Teilnehmer und Teilnehmerinnen ist durch den **Briefkasten** garantiert.
- **Alle** Teilnehmer und Teilnehmerinnen in einem Haus (Rechner) sind **untereinander verbunden**.
- Die Kommunikation zwischen den Häusern wird über **Briefträger** vermittelt.
- Die Auftragserteilung erfolgt durch eine **Nachricht**.
- Die **Antwort** erfolgt im asynchronen Fall **durch Benachrichtigung**, die der Auftragnehmer durchführen muß.

Was in diesem Beispiel nicht zur Geltung gebracht wird, ist die Realisierung der Ortstransparenz eines Aufrufs sowie die Adressierung über den Service statt über den Server.

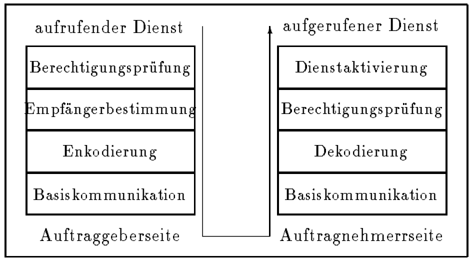


Abbildung 2.2: Schichten der Kommunikation 1: Senden des Auftrags

## 2.4 Aufbau des Kommunikationssystems

Die folgenden Abschnitte betreten wieder formaleres Gelände und beschreiben, was das Kommunikationssystem unterhalb der Schnittstelle zu einem Dienstaufruf leistet.
Wie im Abschnitt 2.1 schon erläutert, wird ein Dienstaufruf vom Kommunikationssystem in Nachrichten , d. h. den Austausch von Nachrichten zwischen zwei Komponenten, umgesetzt. Dabei wird der Auftrag als Nachricht vom Auftraggeber zum Auftragnehmer, die Antwort als Nachricht vom Auftragnehmer zum Auftraggeber geschickt.
Um seiner Aufgabe gerecht zu werden, eine von den physischen Gegebenheiten abstrahierende Schnittstelle für Anwendungen zu bieten, muß das Kommunikationssystem mehrere Funktionen integrieren. Diese Funktionen werden als **Schichten** bezeichnet und sind in Kap. 4 ausführlich beschrieben. Die Schnittstellen der Funktionen, die diese Schichten implementieren, sind im Anhang in Kap. B dokumentiert.
Die Schichten des APRICOTS-Kommunikationssystems lehnen sich an das OSI-7-Schichten-Referenzmodell an.[6] Dieses Modell ist ein allgemein akzeptierter Standard für die Protokolle in verteilten Anwendungen. Seine sieben Schichten bauen aufeinander auf und abstrahieren zunehmend von den Gegebenheiten der Verteilung über ein Netz. Diese Schichten sind in Tab. 2.1 dargestellt. Sie lassen sich in zwei Gruppen teilen:

- Die übertragungsorientierten Schichten dienen der Abstraktion von den physischen Gegebenheiten des Netzes.
- Die datenverarbeitungsorientierten Schichten umfassen die drei oberen Schichten. Sie abstrahieren von den Kommunikationsmechanismen der darunterliegenden Schichten und bieten Ortstransparenz zwischen den Kommunikationspartnern

[6]Ausführliche Darstellungen dieses Modells finden sich u. a. in [17] und [12].



| | | |
|---|---|---|
| Schicht 7 | Anwendung | **Berechtigungsprüfung, Empfängerbestimmung, Dienstaktivierung** |
| Schicht 6 | Darstellung | **Datenkonvertierung** |
| Schicht 5 | Kommunikation | **Basiskommunikation** |
| Schicht 4 | Transport | TCP, UDP |
| Schicht 3 | Vermittlung | IP |
| Schicht 2 | Sicherung | Ethernet |
| Schicht 1 | Bitübertragung | Ethernet |

Tabelle 2.1: OSI-7-Schichten-Referenzmodell

Das Kommunikationssystem setzt auf den übertragungsorientierten Schichten auf und realisiert die datenverarbeitungsorientierten Schichten des OSI-Referenzmodells (siehe Tab. 2.1). Die Realisierung der übertragungsorientierten Schichten ist am Beispiel der Interprozeßkommunikation dargestellt worden. Zu berücksichtigen ist aber, daß das APRICOTS-Kommunikationssystem auch **prozeßlokale** Aufrufe integriert.
Die Kommunikationssteuerschicht steuert den Austausch der Nachrichten auf der Transportverbindung: Aufbau von Kommunikationskanälen, Wechsel der Senderichtung, Wiederanlauf nach Unterbrechungen. Sie ist realisiert in der Basiskommunikationsschicht.
Die Darstellungsschicht legt fest, wie Datentypen kodiert werden, und führt die Konvertierung durch. Diese Aufgabe wird von der Konvertierungsschicht des Kommunikationssystems übernommen.
Die Anwendungsschicht unterstützt verteilt realisierte Anwendungen in mehrerer Hinsicht. Sie wird im Kommunikationssystem in verschiedene Funktionen aufgeteilt: Berechtigungsprüfung und Empfängerbestimmung auf der Senderseite einer Nachricht, Dienstaktivierung auf der Empfängerseite einer Nachricht.
Bei einem Dienstaufruf werden sowohl beim Versenden des Auftrags wie auch beim Zurücksenden der Antwort alle Schichten und zwar jeweils auf Auftraggeber- **und** Auftragnehmerseite durchlaufen (Abb. 2.2 und 2.3).[7]
Die **Berechtigungsprüfung** sichert die Integrität des gesamten Systems im Hinblick auf das Kommunikationssystem. Insbesondere sollen unberechtigte Eingriffe in die Konfiguration wie das Erzeugen bzw. Suspendieren von Komponenten verhindert werden.
Die **Empfängerbestimmung** legt auf der Senderseite eines Dienstaufrufes anhand spezifizierter Angaben aus der oberen Schnittstelle fest, welche Empfängerkomponente den Auftrag erhält. Sie führt damit ein sogenanntes **server service binding** durch, wie es in Cygnus realisiert ist ([8]).[8]
Auf der Empfängerseite enspricht dies der **Aktivierung** des von einer Nach-

[7]Für den prozeßlokalen Fall gilt das mit Einschränkungen, wie in Kap. 4 in Abschnitt 4.3.1 erläutert wird.
[8]Vgl. hierzu auch das Modell der Location Independent Invocation in [6], das von Client-Service-Beziehung spricht.

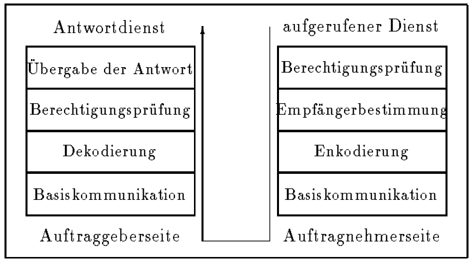


Abbildung 2.3: Schichten der Kommunikation 2: Senden der Antwort

richt adressierten Dienstes bei der Auftragserteilung.
Die **Datenkonvertierung** packt auf der Senderseite die Daten des Dienstaufrufs in das APRICOTS-Übertragungsformat. Auf der Empfängerseite werden die Daten entpackt und dem aktivierten Dienst als Parameter übergeben.
Die **Basiskommunikationsschicht** übernimmt den Aufbau und Abbau von Verbindungen zu den durch die Adreßbestimmungsschicht bestimmten Komponenten und überträgt die Basis mit einem Mechanismus, der durch die Distanz zwischen Sender und Empfänger festgelegt ist. Auf der Empfängerseite führt diese Schicht die Bereitstellung einer eingetroffenen Nachricht aus den Nachrichtenwarteschlangen der Empfängerkomponente durch.

## 2.5 Eigenschaften des Kommunikationssystems

Das Kommunikationssystem vermittelt und verteilt die Dienstaufrufe, die bei der ConTract-Verarbeitung anfallen. Diese Dienstaufrufe erfolgen in der Umgebung eines verteilten, heterogenen Systems. Dem Kommunikationssystem fällt dabei die Aufgabe zu, Anwendungen eine Schnittstelle anzubieten, die der Schicht 7 des ISO-Referenzmodells entspricht (**Anwendungsschicht**).[9]
Das ist eine weitergehende Abstraktion von den physischen Gegebenheiten, als sie das Protokoll des Remote Procedure Calls (RPC) anbietet, das der Kommunikationssteuerschicht angehört. Der RPC ist durch folgende Merkmale gekennzeichnet (nach [5]):

- Die Beziehung zwischen Prozessen ist festgelegt. Die Rollen für Server und Klient sind statisch.
- Der RPC ist eine Erweiterung des Funktionsaufrufs auf nicht-prozeßlokale Fälle und wie ein Funktionsaufruf synchron.

[9] Vgl. dazu dit Tabelle 2.1 im vorherigen Abschnitt.



- Die Adressierung erfolgt unter Angabe des Zielrechners und des Server-Namen.
- Der RPC verlangt auf Klientenseite einen expliziten Verbindungsaufbau und -abbau.
- Der RPC-Code fungiert auf Serverseite als Nachrichtenverteiler. Die vom Server angebotenen Dienste sind Submodule dieses Nachrichtenverteilers.

Das hier vorgestellte Kommunikationssystem weist demgegenüber wesentliche Unterschiede auf.
Die Anlehnung an die Anwendungsschicht des ISO-7-Schichten-Referenzmodells beinhaltet, daß sich Anwendungen einschließlich der Resource Manager selbst, die sich des Kommunikationssystems bedienen, nicht um Fragen der physischen Adressierung, der Lokation anderer Resource Manager, der Art der Verbindung zum Empfänger oder der Konversion der zu übertragenden Daten zu kümmern haben.
Die physische Adressierung und das Übertragungsprotokoll werden möglichst vollständig vor den sendenden und empfangenden Komponenten verborgen. Die ortstransparente Adressierung erfolgt allein mit dem Dienstnamen, nicht mit dem Namen des Servers oder gar des Rechners.
Eine Konsequenz ist die Unabhängigkeit des Anwendungscodes von Rekonfigurationen.[10]
Die Adressierung über logische Namen erfordert im Vergleich zum RPC eine weitere Abstraktionsebene. Der Dienstname muß zunächst in einen Komponentennamen umgesetzt werden (service-server binding), erst danach erfolgt die Bestimmung der physischen Adresse.
Neben der Adressierung über den Dienstnamen existiert aber auch die Möglichkeit, wie bei einem RPC explizit einen Empfänger anzugeben oder Lokalitätsbedingungen (prozeßlokal/nicht prozeßlokal) zu spezifizieren (Abb. 2.4).[11]
Die Integration prozeßlokaler Aufrufe ist im RPC-Konzept wie auch in allen anderen, mir bekannten Kommunikationsmechanismen nicht verwirklicht.

Der Empfang von Nachrichten und Antworten erfolgt über das Kommunikationssystem.
Alle Dienste, die ein Programm anbietet, werden über Nachrichten, die das Kommunikationssystem empfängt, aktiviert. Aus diesem Grund bieten alle Dienste nach oben dieselbe Schnittstelle an. Dies ist analog zu Diensten eines RPC-Servers.
Da alle Komponenten in APRICOTS Dienste anbieten, sind alle Komponenten im Client-Server-Modell auch Server. Wie bei einem RPC-Server spiegelt sich das in der zentralen Rolle des Kommunikationssystems als Nachrichtenverteiler. Alle Resource Manager sind somit Submodule des Kommunikationssystems.
Dies ist wie beim RPC notwendig, um die ständige Empfangsbereitschaft einer Komponente zu garantieren.

[10] Dies ist analog zu Cygnus ([8]).
[11] Vgl. die Beispiele im Abschnitt 2.1.

RM_call

| | |
|---|---|
| Aufrufidentifikation | Nachrichtenhenkel |
| Adreßangaben | Dienstname; callback-Adresse; Name der Resource Manager Klasse; Lokalitätsbedingungen; Empfängeradresse |
| Transaktionsangaben | globaler Transaktionsbezeichner |
| Daten | Datentyp; Länge; Auftragsdaten bzw. Antwortdaten |

Abbildung 2.4: Schnittstelle eines Dienstaufrufes

Im Gegensatz zu einem RPC-Server können jedoch alle Komponenten in APRICOTS selber Aufträge versenden, auch an ihre eigenen Klienten (Duplexverbindung).

Die Schnittstelle zu einem Dienstaufruf ist einfach und flexibel aufgebaut. Es existieren zwei Aufrufe: **RM_call_sync** für synchrone und **RM_call_async** für asynchrone Dienstaufrufe.
Die Schnittstelle zu einem Dienstaufruf ist schematisch in Abb. 2.4 wiedergegeben. Der Ablauf eines synchronen Aufrufs ist analog zu einem RPC: Der Auftraggeber versendet seine Nachricht und wird blockiert, bis die Antwort eintrifft.
Ein wichtiges Kriterium für den Entwurf des Kommunikationssystems ist die Unterstützung des ereignisorientierten Ablaufs bei der ConTract-Verarbeitung (Event-Trigger-Mechanismus).[12]
Ein RPC bietet nur eine synchrone Schnittstelle an. Es gibt zwar mittlerweile Erweiterungen für einen asynchronen RPC (ASTRA [1], Promises [14]), jedoch sind diese nicht ereignisorientiert. Der Auftraggeber muß selbst ein blockierendes *await_answer()* ausführen, um die Antwort zu erhalten.
Der asynchrone Aufruf in APRICOTS zieht dagegen kein Warten nach sich: Die Antwort auf einen asynchronen Aufruf erfolgt ebenfalls durch einen Dienstaufruf an die Komponente des ursprünglichen Auftraggebers, also durch Benachrichtigung seitens des Auftragnehmers. Dafür muß der Aufrufer eines asynchronen Auftrags den Namen desjenigen Dienstes angeben, der die Antwort erhalten soll. Das ist die sogenannte **callback**-Adresse in Abb. 2.4.

Die Antwort des Empfängers erfolgt im synchronen wie in asynchronen Fall über einen Aufruf von *RM_call_return*, sodaß es für den aufgerufenen Dienst transparent bleibt, ob er synchron oder asynchron aktiviert wurde.

Die zu übertragenden Daten werden dem Kommunikationssystem als Byte-String übergeben, um die Unabhängigkeit des Kommunikationssystems von Änderungen in der Struktur der zu übertragenden Daten zu bewahren. Der

[12] Vgl. dazu das Beispiel für eine Step-Aktivierung in Abschnitt 3.3.2.



Parameter für den Datentyp (Abb. 2.4) dient dabei als Schlüssel zur entsprechenden Konvertierungsroutine, die die Konvertierungsschicht des Kommunikationssystems benötigt.
Diese Konvertierungsroutinen sind in einer dynamisch zu ladenden Bibliothek aufzubewahren.[13]

Der Aufbau einer „Session" zwischen zwei Komponenten wird von der Schnittstelle ermöglicht, indem ein bereits vorher aufgerufener Empfänger bei einem Dienstaufruf angegeben werden kann.[14] Dadurch lassen sich intern in den Schichten des Kommunikationssystems eine erneute Adreßrecherche, ein neuer Verbindungsaufbau ebenso wie das Versenden umfangreichen Kontextes zwischen neuen Kommunikationspartnern vermeiden.

Eine Nachricht wird auch bei kurzfristigem Ausfall einer der beiden beteiligten Komponenten, d. h. Prozeß- bzw. Rechnerabsturz mit anschließendem Recovery oder Migration, ohne zusätzliche Sicherungsmaßnahmen zugestellt. Dadurch werden Anwendungen des Kommunikationssystems von der Kenntnis der Details der zugrundeliegenden Kommunikationssteuerschicht wie einem Neuaufbau der Verbindung entlastet und die Ortstransparenz der Adressierung aufrechterhalten.
Auch muß eine Komponente erreichbar sein, ohne daß sie in einer „busy-wait"-Schleife auf eintreffende Nachrichten wartet, um unnötige Wartezeiten für den Sender einer Nachricht zu vermeiden. Aus diesem Grund verfügen alle Komponenten in APRICOTS über eine Mailbox, die für andere Komponenten ständig empfangsbereit ist.
Fehlertoleranz und ständige Empfangsbereitschaft im Sinne von quittierter Nachrichtenübergabe werden von einem RPC nicht realisiert.

Entsprechend der in [11] vorgeschlagenen Prinzipien ist das Kommunikationssystem im Sinne des **Transactional Remote Procedure Call** (TRPC) implementiert. Die Konsequenzen für das Kommunikationssystem sind:

- Jeder Dienstaufruf muß als TRPC mit einem Transaktionsbezeichner versehen werden (Abb. 2.4).
- Jeder Resource Manager, der einen TRPC erhält, partizipiert an der spezifizierten Transaktion und wird vom zuständigen Transaction Manager kontrolliert.[15] Das Kommunikationssystem muß diesem Transaction Manager also die Adressen der beteiligten Resource Manager bereitstellen.
- Das Kommunikationssystem muß die Erhaltung aller transaktionsrelevanten Information sichern.[16]

[13] Diese Bibliothek für Routinen ist bislang nur in der Entwurfsphase. Für die Kommunikation zwischen Rechnern ist sie jedoch notwendig.
[14] Der aufrufende Dienst erhält die vom Kommunikationssystem ermittelte Empfängeradresse zurück, vgl. 4.1.4.
[15] Das bezieht sich in erster Linie auf Entscheidungen über ein *abort* oder *commit*.
[16] In Kap. 6 wird beschrieben, wie ein Dienstaufruf „recoverable" gemacht werden kann.

- Das Kommunikationsprotokoll muß das Transaktionsprotokoll angemessen unterstützen.

# Kapitel 3

# Ausführung eines Resource Manager Calls

## 3.1 Aufbau einer APRICOTS-Komponente

Zum Verständnis der folgenden Abschnitte ist es nützlich und hilfreich, sich den Aufbau einer Komponente in APRICOTS (vgl. Kap. 1) zu vergegenwärtigen und die Schnittstelle für einen Resource Manager Call, wie sie in Abb. 2.4 beschrieben ist, zu rekapitulieren.
Eine Komponente ist ein aktivierter Prozeß in APRICOTS, der den Code für die Dienste von ein oder mehreren Resource Managern sowie den Code des Kommunikationssystems umfaßt.
Der Aufbau ist in Abb. 3.1 am Beispiel der Komponente *contract_comp* aus der Beispielkonfiguration von Abb. 1.5 verdeutlicht, die den ConTract Manager und den Transaction Manager zusammenfaßt. Jeder der beiden Resource Manager bietet dabei mehrere Dienste an. Wenn eine Komponente gestartet wird, initialisiert das Kommunikationssystem zunächst seine Datenstrukturen und geht danach in eine Endlosschleife (Abb. 3.2).[1] In dieser Schleife wird gewartet, bis eine Nachricht eintrifft. Eine eingetroffene Nachricht wird auf ihre Berechtigung hin überprüft und bei positivem Ergebnis dekodiert. Anschließend wird die Adresse des Empfängerdienstes ermittelt und dieser sodann aktiviert. Der aktivierte Dienst bearbeitet die Nachricht, führt entsprechende Aktionen aus und bereitet schließlich die Antwort vor, die mit *RM_call_return* angestoßen wird.
Bei einer asynchronen Dienstaktivierung wird im Makro *RM_call_return* ein asynchroner Dienstaufruf *RM_call_async* mit anschließendem *return*, bei einer synchronen Aktivierung nur ein *return* ausgeführt.
Ist der Dienstaufruf synchron, so wird die Antwort **nach** der Dienstausführung an den Auftraggeber zurückgeschickt.
Danach kehrt die Routine *main* wieder an den Anfang der Nachrichtenverteilerschleife zurück.

[1] Vgl. hierzu den Anfang des Kap. 4 mit der Abb. 4.2.

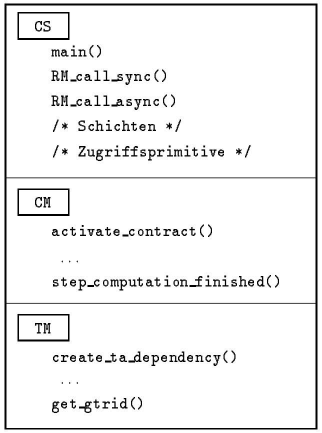


Abbildung 3.1: Aufbau der Komponente *contract_comp*

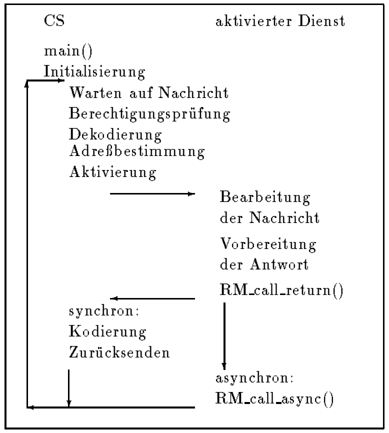


Abbildung 3.2: Kontrollfluß in einer APRICOTS-Komponente



## 3.2 Ablauf bei einem Resource Manager Call

Grundsätzlich sind synchrone Resource Manager Calls von asynchronen zu unterscheiden:

**Synchrone Resource Manager Calls** verhalten sich wie einfache Funktionsaufrufe in einem sequentiellen Kontrollfluß und können Rückgabeparameter enthalten. Sie verwenden für die Nachricht und die Antwort denselben Puffer (vgl. [10], [18]).

**Asynchrone Resource Manager Calls** erhalten lediglich eine Garantie des Kommunikationssystems zurück, daß der Resource Manager Call an die Empfängerkomponente überbracht wird. In der Folge wird der Empfänger aktiviert und sendet die Antwort selbst wieder in einem asynchronen Resource Manager Call an die ursprüngliche Senderkomponente zurück, jedoch an einen eigenen Dienst zur Bearbeitung der Antwort.
Beide beteiligten Komponenten sind demnach wechselweise Sender und Empfänger. Deshalb ist es ist es angemessen, statt von Sender und Empfänger eines Resource Manager Calls von Auftraggeber und Auftragnehmer zu sprechen.

Aus den vorherigen Betrachtungen läßt sich eine Phasenanalyse eines Auftrags innerhalb des APRICOTS-Systems erstellen:

1. Der Auftrag wird vom Auftraggeber in Form eines Resource Manager Calls versendet.
2. Der Auftrag wird vom Auftragnehmer in seinen Parametern, dem „Service Call Interface", entgegengenommen und bearbeitet.
3. Die Antwort, d.h. das Ergebnis der Auftragsbearbeitung wird vom Auftragnehmerdienst am Ende seiner Verarbeitung zurückgegeben.
4. Die Antwort wird vom Auftraggeber entgegengenommen und weiterverarbeitet.

Bei synchronen Aufträgen werden die erste und die vierte Phase von demselben Dienst ausgeführt, bei asynchronen Aufträgen können die ausführenden Dienste der ersten und vierten Phase verschiedene Dienste innerhalb derselben Komponente sein.
Bei synchronen Aufträgen ist nur der Auftragnehmer initiativ und wartet auf die Übertragung der Antwort durch den Auftragnehmer. Bei asynchronen Aufträgen ist der Auftragnehmer in der dritten Phase selbst initiativ, indem er einen asynchronen Diensataufruf ausführt.
Im folgenden werden wegen der grundlegenden Unterschiede zwischen synchronen und asynchronen Resource Manager Calls, vor allem auf der Seite des Auftraggebers, beide jeweils separat vorgestellt.

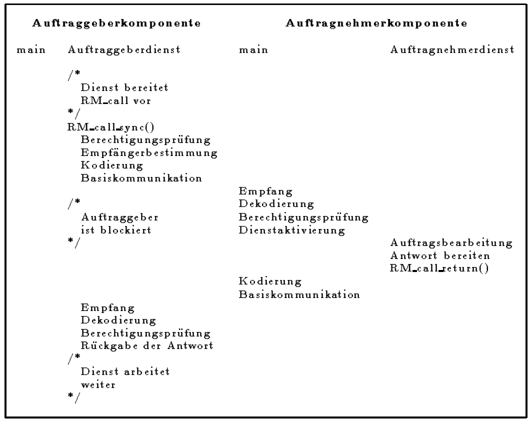


Abbildung 3.3: Kontrollfluß bei einem synchronen Resource Manager Call

### 3.2.1 Ablauf bei einem synchronen Resource Manager Call

Der Kontrollfluß bei einem synchronen Resource Manager Call ähnelt dem Ablauf bei einem Remote Procedure Call. Die Abb. 3.3 zeigt den zeitlichen Verlauf bei einem synchronen Dienstaufruf. Die jeweils zu einem Zeitpunkt aktiven Funktionen sind mit Angaben zu ihrer Tätigkeit versehen.
Der Sender wartet synchron und arbeitet weiter, wenn er vom Empfänger die Antwort erhalten hat. Zu beachten sind hier folgende Merkmale:

- Der Auftraggeber wartet synchron, d.h. er ist blockiert, bis der Auftragnehmer den Resource Manager Call vollständig bearbeitet und die Antwort zurückgesendet hat.
  Jedoch besteht die Möglichkeit, den Auftrag zu stornieren, wenn er nach einem festgelegten Zeitintervall noch immer nicht vom Empfänger bearbeitet worden ist.

- Das Senden des Dienstaufrufs und das Empfangen der Antwort werden von demselben Dienst in der Auftraggeberkomponente durchgeführt.
  Die Adresse der Antwort ist identisch mit dem Absender des Auftrags.

- Die Antwort wird deshalb vom Kommunikationssystem beim Aufruf von *RM_call_return* auch nicht in einen *RM_call_sync* umgesetzt.
  Der *RM_call_sync* empfängt die Antwort des Auftragnehmers selbst und leitet sie an den Auftraggeber weiter. Die Funktion *RM_call_sync* wirkt hier als selektiver Nachrichtenverteiler.[2]

[2]Vergleiche hierzu die Beschreibung von *RM_call_sync* in Kap. 4.



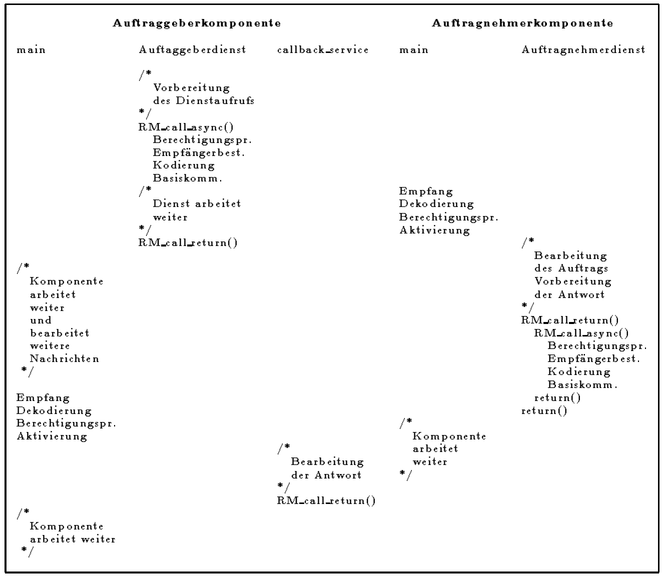


Abbildung 3.4: Kontrollfluß bei einem asynchronen Resource Manager Call

  Für die Antwort wird der vom Auftraggeber bereitgestellte Puffer benutzt.

- Der Auftraggeber ist zwar in seinem Kontrollfluß blockiert, jedoch können andere Komponenten Nachrichten in seinem Briefkasten hinterlegen.

### 3.2.2 Ablauf bei einem asynchronen Resource Manager Call

Für einen asynchronen Dienstaufruf ergibt sich ein in wesentlichen Punkten anderer Kontrollfluß. Abb. 3.4 zeigt analog zu Abb. 3.3 den zeitlichen Verlauf.
Mit dem Aufruf von *RM_call_async* wird der Auftrag in der Basiskommunikationsschicht bei der Auftragnehmerkomponente in eine Nachrichtenwarteschlange eingefügt. Anschließend arbeitet der Auftraggeberdienst weiter und beendet seine Tätigkeit durch den Aufruf von *RM_call_return*. Im Unterschied zu einem synchronen Resource Manager Call fallen folgende Merkmale ins Auge:

- Während die Auftraggeberkomponente weitere Nachrichten aus ihren Eingabewarteschlangen bearbeitet, wird die Auftragnehmerkomponente den Auftrag in ihrer Nachrichtenwarteschlange vorfinden und den Empfängerdienst aktivieren.[3]

- Der Auftragnehmerdienst übergibt die Antwort in einem *RM_call_return*, der vom Kommunikationssystem in einen *RM_call_async* umgesetzt wird.
- Die Bearbeitung der Antwort erfolgt durch einen eigenen speziellen Dienstaufruf des Auftraggebers.
  Dieser Dienst zur Bearbeitung der Antwort hat einen eigenen Namen, der vom Auftraggeber bei Auftragserteilung im Resource Manager Call als Parameter *callback* mitgegeben wird.[4]

## 3.3 Beispiele

In diesem Abschnitt wird an zwei Dienstaufrufen exemplarisch gezeigt, wie die Abwicklung synchroner und asynchroner Resource Manager Calls verläuft.
Grundlage der Beispiele sind das in Kap. 1 vorgestellte Skript (Abb. 1.1) sowie die im selben Kapitel gezeigte APRICOTS-Konfiguration (Abschnitt 1.3.1). Die gewählten Dienstaufrufe sind in Kap. 2 vorgestellt worden. Es handelt sich um die Dienste *get_gtrid* und *flugbuchung*. Die Konfiguration aus Abb. 1.5 ist für die Beispiele auf die hier benötigten Komponenten reduziert und um die Dienste der Resource Manager erweitert worden (Abb. 3.5).
Der Verarbeitungsstand des ConTracts ist wie folgt: Der ConTract Manager *contract_comp* auf dem Rechner *agamemnon* ist mit dem in Abb. 3.6 gezeigten Dienst *activate_contract* aktiv.
Die ConTract-Id ist bereits berechnet und an den ConTract Client übergeben worden.Die Flugdateneingabe ist bereits abgeschlossen.
Der ConTract Manager benötigt nun für die beiden parallel ausführbaren Steps *flugbuchung* und *hotelreservierung*, die beide unter Transaktionsschutz laufen, je einen neuen Transaktionsbezeichner vom Transaction Manager. Der Aufruf des Dienstes *get_gtrid* wird im Abschnitt 3.3.1 detailliert beschrieben.
Nach Erhalt der Transaktionsbezeichner trägt der ConTract Manager beim Transaction Manager die Transaktionsbeziehungen zwischen den beiden Steps ein. Anschließend können die Steps über einen Dienstaufruf aktiviert werden. Der Aufruf von *flugbuchung* wird in Abschnitt 3.3.2 erläutert.

Die Abbildung 3.6 gibt den Dienst *activate_contract* in Pseudocode wieder. Die Beispiele stellen den realen Ablauf verkürzt dar. So wird die Verwaltung der Transaktionsbeziehungen z. B. nicht berücksichtigt.

### 3.3.1 *get_gtrid* als synchroner Dienstaufruf

Wie in Abschnitt 2.1 bereits festgestellt, ist *get_gtrid* ein synchroner Aufruf, der geeigneterweise an den prozeßlokal verfügbaren Transaction Manager geschickt

[3] Bei Identität von Auftraggeber- und Auftragnehmerkomponente ist selbstverständlich keine parallele Aktivität möglich.
[4] Vgl. Abb. 2.4.



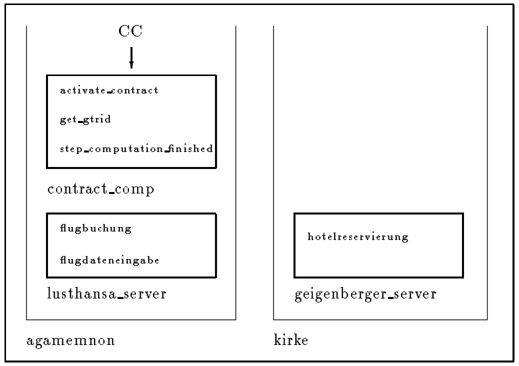


Abbildung 3.5: APRICOTS-Beispielkonfiguration

```
activate_contract( ... )
{
   /* Skript einlesen und ConTract-Id generieren */
           ...
   /* Flugdateneingabe */
   /* Steps flugbuchung und hotelreservierung triggern */
           ...
   RM_call_sync(GET_GTRID,TM,pto_ta_id_buffer1) ;
   RM_call_sync(GET_GTRID,TM,pto_ta_id_buffer2) ;
           ...
   /* Eintragung der Transaktionsbeziehung */
           ...
   RM_call_async(FLUGBUCHUNG,LUSTHANSA,
                 STEP_COMPUTATION_FINISHED,
                  "nicht prozesslokale",pto_step_buffer1) ;
   RM_call_async(HOTELRESERVIERUNG,GEIGENBERGER,
                  STEP_COMPUTATION_FINISHED,
                  "nicht prozesslokale",pto_step_buffer2) ;
           ...
   RM_call_return(...) ;
}
```

Abbildung 3.6: Ablauf für *activate_contract*

werden sollte. Entsprechend sendet der ConTract Manager einen *RM_call_sync* und belegt den Parameter *local* mit dem Wert „prozeßlokal". Der Dienstname ist *GET_GTRID* und der Name der Resource Manager Klasse ist *TM*.
*activate_contract* stellt einen Puffer zur Verfügung, der groß genug ist, den Transaktionsbezeichner aufzunehmen, und der Aufruf wird abgeschickt.
*RM_call_sync* überprüft als erstes die Berechtigung der aufrufenden Komponente *contract_comp*, das Kommunikationssystem zu benutzen (Berechtigungsprüfungsschicht).
Die Empfängerbestimmung erfolgt zunächst in der prozeßlokalen Einsprungadressentabelle. Das Ergebnis ist, daß als Empfängerkomponente ebenfalls *contract_comp* bestimmt wird, da *contract_comp* auch den Code für die Resource Manager Klasse *TM* enthält.
Nun erfolgt eine Abfrage in *RM_call_sync*, ob der Aufruf prozeßlokal ist. Da er es in diesem Fall ist, wird die Optimierungsvariante für synchrone, prozeßlokale Aufrufe eingesetzt.[5] Das bedeutet, daß beim Senden an den Empfänger keine Kodierung in das APRICOTS-Übertragungsformat und beim Empfang keine Dekodierung aus demselben Format erfolgt.
So geht der Aufruf direkt in der Schicht der Basiskommunikation weiter. Es wird die Einsprungadresse bestimmt und der Dienst *get_gtrid* aktiviert.
*get_gtrid* generiert einen neuen Transaktionsbezeichner und belegt den Nachrichtenpuffer mit dessen Wert. Anschließend erfolgt der Aufruf *RM_call_return*, womit der Kontrollfluß zum Kommunikationssystem, also hier in die Routine *RM_call_sync*, zurückkehrt.
Da der Aufruf von *RM_call_sync* prozeßlokal ist, bedarf es keiner Dekodierung und Berechtigungsprüfung. Die Antwort wird dem auftraggebenden Dienst direkt im Nachrichtenpuffer zurückgegeben.
Mit dem neuen Transaktionsbezeichner kann der Dienst *activate_contract* nun weiterarbeiten. So werden nun die Transaktionsbeziehungen zwischen den Steps *flugbuchung* und *hotelreservierung* beim Transaction Manager in das Transaktionsbeziehungsnetz (TBN) eingetragen.[6]

### 3.3.2 Step-Aktivierungen als asynchrone Dienstaufrufe

Nachdem die Transaktionsbezeichner festgelegt sind und ihre gegenseitige Abhängigkeit in das TBN eingetragen ist, können die Steps *flugbuchung* und *hotelreservierung* gestartet werden.
Nach dem in Kapitel 2 erarbeiteten Kriterien sind dabei folgende Bedingungen zu berücksichtigen:

1. Um echte Parallelität zu ermöglichen, werden die beiden Steps durch asynchrone Dienstaufrufe aktiviert.

2. Um die Sequentialisierung in Anbetracht des Single-Threadings innerhalb einer APRICOTS-Komponente zu vermeiden, werden die Dienstaufrufe mit der Lokalitätsbedingung „nicht prozeßlokal" durchgeführt.

[5]Diese Optimierung wird in Abschnitt 4.3.1 beschrieben.
[6]Dies ist ebenfalls ein prozeßlokaler, synchroner Dienstaufruf.



Entsprechend werden der Dienstaufruf *RM_call_async* gewählt und der Parameter *local* mit dem Wert „nicht prozeßlokal" belegt. Die Ortstransparenz des Dienstaufrufs bleibt trotzdem gewahrt, da kein Server als Empfänger, sondern lediglich ein Hinweis für die Empfängerbestimmung angegeben wird.
Die Größe des Nachrichtenpuffers ist Step-spezifisch. Der Parameter *gtrid* wird mit dem von *get_gtrid* gelieferten Transaktionsbezeichner belegt. Für den Parameter *service* erhalten die Dienstaufrufe die logischen Namen der Dienste, also *FLUGBUCHUNG* und *HOTELRESERVIERUNG*, übergeben. Der Antwortdienst wird mit seinem Namen *STEP_COMPUTATION_FINISHED* spezifiziert.
Ich erläutere im weiteren nur die Vorgehensweise für den Aufruf von *flugbuchung*. Für den Aufruf von *hotelreservierung* ist der Ablauf durch die Zwischenschaltung eines Briefträgers für Auftrag und Antwort etwas ausgedehnter. Statt des direkten Schreibens in die Mailbox der Zielkomponente bei *flugbuchung* erfolgt jedesmal eine Übertragung an den Briefträger auf dem Zielrechner. Der Briefträger schreibt die Nachricht danach in die Mailbox der Zielkomponente.[7]

Zunächst wird die Berechtigungsprüfung unternommen und festgestellt, daß *contract_comp* das Kommunikationssystem benutzen darf.
Die Adreßbestimmung erfolgt, da „nicht prozeßlokal" vom Aufrufer vorgegeben ist, **nicht** in der Einsprungadressentabelle der Komponente, sondern im prozeßlokalen Adreß-Cache. Falls dort kein Eintrag für den Dienst *FLUGBUCHUNG* zu finden ist, muß eine Anfrage beim Name Service erfolgen.
Nach der Adreßbestimmung, die für den Empfänger eine entfernte Komponente ergab, muß die Nachricht in das APRICOTS-Übertragungsformat gebracht werden. Dazu wird die Nachricht zusammen mit der für das Kommunikationssystem notwendigen Information in eine neugenerierte Datenstruktur eingefügt.
Danach erfolgt die Übertragung an die Empfängerkomponente, die in unserem Fall einer Eintragung in die Mailbox der Komponente *lusthansa_server* entspricht. Wenn die Eintragung erfolgreich war, kehrt *RM_call_async* zum aufrufenden Dienst *activate_contract* zurück.
Nachdem *activate_contract* auch den Dienstaufruf für *hotelreservierung* abgesetzt hat, führt es selbst ein *RM_call_return* aus.
Während nun die Komponenten *contract_comp* und *lusthansa_server* in ihren Nachrichtenverteilerschleife arbeiten, wird der Auftrag, den Step *flugbuchung* auszuführen, früher und hoffentlich nicht später aus der Mailbox geholt und von *lusthansa_server* dekodiert und überprüft.
Die Adresse des Dienstes *flugbuchung* wird anhand der Einsprungadressentabelle bestimmt und der Dienst aktiviert.
Nach Beendigung des Steps erfolgt ein *RM_call_return*, das, da es sich um eine asynchrone Dienstausführung handelt, in einen *RM_call_async* umgesetzt wird. Der Empfängerdienst wird aus der callback-Adresse entnommen und an den Dienstaufruf übergeben. Das bedeutet, daß ein vollständig spezifizierter Adreßkontrollblock an das Kommunikationssystem übergeben wird, da die Empfängerkomponente bekannt ist und kein anderer Empfänger als der Auftraggeber

[7]Zur Übertragung zwischen Rechnern enthält das Kap. 4 eine ausführliche Beschreibung.

des Steps durch das Kommunikationssystem bestimmt werden soll.
Die Berechtigungsprüfung und Kodierung verlaufen wie bei der Auftragsvergabe durch *activate_contract*. Die Empfängerbestimmung ist durch den spezifizierten Adreßkontrollblock abgekürzt, wie gerade angeführt. Die Basiskommunikation verläuft analog zur Auftragsvergabe: die Nachricht wird in die Mailbox der Komponente *contract_comp* eingetragen.
Der *lusthansa_server* kehrt von der Dienstaktivierung zurück und geht wieder in seine Nachrichtenverteilerschleife.
*contract_comp* findet die Antwort in seiner Mailbox vor, prüft ihre Berechtigung und dekodiert sie.
Dann wird mithilfe der Einsprungadressentabelle der Empfängerdienst *step_computation_finished* aktiviert und die Antwort kann verarbeitet werden.

An diesem Beispiel zeigt sich zweierlei:

1. Die Asynchronität eines Aufrufs ermöglicht echte Parallelität und beschleunigt somit die Abwicklung eines ConTracts, wie die parallele Ausführung der Dienste *flugbuchung* und *hotelreservierung* zeigt, die beide als Datenbankoperationen aufwendig sind und bei sequentieller Ausführung die Abwicklung des ConTracts verlängern würden.

2. Die Ereignisorientierung des asynchronen Dienstaufrufs, d. h. die Antwort durch Benachrichtigung, verhindert, daß eine Komponente durch ein blokkierendes *await_answer* letztlich synchron warten muß.

# Kapitel 4

# Realisierung

Dieses Kapitel stellt den Aufbau und die Funktionsweise des APRICOTS-Kommunikationssystems unter Implementierungsaspekten vor.
Ausgehend vom Aufbau einer APRICOTS-Komponente und eines Dienstes wird die obere Schnittstelle des Kommunikationssystems vorgestellt.
Daran schließt sich eine nähere Betrachtung der Schichten an, aus denen sich das Kommunikationssystem zusammensetzt.
Abschließend werden die Adressierung und die Adreßumsetzung beschrieben.

## 4.1 Aufbau des Kommunikationssystems

### 4.1.1 Aufbau einer APRICOTS-Komponente

Wie schon in Abschnitt 1.4 ausgeführt, besteht eine APRICOTS-Komponente aus dem Kommunikationssystem und ein oder mehreren Resource Managern. Die Resource Manager implementieren wiederum einen oder mehrere Dienste.
Abb. 4.1 zeigt den Aufbau einer APRICOTS-Komponente. Der Code des Kommunikationssystems umfaßt die Routine *main* sowie die Primitive für einen Dienstaufruf: *RM_call_sync, RM_call_async* und *RM_call_return*. Diese Primitive werden im Unterabschnitt 4.1.4 erläutert. Die Funktionen zur Realisierung der Schichten des Kommunikationssystems sind nicht in der Abbildung aufgeführt.
Die Komponente *lusthansa_server* aus der Beispielkonfiguration in Abb. 1.5 besteht aus zwei Resource Managern, deren Namen mit *LUSTHANSA* und *TM* angegeben ist. Die Resource Manager Instanz *LUSTHANSA* bietet u. a. die Dienste *flugdateneingabe* und *flugbuchung* an, *TM* bietet neben weiteren die Dienste *get_gtrid* und *create_ta_dependency* an.
Der Kontrollfluß innerhalb einer Komponente bewegt sich wie bei einem RPC-Server in einer Nachrichtenverteilerschleife, wie es die *while*-Schleife in Abb. 4.2 zeigt. Diese Schleife kann nur durch eine Suspendierungs- oder eine Shutdown-Anforderung des ConTract Processing Monitor verlassen werden.

```
/* CS */
   main()
   RM_call_async()
   RM_call_sync()
   RM_call_return()

/* TM */
   get_gtrid()
       ...
   create_ta_dependency()

/* LUSTHANSA */
   flugdateneingabe()
       ...
   flugbuchung()

lusthansa_server
```

Abbildung 4.1: Aufbau der Komponente *lusthansa_server*

```
main()
{
  init()
  while true do
    lookup_message()
    decode_message()
    authorizate_message()
    activate_service()
    if message synchron do
      encode_answer()
      transmit_answer()
    endif
  endwhile
}
```

Abbildung 4.2: Nachrichtenverteilerschleife in APRICOTS



<service_name>

| | |
|---|---|
| &pto_message_handle | Zeiger auf den Henkel des Dienstaufrufes |
| &callee | Adreßkontrollblock des aufgerufenenen Dienstes |
| &gtrid | globaler Transaktionsbezeichner |
| &data_type | Datentyp der übergebenen Struktur |
| &message_length | Länge der übergebenen Struktur |
| &message_string | eigentliche Auftragsdaten |

Abbildung 4.3: Schnittstelle eines APRICOTS-Dienstes

### 4.1.2 Nachrichtenempfang und -verteilung

Das Kommunikationssystem übernimmt die Verteilung der Nachrichten an die adressierten Dienste innerhalb einer Komponente über die Routine *main*, wie es in Abb. 4.2 dargestellt ist.
*main* führt beim Start einer Komponente die Initialisierung der Datenstrukturen zur Verwaltung der Kommunikation durch und geht dann in die Nachrichtenverteilerschleife, eine Endlosschleife, in der ankommende Nachrichten empfangen und weitergeleitet werden (*while*-Schleife).
Trifft eine Nachricht an den Eingangswarteschlangen der Komponente[1] ein, so wird die Nachricht aus dem APRICOTS-Übertragungsformat entpackt. Anschließend wird eine Berechtigungsprüfung vorgenommen und, bei Erfolg, der adressierte Dienst bestimmt und aktiviert.
Ist der Dienstaufruf synchron, so übernimmt *main* nach Beendigung des Dienstaufrufs die Übersendung der Antwort an den Auftraggeber.[2]

### 4.1.3 Schnittstelle eines APRICOTS-Dienstes

Ein APRICOTS-Dienst bietet für synchrone als auch asynchrone Aufrufe dieselbe Schnittstelle an (vgl. Abb. 4.3).
Der Zeiger *pto_callee* dient zur Identifikation des Auftrags und wird mit der Antwort wieder zurückgeschickt. *gtrid* enthält den globalen Bezeichner der Transaktion, unter deren Schutz der Dienstaufruf ausgeführt wird.
Die übrigen Parameter sind in den folgenden Abschnitten mit den Schnittstellen eines Dienstaufrufes beschrieben.
Abb. 4.4 zeigt den Schemaaufbau eines Dienstes. Nach Empfang eines Auftrags wird dieser bearbeitet und die Antwort in einem *RM_call_return* zurückgesendet.
Die Antwort wird jedoch je nach Art der Synchronität entweder an denselben Dienst des Auftraggebers (synchron) oder an einen anderen (asynchron) geschickt. Diese Differenzierung wird vom Kommunikationssystem innerhalb des

[1] Das sind für prozeßlokale Nachrichten eine Queue und für Nachrichten von anderen Komponenten eine Mailbox, siehe Kap. 5.
[2] Eine Ausnahme bildet der synchrone prozeßlokale Aufruf, vgl. Abschnitt 4.3.1.

```
CS_RETURNCODE  flugbuchung  (pto_msg_handle, callee, gtrid,
                             data_type, msg_len, msg_buf)
{
    ...
    /*
    Bearbeitung der Eingangsdaten aus msg_buf
    ...
    Ermittlung und Buchung eines Flugs unter Transaktionsschutz
    ...
    Übertragung der Flugdaten in msg_buf
    */
    RM_call_return(msg_handle,data_type,msg_len,msg_buf) ;
}
```

Abbildung 4.4: Aufbau des Dienstes *flugbuchung*

Aufrufs von *RM_call_return* durchgeführt.

### 4.1.4 Die obere Schnittstelle des Kommunikationssystems

Die Schnittstelle zu den APRICOTS-Anwendungen ist an die Schicht 7 des OSI-Referenzmodells angelehnt.
Es existieren drei Primitive zur Realisierung eines Resource Manager Calls (Dienstaufrufs):

**RM_call_sync()** für die synchrone Ausführung eines Dienstes

**RM_call_async()** für die asynchrone Ausführung eines Dienstes

**RM_call_return()** für die Rückkehr von einem Dienstaufruf

*RM_call_sync* und *RM_call_async* werden vom Auftraggeber eines Dienstaufrufes verwendet, um die Ausführung des Auftrags anzustoßen. Die Schnittstelle zur Durchführung eines Dienstaufrufes umfaßt nur wenige Primitive, um die Programmierung einfach zu halten. Andererseits wurden die Aufrufe für synchrone und asynchrone Dienstaufrufe getrennt, um der unterschiedlichen Semantik von asynchronem und synchronem Dienstaufruf gerecht zu werden. Darüberhinaus unterscheiden sich beide Primitive im Parameter *callback*, der nur für den asynchronen Aufruf benötigt wird. Er spezifiziert den Namen des Dienstes, der die Antwort des Aufrufs erhalten soll.
*RM_call_return* wird vom Auftragnehmer eines Dienstaufrufes, d. h. dem ausführenden Dienst aufgerufen, um die Ausführung des Dienstes zu beenden und die Übermittlung der Antwort einzuleiten.
Ein Primitiv für das Warten auf die Antwort eines asynchronen Dienstaufrufes wird nicht benötigt, da diese Antwort selbst wieder als Dienstaufruf an den



Auftraggeber realisiert ist (ereignisorientierte Nachrichtenübermittlung).[3]
Ruft der ConTract Manager aus der Komponente *contract_comp* (Abb. 1.5) den Dienst *flugbuchung* asynchron auf, so sendet er mit den Daten auch die callback-Adresse mit, wie es in Abb. 3.6 zu sehen ist. Da es sich bei *flugbuchung* um einen Step handelt, ist der Dienst, der die Antwort erhält, *step_computation_finished*. Dieser Dienst wird vom ConTract Manager angeboten. Wenn der Dienst *flugbuchung* seine Ausführung beendet, wird *RM_call_return* in einen asynchronen Dienstaufruf an *step_computation_finished* umgesetzt.
Primitive zum Verbindungsaufbau bzw. -abbau, zur Adreßumsetzung, zur Fehlerbehandlung bei Störungen der Kommunikationskanäle sowie zur Bereitstellung von Resourcen werden an der oberen Schnittstelle nicht benötigt, da die obere Schnittstelle der Anwendungsschicht des OSI-7-Schichten-Referenzmodells entspricht.[4]
Die Unabhängigkeit des Dienstaufrufs von den physischen Gegebenheiten der Kommunikation wie Übertragungsmechanismus und Ort des Empfängers ermöglicht es, den Code der Resource Manager unabhängig von den Anwendungen zu implementieren und interne Änderungen im Kommunikationssystem durchzuführen, ohne daß die Resource Manager davon betroffen werden.
Der Code von Step Computation Servern und Teilkomponenten wie ConTract Manager und Transaction Manager ist dadurch unabhängig von der tatsächlichen Konfiguration des Systems und den zugrundeliegenden Kommunikationsmechanismen und ist insbesondere stabil gegenüber einer Erweiterung von APRICOTS auf ein heterogenes verteiltes System. Ausführliche Details zur oberen Schnittstelle finden sich im Anhang in Kapitel A.
Die Schnittstelle ist unabhängig vom übertragenen Datentyp, sodaß beliebige neue Dienste in das System integriert werden können, ohne das Kommunikationssystem neu compilieren zu müssen.
Dies wird erreicht, indem der Datentyp durch einen logischen Namen gekennzeichnet wird. Dieser logische Name wird als Parameter an die Primitive der oberen Schnittstelle übergeben und dient in der Konvertierungsschicht des Kommunikationssystems als Schlüssel für Konvertierungsroutinen.[5] Die Nachricht selbst wird über eine *cast*-Operation als String übergeben bzw. empfangen.

#### RM_call_sync

In Abb. 4.5 ist die Schnittstelle für *RM_call_sync* aufgeführt.
Der Nachrichtenhenkel wird vom Kommunikationssystem beim Beginn eines Dienstaufrufes belegt und ist das eindeutige Identifikationsmerkmal des Aufrufs. Seine Funktion für die Protokollierung von Nachrichten und für interne Funktionen des Kommunikationssystems sind in Kap. 5 ausführlich beschrieben. Er muß bei einem *RM_call_return* wieder an das Kommunikationssystem übergeben werden (vgl. die Schnittstelle von *RM_call_return* in Abb. 4.7).
Der Parameter *caller* identifiziert den aufrufenden Dienst selbst. Er ist im ur-

[3] Vgl. dazu den Abschnitt 2.5 im Kap. 2.
[4] Vgl. dazu den Abschnitt 2.4 in Kap. 2 und den Abschnitt 4.2 weiter unten.
[5] Vgl. Abschnitt 2.4 mit einer Beschreibung der Darstellungsschicht.

| RM_call_sync | |
|---|---|
| &pto_msg_handle | Zeiger auf den Nachrichtenhenkel |
| &caller | Adreßkontrollblock des aufrufenden Dienstes |
| &pto_callee | optionaler Zeiger auf den Adreßkontrollblock des aufzurufenden Dienstes |
| gtrid | globaler Transaktionsbezeichner |
| service | Name des gewünschten Dienstes |
| rm_class | optionaler Name der Resource Manager Klasse des gewünschten Dienstes |
| local | optionale Angabe einer Lokalitätsbedingung für die Auswahl des Empfängers |
| &data_type | Datentyp der zu übertragenden Struktur |
| &message_length | Länge der zu übertragenden Struktur |
| &message_string | eigentliche Auftragsdaten |

Abbildung 4.5: Schnitttstelle von RM_call_sync

sprünglichen Entwurf für das Kommunikationssystem eingeführt worden und wird in zukünftigen Versionen eliminiert. Er ist redundant und wird durch eine Verwaltung der Aufrufhierarchie von Diensten mittels eines Stacks ersetzt werden.[6]
Der Zeiger auf den Adreßkontrollblock des Empfängers wird, wenn er nicht intialisiert ist, vom Kommunikationssystem belegt und an den aufrufenden Dienst zurückgegeben. Bei späteren Aufrufen kann der aufrufende Dienst diesen Zeiger verwenden, um eine bestimmte Empfängerkomponente vorzugeben. Die Funktionalität dieser Option ist eine zweifache:

1. Der aufrufende Dienst kann mithilfe eines initialisierten Empfänger-Adreßkontrollblocks eine Session zur Empfängerkomponente herstellen.

2. Es kann eine Optimierung auf der Ebene der Basiskommunikation erzwungen werden, wenn die Verbindung zu der Empfängerkomponente schon bei einem früheren Aufruf aufgebaut worden ist.

Der Parameter *service* ist obligatorisch und spezifiziert den gewünschten Dienst, die Angabe einer Resource Manager Klasse ist optional.
Der Parameter *local* kann eingesetzt werden, um eine Optimierung hinsichtlich der Auswahl des Empfängers, wie sie in Abschnitt 2.1 für die Beispiele *get_gtrid* und *flugbuchung* beschrieben worden ist, zu erzwingen. Dieser Parameter steuert die Auswahl entweder eines prozeßlokalen oder eines nicht-prozeßlokalen Empfängerdienstes.
Implementiert wird die Funktion *RM_call_sync* durch die im Abschnitt 4.2 beschriebenen Schritte.

[6]Dazu ist die Single-Threading-Implementierung eine Voraussetzung.



| RM_call_async | |
|---|---|
| &pto_msg_handle | Zeiger auf den Nachrichtenhenkel |
| &caller | Adreßkontrollblock des aufrufenden Dienstes |
| &pto_callee | optionaler Zeiger auf den Adreßkontrollblock des aufzurufenden Dienstes |
| gtrid | globaler Transaktionsbezeichner |
| service | Name des gewünschten Dienstes |
| callback_service | Name des Dienstes, der die Antwortnachricht erhalten soll |
| rm_class | optionaler Name der Resource Manager Klasse des gewünschten Dienstes |
| local | optionale Angabe einer Lokalitätsbedingung für die Auswahl des Empfängers |
| &data_type | Datentyp der zu übertragenden Struktur |
| &message_length | Länge der zu übertragenden Struktur |
| &message_string | eigentliche Auftragsdaten |

Abbildung 4.6: Schnitttstelle von RM_call_async

### RM_call_async

Die Schnittstelle, wie sie in Abb. 4.6 zu sehen ist, ist analog zu derjenigen des Primitivs *RM_call_sync*.
Der Sender muß nur zusätzlich den Empfänger der Antwortnachricht an der Schnittstelle für *RM_call_async* explizit spezifizieren, d. h. der Auftraggeber übergibt im Aufruf von *RM_call_async* den Namen des Dienstes innerhalb der Auftraggeberkomponente, der die Ergebnisse des Dienstaufrufes vom Auftragnehmer entgegennimmt, im Parameter *callback_service*.
Die Implementierung unterscheidet sich von einem *RM_call_sync* dadurch, daß keine Antwort der Empfängerkomponente abgewartet wird und die Parameter *data_type*, *message_length* sowie *message_string* bei der Rückkehr die ursprünglich übergebenen Werte enthalten.
Die Konsequenz für den aufrufenden Dienst ist, daß er nicht blockiert wird, weiterarbeiten kann und sich selbst mit *RM_call_return* beenden kann, ohne eine Antwort auf seinen Dienstaufruf erhalten zu haben.
Die Auftraggeberkomponente kann also weitere Nachrichten bearbeiten, die mittlerweile eingetroffen sind.[7]

### RM_call_return

*RM_call_return* ist ein Makro, das von einem APRICOTS-Dienst nach Beendigung des Auftrags aufgerufen wird und durch den Aufruf von *return* zum Ver-

[7]Vgl. die Beschreibung im Kap. 3, insbesondere die Abb. 3.4.

RM_call_return

| | |
|---|---|
| &pto_msg_handle | Zeiger auf den Nachrichtenhenkel |
| &data_type | Datentyp der zu übertragenden Struktur |
| &message_length | Länge der zu übertragenden Struktur |
| &message_string | eigentliche Antwortdaten |

Abbildung 4.7: Schnitttstelle von RM_call_return

lassen des Dienstes führt. Bei Antworten auf asynchrone Dienstaufrufe wird ein asynchroner Dienstaufruf an die callback-Adresse des Auftraggebers durchgeführt.
Die Schnittstelle in Abb. 4.7 weist gegenüber der Rückkehr von einem Remote Procedure Call, der für einen Dienst ein einfaches *return* vorsieht, eine höhere Komplexität auf.
Entsprechend dem Paradigma, daß das Kommunikationssystem von Änderungen der zu übertragenden Datenstruktur unabhängig sein soll, müssen zusätzlich zur eigentlichen Nachricht der Datentyp und die Länge der Nachricht angegeben werden, um die Konvertierung in das Übertragungsformat durchführen zu können.
Der Parameter *pto_msg_handle* verweist auf den Nachrichtenhenkel, mit dem der Dienst aufgerufen wurde und dient dem Kommunikationssystem für die Zuordnung von Auftrag und Antwort.

## 4.2 Schichten

Wie in Kap. 2 im Abschnitt 2.4 schon ausgeführt, umfaßt das Kommunikationssystem mehrere Funktionen und ist daher in vier Schichten implementiert, die an der angegebenen Stelle den Schichten OSI-7-Schichten-Referenzmodells zugeordnet werden.

- Berechtigungsprüfung
- Empfänger- und Adreßbestimmung
- Datenenkodierung bzw. -dekodierung
- Basiskommunikation

Die Algorithmen dieser Schichten werden in ihren wichtigsten Aspekten in den nächsten Unterabschnitten besprochen. Die Basiskommunikation wird wegen ihrer Gewichtigkeit und ihres Umfangs in einem separaten Abschnitt beschrieben.



### 4.2.1 Berechtigungsprüfung

Sie erfolgt, um die Integrität des Systems zu wahren. Ihr Ziel ist, zwei Möglichkeiten zum Mißbrauch des Kommunikationssystems in APRICOTS zu unterbinden:

1. Eingriffe von nicht-autorisierten Komponenten in den Ablauf der ConTract-Verarbeitung
2. Eingriffe von nicht-autorisierten Komponenten in die Konfiguration von APRICOTS

Der Ablauf der ConTract-Verarbeitung darf nur von der Komponente, die den ConTract angestoßen hat, durch eine Suspendierung bzw. einen Wiederanlauf oder ein globales Rücksetzen manipuliert werden.
Die Konfiguration von APRICOTS darf nur von einer Administrationsinstanz, z. B. einem ausgezeichneten ConTract Processing Monitor, verändert werden. Diese Instanz hat auch die Berechtigung, einen ConTract in seinem Ablauf zu beeinflussen.
Die Berechtigungsprüfung erfolgt bei

1. jedem Aufruf von *RM_call_sync* oder *RM_call_async* und
2. bei jedem Empfang einer Nachricht durch *main*.

Sie überprüft dabei den Komponentennamen der aufrufenden Instanz mit der Liste der berechtigten Komponenten, die beim Name Service abgefragt werden kann. Ist der Name der aufrufenden Instanz beim Name Service nicht bekannt oder besitzt sie nicht die benötigten Rechte für einen Eingriff in die ConTract-Verarbeitung bzw. die APRICOTS-Konfiguration, so wird der Aufruf zurückgewiesen.
Eine Implementierung dieser Funktionen konnte bis zum Abschluß dieser Arbeit nicht durchgeführt werden, da die Realisierung des globalen Name Service, der zur Identifizierung der berechtigten Komponenten benötigt wird, nicht geleistet werden konnte.[8]

### 4.2.2 Empfänger- und Adreßbestimmung

Eine weitergehendere Erklärung der Adressierung und der Adreßumsetzung findet sich in Abschnitt 4.4. Hier werden nur die Eigenschaften dieser Schicht im Rahmen der anderen Schichten des Kommunikationssystems beschrieben.
Die Bestimmung der Empfängerkomponente erfolgt mithilfe der Parameter *service*, *rm_class*, *local* und *pto_callee* aus der Schnittstelle des Resource Manager Calls.

[8]Die Implementierung des Name Service wurde aus der Diplomarbeit ausgegliedert. Die Implementierung wurde durch die Nichtverfügbarkeit der Ingres-Maschine Ende Februar 1992 verhindert.

Verweist der Zeiger *pto_callee* auf einen Adreßkontrollblock, so wird die dort spezifizierte Komponente als Empfängeradresse übernommen. Eine Bestimmung der Adresse ist nicht mehr notwendig, da die Empfängerkomponente schon bereits bei einem früheren Dienstaufruf, der diesen Adreßkontrollblock der aufrufenden Komponente zurückgegeben hat, angesprochen worden ist. Auch die Daten aus dem Verbindungsaufbau sind bereits eingetragen, sodaß sich folgende Vorteile aus diesem Vorgehen ergeben:

**Optimierung:** Da die Adresse des Empfängers schon bekannt und damit keine Anfrage beim Name Service notwendig und die Verbindung zum Empfänger schon aufgebaut ist,[9] ist dieses Verfahren i. a. wesentlich schneller als die Suche nach einer geeigneten Empfängerkomponente. Dies gilt natürlich nur, wenn der aufrufende Dienst einen Empfänger spezifiziert, der sich bisher als zuverlässig und schnell erwiesen hat.

**Kontexterhaltung:** Statt bei jedem Aufruf von *step_computation_request* den benötigten Kontext mitzusenden bzw. diesen Kontext durch den aufgerufenen Step recherchieren zu lassen, kann der aufrufende Dienst diesen Kontext bei einem Dienst, der zuvor bereits aktiviert wurde, als bekannt voraussetzen.[10]

Ist der Zeiger *pto_callee* nicht initialisiert, wird Speicherplatz für ihn allokiert und die Struktur mit den Werten für *service* und *rm_class* belegt. Entsprechend dem Wert des Parameters *local* wird entweder eine prozeßlokale Tabelle mit den Einsprungadressen aller Dienste innerhalb der Komponente oder ein prozeßlokaler Adreßcache mit allen bislang recherchierten Diensten anderer Komponenten durchsucht.[11] Ist die Suche erfolglos, so wird sie für die Spezifikation „prozeßlokaler Empfänger erwünscht" abgebrochen. Sonst erfolgt eine Anfrage beim Name Service mit den Angaben für *service* und *rm_class*.
Die Anfrage beim Name Service liefert eine Liste von Adreßkontrollblöcken zurück. Die Liste umfaßt alle „Treffer", die mit dem Dienstnamen übereinstimmen. Falls die Resource Manager Klasse bei der Anfrage spezifiziert war, werden zusätzlich alle Dienste der „Treffer"-Komponenten zurückgeliefert, die mit der Resource Manager Klasse und der Komponente der Treffer übereinstimmen.
Dies bewirkt eine Beschleunigung des Suchverfahrens bei künftigen Adreßbestimmungen, da anzunehmen ist, daß bei späteren Dienstaufrufen, die an andere Steps derselben Resource Manager Klasse gehen, der Adreßkontrollblock des zuvor aufgerufenen Dienstes verwendet wird.

### 4.2.3 Datenkonvertierung

Eine Konvertierung der Daten für die Übertragung ist notwendig, um die Nachrichten mit den Kontrollinformationen wie den Adreßkontrollblöcken und dem

[9] Diese Kenntnis rührt aus einem früheren Dienstaufruf her, aus dem der aufrufenden Komponente dieser Adreßkontrollblock mitgeteilt worden ist.
[10] Die Verwaltung von Kontext bleibt dabei aber vollkommen eine Angelegenheit der kommunizierenden Anwendungen.
[11] Zu den Datenstrukturen für die Adreßverwaltung siehe Kap. 5.



Nachrichtenhenkel zu verknüpfen und um die Übertragung in einer heterogenen Rechnerumgebung mit unterschiedlicher Byte-Ordnung zu ermöglichen.[12]
Die eigentliche Nachricht kann auf mehrere Weisen im Dienstaufruf übergeben werden. Entweder wird sie mithilfe von XDR-Routinen schon seitens des Senders in einen Bytestream umgeformt, der vom Empfänger selbst in das eigentliche Nachrichtenformat zurückgeformt wird, sodaß das Basiskommunikationssystem stets nur einen Bytestream transportiert, oder der Sender gibt zusätzlich zur zu übertragenden Nachricht eine Typkennzeichnung an und das Kommunikationssystem muß falls notwendig eine Kodierung und - im Empfänger - eine Dekodierung der Daten durchführen.
Die erste Alternative führt zu einer Belastung der Systemkomponenten und im prozeß- und rechnerlokalen Fall zu unnötigen Datenkonversionen und wird deshalb nicht in Erwägung gezogen.
Die zweite Alternative macht eine zusätzliche Datentyptabelle sowie eine dynamisch änderbare Bibliothek mit Konvertierungsfunktionen notwendig, auf die das Kommunikationssystem zugreifen kann.[13]
Die Kodierung geschieht in zwei Schritten. Zuerst erfolgt der Aufbau des Nachrichtenkontrollblocks. Dazu werden alle Angaben, die auch in der Schnittstelle eines Dienstaufrufes erscheinen, in eine Datenstruktur gebracht, die sogenannte **APRICOTS-Übertragungseinheit**.
Danach erfolgt die Konvertierung der gesamten APRICOTS-Übertragungseinheit in das XDR-Format ([2]). Der Schlüssel zur nötigen Konversionsroutine ist der Parameter *data_type* aus der Schnittstelle des Dienstaufrufs bzw. des *RM_call_return*.
Dieser zweite Schritt ist optional für die Kommunikation zwischen heterogenen Rechnern.
Bei der Dekodierung auf der Empfängerseite werden diese Schritte in umgekehrter Reihenfolge durchgeführt.

## 4.3 Basiskommunikation

Die Schicht für die Basiskommunikation wird hier in einem eigenen Abschnitt behandelt, weil auf dieser Ebene die umfangreichsten Vorüberlegungen eingehen und die meisten Entwurfsentscheidungen anstehen. Die Mechanismen auf der Ebene der Basiskommunikation müssen folgenden Anforderungen genügen:

- Verklemmungsfreiheit, um ein Rücksetzen auf der Ebene der Nachrichtenübertragung zu vermeiden.[14]

[12] Dies gilt vor dem Hintergrund einer geplanten Erweiterung von APRICOTS auf ein heterogenes System.
[13] Vgl. dazu den Abschnitt 4.2.3.
[14] Deadlocks bei der Kommunikation in einer verteilten Umgebung zu erkennen und zu beheben ist wesentlich komplexer und aufwendiger (Verwaltung aller synchronen Wartesituationen im Netz, mit allen beteiligten TM's koordinierte Rücksetzung bestimmter Komponenten durch die ConTract Processing Monitor) als diese zu vermeiden.

- Minimierung von benötigten Resourcen wie zeitraubenden Plattenzugriffen und Prozeßwechseln, um die Nachrichtenübertragung zu beschleunigen.
- Verfügbarkeit der implementierten Mechanismen auf allen vorgesehenen Systemen (SunOS, VMS, Guardian), um die Portabilität zu sichern und damit den Entwicklungs- und Änderungsaufwand für das Kommunikationssystem in Grenzen zu halten.
- ständige Empfangsbereitschaft auf der Empfängerseite, um bei der Übertragung Wartezeiten des Senders zu vermeiden.

Im folgenden werden die Übertragungsmechanismen für die drei Fälle **prozeßlokaler**, **rechnerlokaler** und **entfernter** Nachrichten behandelt, jeweils nach synchronem und asynchronem Übertragungsmodus unterschieden.
Daß prozeßlokale Nachrichten getrennt von den beiden anderen Fällen realisiert werden können, ist unmittelbar einsichtig, wenn bedacht wird, daß prozeßlokale Übertragung per definitionem keiner Interprozeßkommunikation (IPC) bedarf.
Die Unterscheidung zwischen rechnerlokaler und rechnerentfernten Nachrichten rührt aus der Anforderung her, den Bedarf an Resourcen zu minimieren. So stehen auf rechnerlokaler Ebene weitaus billigere IPC-Mechanismen zur Verfügung als für die IPC zwischen zwei Rechnern. Insbesondere verfügt jedes Betriebssystem über prozeßlokale IPC-Mechanismen, die nicht die gesamte Hierarchie des ISO-7-Schichten-Protokolls durchlaufen müssen.

### 4.3.1 Prozeßlokale Kommunikation

Im synchronen Fall (*RM_call_sync*) wird in der Schicht der Basiskommunikation die Adresse des Empfängerdienstes bestimmt und direkt anschließend der Dienst aktiviert, d.h. die den Dienst implementierende Funktion aufgerufen. Die Rückgabewerte des Empfängerdienstes[15] werden an den aufrufenden Dienst zurückgereicht.
Für den prozeßlokalen Fall ist der Aufruf von *RM_call_sync* optimiert worden. Es findet keine Konvertierung ins APRICOTS-Übertragungsformat, folglich auch keine Dekodierung, statt, da dies durch die Realisierung des Resource Manager Calls als Funktionsaufruf nicht notwendig ist.
Im asynchronen Fall wird die Nachricht in das APRICOTS-Übertragungsformat gebracht und ein Zeiger auf diese Struktur in eine prozeßlokale Nachrichtenwarteschlange eingehängt. Die Funktion *RM_call_async* führt nur die Senderseite des Resource Manager Calls aus.
Nach Beendigung des aufrufenden Dienstes geht die Kontrolle wieder an die Routine *main*. Das Empfangen der Nachricht erfolgt dann in der Nachrichtenverteilerschleife in *main*.

[15] Die Parameter *data_type*, *message_length* und *message_string* der Schnittstelle von *RM_call_sync* sind hier gemeint.



### 4.3.2 Rechnerlokale Kommunikation

Für die Übertragung zwischen zwei Prozessen auf demselben Rechner bieten sich verschiedene IPC-Mechanismen an.
Der **Remote Procedure Call** ([5]) in der Implementierung auf SunOS 4 ist im rechnerlokalen Fall sehr ineffizient, da wie im entfernten Fall **sämtliche** Schichten des ISO-Protokolls durchlaufen werden, was letztendlich bedeutet, daß rechnerlokale und rechnerentfernte Aufrufe genauso schnell respektive langsam sind.[16] Darüberhinaus muß die Senderkomponente warten, bis der Empfänger seinen Eingabekanal abfragt.[17] Eine Umgehung dieses Wartens ließe sich erreichen, wenn der Sender seine Nachricht asynchron sendet. Er muß dann jedoch auf jegliche Quittierung seiner Nachricht durch den Übertragungsmechanismus verzichten. Der Transaktionsschutz durch Logging und das Rücksetzen von Nachrichten bei Verklemmungen werden dadurch erheblich erschwert.
Ein letztes Argument gegen den Remote Procedure Call ist das Paradigma für diesen Mechanismus, daß zwei Komponenten nicht gleichzeitig Client und Server füreinander sein können. Die Implementierung dieser Option ist vergleichsweise aufwendig.

Die Unix-Variante System V bietet zwei Mechanismen für die rechnerlokale IPC: **Pipes** und **Shared Memory Areas**.
Pipes sind in Unix zwar sehr effizient implementiert, aber in ihrer Anwendung restringiert. So können nur Pipes zwischen Prozessen aufgebaut werden, von denen der eine ein Vorfahre des anderen ist.
Ein weiterer Nachteil ist, daß ein Sender, um eine Quittierung zu erhalten, warten muß, bis der Empfänger die übertragene Nachricht aus der Pipe holt.

Eine Mailbox, realisiert als Shared Memory Bereich, bietet demgegenüber mehrere Vorteile.[18]

- Die Indirektion der Übertragung verhindert eine gegenseitige Blockierung. Zwei Komponenten können sich gleichzeitig gegenseitig Nachrichten in ihre Mailboxen eintragen.
- Es muß kein Prozeßwechsel im rechnerlokalen Fall vorgenommen werden, um eine Quittierung der Nachricht zu erhalten.
- Eine Mailbox ist in ständiger Empfangsbereitschaft, die nur durch einen Überlauf beeinträchtigt wird.[19]
- Die Dauerhaftigkeit ist letztendlich ein sehr entscheidendes Kriterium. Eine Mailbox überlebt den Absturz aller angeschlossenen Komponenten

[16] Vgl. dazu die Untersuchungen von [13]
[17] Es sei denn, man implementiert ein Multi-Threading mit den in Kap. 1 aufgeführten Konsequenzen.
[18] Die Realisierung ist an den Mechanismus in [7] angelehnt. Die Implementierung wurde aus der Diplomarbeit ausgegliedert.
[19] Dies gilt aber auch für alle anderen Übertragungsmechanismen in gleicher Weise, da jeder Ein- und Ausgabekanal nur über begrenzte Verwaltungsresourcen verfügt.

und ist somit stabiler als jeder der bisher vorgestellten Übertragungsmechanismen.[20]

Aus diesen Gründen wird die Kommunikation auf einem Rechner über den Mailbox-Mechanismus abgewickelt.

Wird eine Komponente neu gestartet, so wird vom ConTract Processing Monitor eine Mailbox generiert, die der neuen Komponente als Kanal für eintreffende Nachrichten dient. Der ConTract Processing Monitor hängt diese Mailbox in seinen eigenen Adreßbereich ein und kann somit der neugeschaffenen Komponente Nachrichten senden. Diese wiederum erhält bei ihrer Aktivierung eine Id für diese Mailbox als Parameter mit und hängt ihre private Mailbox in ihren Adreßbereich ein. Von diesem Zeitpunkt an ist die Komponente in der Lage, eintreffende Nachrichten zu bearbeiten.

### 4.3.3 Kommunikation zwischen mehreren Rechnern

Für die Kommunikation zwischen Rechnern steht auf SunOS nur ein Mechanismus zur Verfügung: der socketbasierte Remote Procedure Call (RPC).
Wegen der schon bei der Beschreibung der rechnerlokalen IPC angeführten Probleme mit wechselseitigen Client-Server-Beziehungen und der Unmöglichkeit asynchrone Nachrichten mit Empfangsquittierung zu implementieren,[21] wurde eine Variante entwickelt, deren hervorstechendes Merkmal die Indirektion ist.
So sendet eine Komponente ihre Nachricht nicht direkt an die Empfängerkomponente, sondern an einen sogenannten **Briefträger** auf dem Zielrechner (siehe Abb. 4.8).
Dieser Briefträger ist als RPC-Server implementiert, der als einzige Dienste die Funktionen *sende_nachricht_weiter* und *storniere_nachricht* anbietet.
Beim Aufruf von *sende_nachricht_weiter* trägt er die übergebene Nachricht beim spezifizierten Empfänger in die Mailbox ein.
*storniere_nachricht* holt eine mit dem Nachrichtenhenkel spezifizierte Nachricht aus der Mailbox der Komponente, die diese Nachricht bereits erhalten hat, sofern diese Komponente nicht bereits die Verarbeitung dieser Nachricht begonnen hat. Jeder ConTract Processing Monitor, d.h. jeder Rechner im APRICOTS-System, verfügt über einen Briefträger.
Der Briefträger gibt bei Entgegennahme eines Auftrags dem Sender einer Nachricht das gewünschte Acknowledge und garantiert damit die Zustellung an den Empfänger.
Die Kommunikation von der Senderkomponente zum Briefträger erfolgt über einen Remote Procedure Call auf Socket-Basis, die Weiterleitung an die Empfängerkomponente erfolgt als Eintragung in die Mailbox der Empfängerkomponente.
Deadlocks können nicht entstehen, da Briefträger selbst keine Remote Procedure Calls an andere Prozesse senden und daher reine Server-Prozesse sind.

[20] Mit zusätzlichen Maßnahmen ([3]) können Nachrichtenwarteschlangen sogar gegen Knotenausfälle gesichert werden.
[21] Es sei denn, der Sourcecode für Remote Procedure Calls würde geändert.



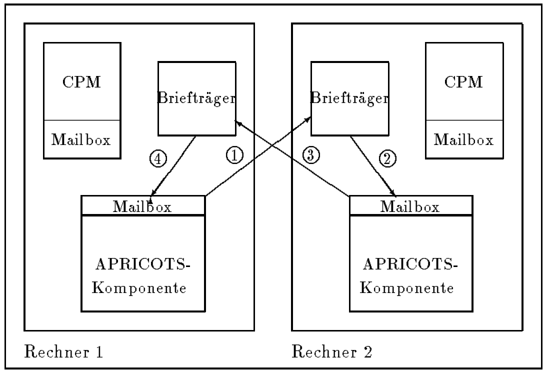


Abbildung 4.8: Briefträgermechanismus in APRICOTS

Der Ablauf eines rechnerentfernten Dienstaufrufes erfolgt dann, wie es in Abb. 4.8 zu sehen ist:

> Zunächst sendet die APRICOTS-Komponente auf Rechner 1 ihren Auftrag an den Briefträger auf Rechner 2 (1). Dieser trägt die Nachricht in die Mailbox der Auftragnehmerkomponente ein (2). Die Antwort wird von der Auftragnehmerkomponente an den Briefträger auf Rechner 1 zurückgeschickt (3), der die Antwort in die Mailbox der Auftraggeberkomponente einträgt (4).

## 4.4 Adreßumsetzung und Adreßverwaltung

Dieser Abschnitt führt in die Verfahren zur Adreßumsetzung und zur Adreßverwaltung ein. Die interne Struktur der verwalteten Daten wird in Kap. 5 erläutert.

### 4.4.1 Adreßumsetzung

Die Adresse eines Dienstes setzt sich zusammen aus:

**auf der Anwendungsebene:** Dienstnamen und Resource Manager Klasse

**auf der Ebene der Empfängerbestimmung:** Komponentennamen

**auf der Ebene der Basiskommunikation:** logischer und physischer Adresse

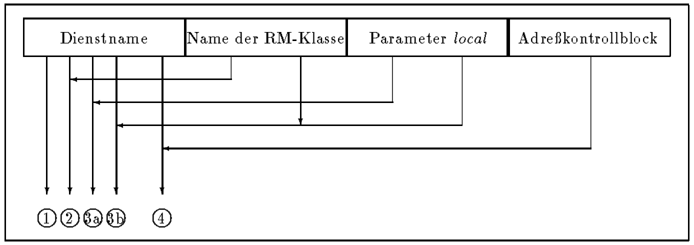


Abbildung 4.9: Adreßbestimmung in APRICOTS

Zur Adressierung eines Dienstes kann eine Anwendung neben dem Dienstnamen und der Resource Manager Klasse noch weitere Angaben spezifizieren. Die Abb. 4.9 verdeutlicht die verschiedenen Varianten der Adressierung.
Für die Adressierung eines Dienstes reicht es aus, den logischen Namen des Empfängerdienstes anzugeben (1). Wenn z. B. alle Drucker in APRICOTS Laserdrucker der gleichen Funktionalität sind, so kann ein Druckauftrag an einen beliebigen Druckerserver resp. dessen Dienst *formulare_drucken* ergehen. Das Kommunikationssystem ermittelt dann allein aufgrund des Dienstnamens die Empfängerkomponente.
Zusätzlich kann die Resource Manager Klasse angegeben werden (2). Ihre Spezifikation ist lediglich optional. Sie kann jedoch von entscheidender Bedeutung sein, wenn verschiedene Resource Manager dieselben Dienste anbieten:

> Ist der Dienst *flugbuchung* sowohl für die Resource Manager Klasse *lusthansa* als auch für die Resource Manager Klasse *tibetan_oneways* verfügbar, so kann der Erfolg einer Transaktion von der Wahl der Klasse abhängen.[22]

Wird auch der Parameter *local* spezifiziert (3a und 3b), so wird diese Angabe benutzt, um entweder einen prozeßlokalen Empfänger oder einen nicht-prozeßlokalen Empfänger zu bestimmen. Dies ist bei Aufträgen wichtig, für die die Parallelität der Ausführung bedeutsam ist, wie es z. B. bei Step-Aufrufen der Fall ist.[23] Eine prozeßlokale Step-Ausführung führt zu einer Sequentialisierung des Ablaufs, die unerwünscht ist. Deshalb sollen solche Step-Aufrufe nach Möglichkeit nicht prozeßlokal erfolgen. Andererseits ist es erwünscht, daß „kurze" Dienstaufrufe prozeßlokal bearbeitet werden, um den Kommunikationsaufwand zu minimieren.[24]
Eine Verlagerung der Entscheidung, ob ein Dienstaufruf prozeßlokal oder nicht

[22] *tibetan_oneways* bietet nur Oneway-Tickets in den Himalaya an, vgl. den Beispiel-ConTract aus Kap. 1.
[23] Vgl. die Aussagen zum Step-Aufruf *flugbuchung* in Abschnitt 2.1.
[24] Dies ist sinnvoll bei den meisten Aufrufen zwischen ConTract Manager und Transaction Manager.



bearbeitet werden muß, in die Empfängerbestimmungsschicht des Resource Manager Call bedeutet, daß das Kommunikationssystem mit Informationen über die Semantik von Diensten belastet wird. Dies beeinträchtigt die Unabhängigkeit des Kommunikationssystems von der Konfiguration von APRICOTS.
Alternativ zu Resource Manager Klasse und dem Parameter *local* kann ein Adreßkontrollblock aus einem früheren Aufruf angegeben werden (4). Die in diesem Adreßkontrollblock spezifizierte Komponente wird vom Kommunikationssystem als Empfängerkomponente übernommen.
Dies ist erwünscht im Zusammenhang mit dem Aufbau einer Session zwischen zwei Komponenten,[25] z. B. einem ConTract Manager und einem Step Computation Server, die die Kontextverwaltung zwischen mehreren Steps effizienter macht, da der Step Computation Server für den Step-Kontext nicht jedesmal erneut auf eine externe Datenbasis zurückgreifen muß.

Die eigentliche Adreßbestimmung geschieht in vier Schritten.
Zunächst wird der logische Komponentenname des Empfängers bestimmt. Da der Komponentenname eindeutig ist und auch einen Absturz überdauert, ist garantiert, daß eine Antwort für einen Dienstaufruf auch nach Absturz bzw. Migration der aufrufenden Komponente zugestellt werden kann (Empfängerbestimmungsschicht).
Danach erfolgt die Bestimmung der prozeßunabhängigen Adreßinformation, d.h. die Bestimmung des Ortes, an dem sich die Empfängerkomponente befindet und die nötigen Angaben zur Adressierung der Verbindung über das vorhandene Basiskommunikationsprotokoll (Mailbox-Id bzw. Portnummer).
Im vierten Schritt werden die prozeßspezifischen Informationen bestimmt wie *mailbox_handle* oder *socket_handle*, die beim Aufbau eines Kommunikationskanals erzeugt werden.
Die beiden letzten Schritte erfolgen in der Basiskommunikationsschicht.

### 4.4.2 Globale Adreßhaltung

Der Name Service ist kein Schwerpunkt der vorliegenden Arbeit. Die Kriterien für den Entwurf waren deshalb eine einfache Realisierbarkeit und eine an die Adreßbestimmung und -umsetzung angelehnte Funktionalität, aber nicht die Effizienz des implementierten Mechanismus.[26]
Der **Name Service** ist eine globale Tabelle, die alle nicht-prozeßspezifischen Adreßinformationen zu allen Komponenten in APRICOTS enthält. Das sind Angaben wie Komponentennamen, Mailbox-Ids oder Socket-Namen.
Diese Tabelle wird benötigt zur Adreßrecherche bei der Bestimmung einer Empfängerkomponente oder zur Aktualisierung veralteter Adressen im prozeßlokalen Cache. Daneben ist sie auch notwendig für die Überprüfung, ob ein Komponentenname gültig ist, z. B. im Zusammenhang mit der Berechtigungsprüfung.
Für die Implementierung stehen drei Alternativen zur Wahl,

[25] Vgl. hierzu den Abschnitt 4.2.
[26] Näheres zum Thema „Name Service" findet sich in [9], [3], [18] u. a.

1. als private Datenstruktur des ConTract Processing Monitor,
2. als Shared Memory Bereich unter Kontrolle des ConTract Processing Monitor oder
3. als Datenbankrelation.

Die ersten beiden Alternativen lassen sich in verschiedenen Varianten implementieren, die nach dem Grad der Replikation unterschieden sind. Wird nicht der gesamte Name Service für jede realisierte Tabelle repliziert, so ist zwar der Aufwand für ein Update vermindert, da nicht alle Tabellen bei einer Änderung aktualisiert werden müssen, andererseits muß die Recherche einer Adresse eventuell über mehrere ConTract Processing Monitore laufen, um einen ConTract Processing Monitor zu erreichen, der die benötigte Adresse verwaltet. Das führt zu einer erhöhten Anzahl von Prozeßwechseln und zu langen Wartezeiten.
Die Verwaltung einer vollständig replizierten Tabelle ist zwar in Hinsicht auf den Aufwand für ein Update ungünstiger, dafür ist die Recherche jedoch wesentlich einfacher und schneller, insbesondere da eine Anfrage beim Name Service **alle** „Treffer" zurückliefern soll, um eine Lastbalancierung nicht der zufälligen Verteilung der Tabellen auf die ConTract Processing Monitore anheim zu stellen. Wenn eine Replikation erfolgt, so ist es besser, sie ist vollständig.
Die Implementierung als Shared Memory Area hat dann gegenüber einer privaten Datenhaltung den Vorteil, daß der Zugriff auf die Name Service Tabelle ohne Prozeßwechsel stattfinden kann. Dies wiegt den Nachteil auf, daß der Zugriff auf einen gemeinsamen Speicherbereich synchronisiert werden muß. Zudem kann dieser Bereich als ein häufig benutzter bei vernünftiger Hauptspeicherverwaltung im Hauptspeicher gehalten werden.

Der Name Service als Datenbankrelation hat demgegenüber den schweren Nachteil, daß Datenbankzugriffe meistens mit Plattenzugriffen verbunden sind. Andererseits entstehen aus der Haltung in einer Datenbankrelation auch große Vorteile. So ist kein „Broadcasting" für Änderungen in Replikationen notwendig, Logging und Recovery werden vom Datenbankverwaltungssystem durchgeführt, sodaß die Konsistenz der Name Service Einträge zu jedem Zeitpunkt erhalten bleibt.
Darüberhinaus steht der Zugriff über die relationale Anfragesprache SQL im Einklang mit der Adreßauflösung:

$$\text{Dienstname} \mapsto [\ \textbf{Menge} \text{ von Empfänger-Resource-Managern}\ ]$$

Deshalb wurde für die Realisierung diese Alternative gewählt und durch Maßnahmen für prozeßlokales Name Caching optimiert.

### 4.4.3 Prozeßlokales Name Caching

Innerhalb einer APRICOTS-Komponente wird ein Name Caching unter der Kontrolle des Kommunikationssystems durchgeführt. Dies erfolgt aus der Überlegung heraus, daß für eine Anfrage beim Name Service sehr aufwendige Datenbankanfragen notwendig sind, die zeitraubende Prozeßwechsel und Plattenzugriffe erfordern.



Realisiert wird dies durch eine Liste von Resource Managern und ihren Diensten. Eingetragen werden alle „Treffer", die die Komponente bei ihren Anfragen vom Name Service zurückerhalten hat. Wird eine Verbindung zu einer Komponente aufgebaut, so wird dies im lokalen Cache eingetragen.
Ein Optimierungseffekt dabei ist, daß bereits installierte Verbindungen zu anderen Komponenten bei vorherigen Dienstaufrufen erhalten bleiben und der Aufwand für Verbindungsaufbau bzw. -abbau bei erneuten Dienstaufrufen an diese Komponenten wegfällt.[27]

[27] Die interne Struktur des lokalen Cache wird in Kap. 5 beschrieben.

# Kapitel 5

# Datenstrukturen

Die Beschreibung des Aufbaus einer Adresse ist bewußt aufgeschoben worden, da die Strukturen einer ausführlichen Erläuterung bedürfen.
Adressen werden in zwei Zusammenhängen verwendet: als Funktionsparameter und als Einträge in Verwaltungsstrukturen wie dem Name Service oder dem prozeßlokalen Adreßcache. Als Funktionsparameter werden sie auf verschiedenen Ebenen des Kommunikationssystems verschieden verwendet: je anwendungsnäher, desto mehr wird von den zugrundeliegenden physischen Mechanismen und Gegebenheiten abstrahiert.
Dieses Kapitel beschreibt den Aufbau eines Adreßkontrollblocks unter Berücksichtigung dieser Abstraktionsstufen und erläutert weitere wichtige Datenstrukturen, die vom Kommunikationssystem verwendet werden.
Daran schließt sich die Beschreibung der Adreßtabellen, die zur Realisierung des Name Service, des prozeßlokalen Adreßcache und der Einsprungadressentabelle notwendig sind. Abschließend werden die Strukturen der Nachrichtenwarteschlangen betrachtet.

## 5.1 Adressen und Adreßkontrollblöcke

Auf der Anwendungsebene[1] setzt sich die Adresse aus dem Dienstnamen und dem optionalen Namen der Resource Manager Klasse zusammen.
Aus diesen Angaben ermittelt das Kommunikationssystem unter Berücksichtigung zusätzlicher Angaben[2] eine Liste möglicher Empfängerkomponenten, aus denen eine ausgewählt wird (Adreßbestimmungsschicht).
In der Basiskommunikationsschicht wird der Komponentenname in die physische Adresse der Komponente umgesetzt. Diese setzt sich aus dem logischen Namen zur Identifizierung des Eingabekanals und den Angaben zur physischen Adressierung des Eingabekanals zusammen. So ist ein Socket gekennzeichnet durch einen Namen, über den ihn jeder Prozeß ansprechen kann, um einen

[1] Damit ist die Schnittstelle der Dienstaufrufe *RM_call_sync* und *RM_call_async* gemeint.
[2] Zu den Lokalitätsbedingungen und der Angabe einer verbindlichen Empfängeradresse vgl. im Kap. 4 den Abschnitt 4.4.





| | |
|---|---|
| ConTract Processing Monitor | *cpm_<hostname>* |
| Briefträger | *mail_<hostname>* |
| sonstige Komponente | *<program_name>_<hostname>_<conter>* |

Tabelle 5.1: Verfahren zur Bildung des Komponentennamens

Verbindungsaufbau einzuleiten, und durch den Henkel, mit dem eine aufgebaute Verbindung adressiert wird.
Auf der Empfängerseite werden die Adreßattribute „Dienstname" und „Name der Resource Manager Klasse" in die Einsprungadresse der Funktion umgesetzt.
Bei der Betrachtung der Schichten des Kommunikationssystems sind also drei Gruppen von Adreßangaben zu erkennen:

1. der Dienstname und der Name der Resource Manager Klasse,
2. der eindeutige Komponentenname sowie
3. die Adresse auf der Ebene der Basiskommunikationsmechanismen.

Der logische Dienstname ist ein String, der den Namen der implementierten Funktion in Großbuchstaben enthält.
Der Name einer Resource Manager Klasse wird ebenfalls mit Großbuchstaben gebildet.
Die Eindeutigkeit des Komponentennamens wird durch einen einfachen Algorithmus erreicht. Der Name einer neuerzeugten Komponente wird vom ConTract Processing Monitor auf dem Rechner der zu startenden Komponente vergeben. Der Name wird nach dem in Tabelle 5.1 gezeigten Verfahren gebildet. Der Zähler wird vom ConTract Processing Monitor verwaltet und garantiert die Eindeutigkeit. Die Dauerhaftigkeit des Komponentennamens verbunden mit der Eindeutigkeit sichert die Zustellung einer Antwort an die Auftraggeberkomponente eines Dienstaufrufs auch bei Migration des Auftraggebers. Die Adresse für die Basiskommunikation zerfällt in eine logische und eine physische Adresse. Die logische Adresse ist eine global gültige Identifikation und entspricht dem Dateinamen. Die physische Adresse ist prozeßspezifisch und findet ihr Pendant im Dateideskriptor. Die physische Adresse wird wie der Dateideskriptor erst beim Verbindungsaufbau, dem „Öffnen" des Kanals, generiert.
Die Adresse für die Basiskommunikation wird nach Entfernung zwischen adressierendem und adressiertem Dienst unterschieden, wie in Tabelle 5.2 zu sehen ist.
Zur Adreßhaltung und zur Nachrichtenübertragung sind die Kennzeichnung des Komponentenzustandes sowie des Zustands der Verbindung der adreßhaltenden (adressierenden) zu der adressierten Komponente als weitere wichtige Bestandteile eines Adreßblocks notwendig.
Beim Entwurf der Datenstrukturen war zu berücksichtigen, daß nicht alle Informationen immer gleichzeitig benötigt werden:

| Entfernung | logische Adresse | physische Adresse |
|---|---|---|
| prozeßlokal | Funktionsname (Dienstname) | Einsprungadresse innerhalb des Speicherbereiches des Prozesses |
| rechnerlokal | Mailbox-Id | Startadresse der Mailbox im Speicherbereich des adressierenden Prozesses |
| entfernt | Socketname | Sockethenkel |

Tabelle 5.2: Adressen der Basiskommunikationsschicht

```
address_control_block
    service_name
    resource_manager_class_name
    component_name
    hostname
    state_of_component
    address_type
    logical_address
    state_of_connection
    physical_address
```

Abbildung 5.1: Aufbau eines Adreßkontrollblocks

- Außer in den Name Service Tabellen wird in allen Verwendungen nur jeweils eine der drei Adreßtypen Funktionsadresse, Mailboxadresse oder Socketadresse benötigt.
- Im prozeßlokalen Fall ist die Frage nach dem Komponenten- und dem Verbindungszustand obsolet.

Statt für jeden möglichen Gebrauch eine eigene Datenstruktur zu definieren, ist ein einheitlicher Datentyp geschaffen worden und nur für die interne Struktur der Name Service Tabelle und der Einsprungadressentabelle ein abweichender Typ definiert worden.
Diese einheitliche Datenstruktur wird **Adreßkontrollblock** genannt, ihr Aufbau ist in Abb. 5.1 dargestellt. Die Felder für die logische und die physische Adresse sind jeweils als Union realisiert, da nur zu einem Zeitpunkt eine der drei Varianten Funktionsadresse Mailboxadresse oder Socketadresse benötigt wird.
Für die Einsprungadressentabelle wird eine reduzierte Struktur verwendet, die nur den Komponentennamen, den Dienstnamen, den Namen der Resource Manager Klasse und die Einsprungadresse des Dienstes umfaßt.
Der Name Service enthält dagegen keine prozeßspezifischen Informationen wie die physischen Adressen von Kommunikationskanälen, muß aber gleichzeitig



Mailboxadresse und Socketadresse speichern.

## 5.2 Weitere wichtige Datentypen

Alle im Kommunikationssystem verwendeten Datentypen und Konstanten sind in einem eigenen Papier enthalten und sind in Kürze „Online" verfügbar.
Ich habe mich hier auf zwei wichtige Strukturen beschränkt, die für das Übertragungsprotokoll von besonderem Interesse sind.

### 5.2.1 Nachrichtenhenkel

Der **Nachrichtenhenkel**, auch als *message_handle* bezeichnet, ist der eindeutige Identifikator eines Dienstaufrufs. Seine Funktionalität ist dreifach:

1. Seine Eindeutigkeit dient Zwecken der Fehlerbehandlung.[3]
2. Der Transaktionsschutz für Dienstaufrufe wird über das Logging von Nachrichtenhenkel und Nachricht durchgeführt.
3. Intern verwendet das Kommunikationssystem den Nachrichtenhenkel zur Adressierung und Bestimmung des Verarbeitungszustandes eines Dienstaufrufes.

Zu Protokollzwecken muß die Eindeutigkeit des Nachrichtenhenkels gewährleistet sein. Deshalb umfaßt die Datenstruktur neben den Adreßangaben der aufrufenden Komponente auch einen Zähler. Dieser Zähler wird von der aufrufenden Komponente bei jedem Dienstaufruf imkrementiert.
Die Adreßangaben umfassen jeweils für den aufrufenden Dienst, den aufgerufenen Dienst und - im asynchronen Fall - den Dienst, der die Antwort erhält, den Dienstnamen, den Namen der Resource Manager Klasse sowie den Komponentennamen. Diese Angaben sind ausreichend, um eine erneute Adressierung bei einem Fehlerfall durchführen zu können.
Ein weiteres Feld bezeichnet die Verarbeitungsphase, die der Dienstaufruf erreicht hat, jeweils differenziert nach Synchronitätsmodus. Dieses Feld dient dem Kommunikationssystem als Entscheidungsmerkmal, ob noch eine Antwortnachricht zu erbringen ist, sowie als Identifikationsmerkmal für das Logging.[4]

### 5.2.2 APRICOTS-Übertragungseinheit

Die Datenstruktur der **APRICOTS-Übertragungseinheit** ist ein Kontrollblock, der das Protokoll zur Nachrichtenübertragung definiert. Er enthält

- den Nachrichtenhenkel,

[3] Mithilfe des Nachrichtenhenkels werden Verdopplung bzw. Verlust der Nachricht vermieden, vgl. Kap. 6.
[4] Das Logging ist in Kap. 6 beschrieben.

- Adreßkontrollblöcke für Sender und Empfänger einer Nachricht sowie
- die Nachricht selbst mit den Parametern, die den globalen Transaktionsbezeichner, den Datentyp und die Länge der Nachricht spezifizieren.

Jeder Dienstaufruf[5] durchläuft auf der Senderseite und auf der Empfängerseite die Dekodierung in dieses Format und die Kodierung aus diesem Format. In der Basiskommunikationsschicht wird diese Übertragungseinheit vom Sender zum Empfänger übertragen.

## 5.3 Strukturen zur Adreßverwaltung

Es sind drei Arten von Adreßtabellen zu unterscheiden:

1. Eine **prozeßlokale Tabelle** mit den Einsprungadressen aller von dieser Komponente zur Verfügung gestellten Diensten (Abb. 5.2).
2. Ein **prozeßlokaler Cache**, der tabellarisch alle Komponenten umfaßt, zu denen der Prozeß im Laufe seiner Laufzeit schon Verbindungen hergestellt hat (Abb. 5.3).
3. Der **Name Service**, auf dem alle APRICOTS-Komponenten mit ihren Adressen verzeichnet sind und der als Datenbankrelation verwaltet wird. Sie enthält alle Angaben, die nicht für einen einzigen Prozeß spezifisch sind, also insbesondere keine „Henkel" (*handle*), die erst bei einem Verbindungsaufbau generiert werden (Abb. 5.4).

Die Adreßtabellen werden unterschiedlich verwaltet. Die Einsprungadressentabelle und die prozeßlokale Cache-Tabelle werden vom jeweiligen Prozeß, der die in der Einsprungadressentabelle spezifizierten Dienste anbietet und über die in der Cache-Tabelle aufgeführten Verbindungen verfügt, verwaltet. Der Name Service wird von den ConTract Processing Monitoren verwaltet, die auch für die Aktualisierung der Einträge verantwortlich sind.

### 5.3.1 Einsprungadressentabelle

Diese prozeßlokale Tabelle wird vom Kommunikationssystem bei der Verteilung der eingetroffenen Nachrichten benutzt (Empfangsseite). Eine weitere Verwendung ist die Recherche bei der Empfängerbestimmung (Sendeseite).
Für die Komponente *lusthansa_server* aus der Beispielkonfiguration des Abschnittes 1.4 (Abb. 1.5) ist die in der Abb. 4.1 gezeigte Darstellung in eine Einsprungadressentabelle umgesetzt worden (Abb. 5.2).
Ihre Einträge bestehen aus den Feldern für Dienstname, Name der Resource Manager Klasse, Komponentenname und Einsprungadresse. Sie ist statisch und wird zur Übersetzungszeit bzw. Bindungszeit der Module, die in einem Prozeß

[5] Hiervon ausgenommen sind synchrone, prozeßlokale Dienstaufrufe, vgl. Abschnitt 4.3.1.



| service_name | resource_manager_class | component_name | service_address |
|---|---|---|---|
| FLUGDATENEINGABE | LUSTHANSA | lusthansa_server | &flugdateneingabe |
| ... | LUSTHANSA | lusthansa_server | ... |
| FLUGBUCHUNG | LUSTHANSA | lusthansa_server | &flugbuchung |
| GET_GTRID | TM | lusthansa_server | &get_gtrid |
| ... | TM | lusthansa_server | ... |
| CREATE_TA_DEPENDENCY | TM | lusthansa_server | &create_ta_dependency |

Abbildung 5.2: Einsprungadressentabelle der Komponente *lusthansa_server*

zusammengebunden sind, festgelegt.
Die Initialisierung erfolgt durch eine *make*-File,[6] das aus Definitionsdateien einzelner Module die Tabelle zusammenstellt.
Zur Implementierung wird ein Array verwendet, das vom erwähnten *make*-File *make_apricot_component* mit den notwendigen Adreßangaben belegt wird (initialisiertes Array). Das Ende des Arrays wird durch einen Eintrag mit lauter Nullen festgelegt.

### 5.3.2 Lokaler Cache

Der lokale Cache umfaßt die Adreßangaben zu den Verbindungen, die eine Komponente unterhält:

- alle Mailboxen auf demselben Rechner, zu denen der Prozeß schon eine Verbindung aufgebaut hat und an die er Nachrichten versenden kann.
- alle Komponenten auf anderen Rechnern, zu denen der Prozeß über Sockets schon Verbindungen aufgebaut hat.

Er umfaßt also nicht nur die logischen Adressen von Komponenten, sondern auch die dynamisch beim Verbindungsaufbau erzeugten „Henkel" (*handle*), die den „Kanal" für die Verbindung spezifizieren.
Sein Umfang verändert sich dynamisch während der Laufzeit. Ein Verbindungsaufbau zu einer Komponente führt zu einem Neueintrag im Cache, ein Verbindungsabbau zu einer Änderung oder zu einem Austrag aus dem Cache. Eine Verbindungsstörung, die zu einem Neuaufbau der Verbindung führt, kann eine Änderung der von Feldern in einem Eintrag im lokalen Cache bewirken, die die physische Adresse spezifizieren.
Der lokale Cache enthält in seinen Henkeln rein prozeßlokale Information und ist dadurch dem prozeßlokalen Adreßraum zugeordnet. Der dynamische Umfang

[6] Dieses *make*-File ist ein C-Programm und heißt *make_apricot_component*. Die Erstellung einer APRICOTS-Komponente mit *make_apricot_component* ist in Kap. 7 beschrieben.

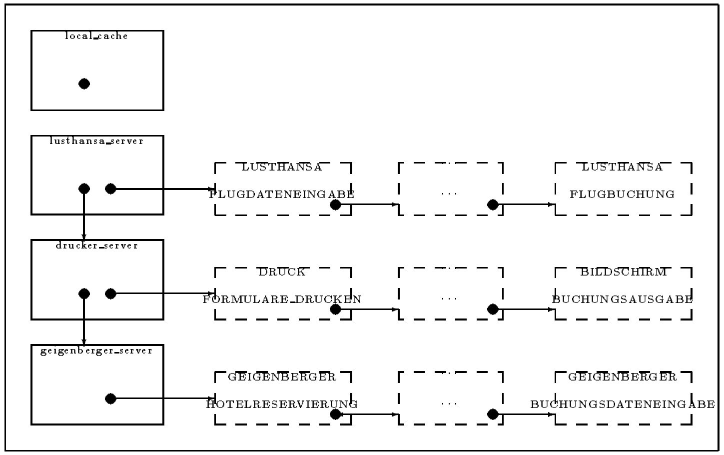


Abbildung 5.3: Prozeßlokaler Cache für *contract_comp*

| service_name | rm_class | component | mbx_address | rpc_address |
|---|---|---|---|---|
| activate_contract | CM | contract_comp | mbx_101 | agamemnon_postman |
| get_gtrid | TM | contract_comp | mbx_101 | agamemnon_postman |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |
| flugbuchung | LUSTHANSA | lusthansa_server | mbx_102 | agamemnon_postman |
| get_gtrid | TM | lusthansa_server | mbx_102 | agamemnon_postman |

Abbildung 5.4: Name Service Tabelle für APRICOTS

erfordert eine dynamische Struktur zu seiner Verwaltung.
Da die Änderungen im lokalen Cache stets komponentenweise erfolgen, wird die Tabelle als Array von Komponenten definiert. Jeder Komponenteneintrag zeigt auf eine Liste der angebotenen Dienste.
Die Abb. 5.3 zeigt den prozeßlokalen Cache für die Komponente *contract_comp* aus der APRICOTS-Konfiguration in Abb. 1.5. Die Einträge sind mit Dienstnamen und Namen der Resource Manager Klassen versehen.
Der Cache-Inhalt deutet an, daß schon zu allen Step Computation Servern der Konfiguration Adreßrecherchen durchgeführt wurden, deren Ergebnisse im Cache eingetragen worden sind.

### 5.3.3 Name Service Tabelle

Die Name Service Tabelle besteht aus einer Datenbankrelation, die nach dem



Schema in Abb. 5.4 aufgebaut ist.
Sie enthält alle angebotenen Dienste des APRICOTS-Systems. Sie verändert sich dynamisch während der Laufzeit unter den folgenden Maßgaben:

- Dienste werden neu eingetragen, wenn neue Komponenten im System generiert werden.
- Dienste werden ausgetragen, wenn Komponenten ihre Arbeit beenden.
- Dienste ändern ihre Adressen, wenn die sie anbietenden Komponenten migriert werden.
- Die Verfügbarkeit von Diensten ändert sich mit der Verfügbarkeit von Prozessen und Rechnern.

Alle Komponenten des APRICOTS-Systems greifen lesend auf die Tabelle zu. Die Name Service Tabelle wird als Datenbankrelation verwaltet. Änderungen werden vom jeweils für eine Komponente zuständigen ConTract Processing Monitor durchgeführt.

## 5.4 Strukturen zur Nachrichtenpufferung

Es werden für jede APRICOTS–Komponente Puffer für eintreffende Nachrichten eingerichtet.
Hierbei wird unterschieden zwischen prozeßlokalen und nicht-prozeßlokalen Nachrichten, da für prozeßlokale Nachrichten keine Interprozeßkommunikationsmechanismen wie z.B. der Mailbox-Mechanismus oder Pipes notwendig sind und somit Synchronisationsmaßnahmen über Prozeßgrenzen hinweg für prozeßlokale Nachrichten entfallen.

### 5.4.1 Warteschlange für prozeßlokale Nachrichten

Diese Warteschlange dient zur Aufnahme prozeßlokaler asynchroner Nachrichten.
Sie wird für jeden Prozeß durch das erwähnte *make*-File einmal definiert.
Sie wird in der Funktion *cs_init_communication_system* initialisiert.
Die Nachrichtenpufferung erfolgt über eine FIFO-Strategie.
Der Zugriff ist nur prozeßlokal möglich über einen Dienstaufruf bzw. die Nachrichtenverteilerschleife in *main*.

### 5.4.2 Mailbox für Nachrichten von anderen Prozessen

Jede APRICOTS-Komponente verfügt über eine eigene Mailbox für den Empfang nicht-prozeßlokaler Nachrichten.
Rechnerlokale Nachrichten werden direkt vom Sender in die Empfängermailbox eingetragen. Entfernte Nachrichten werden vom Sender an den Briefträger auf

dem Zielrechner geschickt, der die Nachricht dann in die Mailbox des Empfängers einträgt.
Die Verwaltung der Nachrichten erfolgt über eine FIFO-Strategie. Diese FIFO-Strategie wird bei nicht prozeßlokalen, synchronen Dienstaufrufen umgangen, wenn eine Komponente auf die Antwort wartet.
Die Realisierung beruht auf dem in [7] entwickelten Mechanismus. Die Mailbox ist als Shared Memory Area implementiert. Die Zugriffssynchronisation erfolgt über Semaphore. Änderungen erfolgen über einen Shadowing-Mechanismus.
Erzeugung und Auflösung von Mailboxen stehen unter Kontrolle des jeweilig zuständigen ConTract Processing Monitors.

### 5.4.3 Briefträger

Der Briefträger wird vom Sender via Socket angesprochen. Die Eingabewarteschlange des Briefträgers ist eine ins Betriebssystem integrierte Datenstruktur. Der Briefträger holt Nachrichten aus seiner Eingabewarteschlange und trägt sie in die Mailbox der Empfängerkomponente ein. Danach quittiert er dem Sender den Empfang.
Der Briefträger wird von dem ConTract Processing Monitor, der den Briefträger starten und stoppen kann, verwaltet.
Um seine Aufgabe erfüllen zu können, erhält der Briefträger vom ConTract Processing Monitor die Mailbox-Adresse jeder neu gestarteten Komponente mitgeteilt und baut eine Verbindung zu deren Mailbox auf.
Die Bearbeitung der Nachrichten in der Eingabewarteschlange des Briefträgers erfolgt nach einer reinen FIFO-Strategie.

# Kapitel 6

# Fehlerbehandlung

Das hier dargestellte Fehlermodell ist der Versuch, alle Unbill, die in einem verteilten System die Integrität des Nachrichten- und damit auch des Kontrollflusses beeinträchtigt, von den Anwendungen des Kommunikationssystems fernzuhalten.
Die Prämissen, die hierbei Analyse und Entwurf begleiten, sind die Robustheit, die Zuverlässigkeit und die Verfügbarkeit des Kommunikationssystems.
Robustheit bedeutet, daß Fehler aufgrund von Leitungsstörungen oder Prozeß- bzw. Rechnerausfällen vom Kommunikationssystem abgefangen werden.
Zuverlässigkeit bedeutet, daß keine Nachricht verloren geht oder unvollständig bzw. dupliziert beim Empfänger oder gar bei einer nicht adressierten Komponente erscheint.
Verfügbarkeit bedeutet, daß eine Nachricht auch bei kurzfristigem Ausfall einer Komponente zugestellt wird.

Dieses Kapitel wird sich zunächst mit einer Klassifikation der möglichen Fehler befassen. Daran schließt sich eine Kurzbeschreibung der einzelnen Fehlerklassen. Danach folgt eine Beschreibung der notwendigen Logging- und Recovery-Maßnahmen.

Dieses Kapitel ist eine Darstellung des Entwurfes zur Fehlerbehandlung. Implementiert wurden bislang nur Routinen zur Fehlerbehandlung im Bereich der Basiskommunikation.

## 6.1 Klassifikation

Es lassen sich zunächst zwei große Klassen nach dem Wesen des auftretenden Fehlers bilden:

- Fehler aufgrund von Verstößen seitens eines Anwendungsprogramms, also gewissermaßen aufgrund von Verbotsübertretungen.
  Sie treten auf, wenn ein ungültiger Name für einen Dienst oder eine Resource Manager Klasse an das Kommunikationssystem übergeben wurde

oder wenn das aufrufende Programm nicht über die Berechtigung, das APRICOTS-System zu benutzen, verfügt.
Gemeinsam ist all diesen Fehlern, daß sie keine Ausweichstrategie zulassen, sondern daß der Aufruf vom Kommunikationssystem zurückgewiesen werden muß.
Es handelt sich also um nicht behebbare Fehler, die zu einer sofortigen Rückgabe der Kontrolle an das aufrufende Programm führen.

- Fehler aufgrund von Störungen, Inkonsistenzen oder Ausfällen innerhalb des APRICOTS-Systems bzw. der die Konfiguration verwaltenden Prozesse und ihrer Datenstrukturen.
  Zu nennen sind hier Komponenten- oder Rechnerausfälle oder Nichtverfügbarkeit des ConTract Processing Monitors.
  Bei dieser Gruppe handelt es sich im allgemeinen um behebbare Fehler. Der Schwerpunkt der Betrachtung wird, da hier auch die Zuverlässigkeit, Verfügbarkeit und Robustheit des Systems selbst betroffen sind, im folgenden auf dieser Fehlergruppe liegen.

Nicht zu vernachlässigen sind Fehler des Programms selbst, die naturgemäß stets auftreten, nie gegen Null konvergieren, aber an unvorhersehbaren Stellen Unheil anrichten.
Die Fehlerbehandlung verfolgt zwei Ziele:

1. Die Fehlerbehebung bei vorübergehenden „lokalen" Störungen wird durch Fehlerbehandlungsroutinen erreicht.
2. Nach teilweisen oder vollständigen Systemzusammenbrüchen wird mittels der Logging-Einträge ein Forward Recovery durchgeführt.

Beide Fälle überschneiden sich zum Teil. In beiden Fällen muß das Kommunikationssystem einen konsistenten Zustand wiederherstellen.

## 6.2 Fehlerbehandlung

Zunächst wird die Klasse der behebbaren Fehler weiter differenziert, wobei zu den einzelnen Fehlerklassen die Fehlerbehandlung kurz beschrieben wird.

#### Komponentenfehler

Hierunter fallen die Störungen Komponentenausfall, Rechnerausfall und Migration.
Ein Komponentenausfall wird vom zuständigen ConTract Processing Monitor durch einen Software-Interrupt abgefangen. Er startet die ausgefallene Komponente neu und versorgt sie mit den notwendigen Initialisierungsinformationen. Die Mailbox dieser Komponente überlebt den Absturz.
Ein Sonderfall ist der Ausfall eines ConTract Processing Monitor. Hier muß ein



Überwachungsmonitor auf einem stabilen Rechner eingesetzt werden, um diese Störung zu erkennen. Dies gilt gleichfalls für den Ausfall eines Knotens.
Eine Migration ist eine kontrollierte Konfigurationsänderung, die unter Kontrolle des ConTract Processing Monitors vorgenommen wird. Durch eine Migration kann es zu Leitungsfehlern kommen, die durch veraltete Adressen in den prozeßlokalen Cache-Tabellen hervorgerufen sind.
Ein ungelöstes Problem ist die gegenseitige Beeinflussung von Resource Managern innerhalb einer Komponente. Da eine Komponente einem Prozeß entspricht, also aus Sicht des Betriebssystems eine Einheit der Betriebsmittelvergabe darstellt, können Resource Manager einer Komponente wechselseitig ihre privaten Daten korrumpieren.
Dieses Fehlverhalten kann vom Kommunikationssystem nicht unterbunden werden, da es auf Betriebssystemebene keine Möglichkeit zur Isolierung von Modulen innerhalb eines Prozesses gibt.

#### Leitungsfehler

Wenn eine Verbindung unterbrochen oder einseitig aufgelöst wurde bzw. die verwendete Adresse durch Migration veraltet ist, wird durch das Protokoll der verwendeten Mechanismen in der Basiskommunikationsschicht eine Fehlermeldung mit Ursachenspezifikation zurückgeliefert. I. a. wird das Problem mit einem neuen Verbindungsaufbau bzw. einer Anfrage beim Name Service und einer erneuten Übertragung behoben.
Ist der Empfänger vom Sender spezifiziert worden, so wird die Nachricht als gescheitert an den Sender zurückgegeben.

#### Warteschlangenfehler

Wenn eine Mailbox bzw. eine Socket-Warteschlange überfüllt ist, so erhält der Sender eine entsprechende Nachricht zurück. Das Kommunikationssystem versucht dann, einen neuen Empfänger zu bestimmen und die Nachricht erneut zu übertragen.

#### Nachrichtenfehler

Wenn eine Konversionsroutine für einen Datentyp fehlerhaft ist, so kann das Kommunikationssystem keine Fehlerbehebung durchführen. Dieser Fall tritt jedoch nur bei der Übertragung zwischen Rechnern mit unterschiedlicher Hardware-Kodierung auf.

#### Betriebssystem

Wenn Systemquoten überschritten werden, dann kann dies vom Kommunikationssystem nur durch Ausnahmebehandlungsroutinen abgefangen werden. Im allgemeinen erfordert eine Verweigerung von Resourcen seitens des Betriebssystems jedoch eine Rekonfiguration von APRICOTS oder eine Erhöhung der Quoten für APRICOTS.[1]

**Deadlocks**

Deadlocks können nur bei synchronen Dienstaufrufen auftreten und dies auch nur im nicht-prozeßlokalen Fall.
Im rechnerlokalen Fall wird die Nachricht nach einer bestimmten Wartezeit dann wieder durch den Sender aus der Mailbox des Empfängers geholt und ein neuer Empfänger bestimmt.
Im entfernten Fall muß der Briefträger des Zielrechners beauftragt werden, die Nachricht aus der Mailbox der Empfängerkomponente zu entfernen.
Beide Verfahren finden nur Anwendung, wenn der Empfänger nicht zwischenzeitlich doch die Verarbeitung der Nachricht begonnen hat, was durch eine Abfrage des Nachrichtenzustandes bei der Empfängerkomponente geklärt wird.

## 6.3 Logging und Recovery

Logging und Recovery gehören eng zusammen. Ein Logging, also die Protokollierung von konsistenten Zuständen bzw. Zustandsübergängen, ist die Voraussetzung für ein Recovery, d. h. die Wiederherstellung eines konsistenten Zustandes nach einem teilweisen oder vollständigen Systemzusammenbruch.
Die Recovery bestimmt dabei unter zwei Aspekten die Aufgaben des Kommunikationssystems:

1. Da ConTracts langlebig sind, dürfen sie im Fehlerfall nicht global zurückgesetzt werden, sondern es muß ein möglichst weit fortgeschrittener, konsistenter Arbeitszustand wiederhergestellt werden (Forward Recovery). Wegen der verteilten Ausführung eines ConTracts gehört dazu auch der Zustand der Verwaltungsstrukturen für die Nachrichtenpufferung (Vermeidung der Duplikation oder des Verlustes von Nachrichten).

2. Da Dienstaufrufe als Transactional Remote Procedure Calls (TRPC) realisiert sind, müssen sie mehrere Anforderungen erfüllen.[2] U. a. sind Dienstaufrufe Bestandteil von Transaktionen. Das Kommmunikationssystem muß deshalb sicherstellen, daß keine transaktionsrelevante Information verloren geht und die Aufrufe protokollieren.

### 6.3.1 Logging

Ein Logging muß einerseits garantieren, daß sich im Fehlerfall beim Wiederanlauf ein konsistenter Zustand wiederherstellen läßt, der ein Weiterarbeiten ohne

[1] Beispielsweise können nur zehn Mailboxen unter einem Benutzernamen in SunOS installiert werden.
[2] Vgl. hierzu Abschnitt 2.5 und [11].



Informationsverluste ermöglicht. Andererseits darf das Logging nicht exzessiv ausgeweitet werden, da jeder Logging-Eintrag mindestens einen Plattenzugriff, d.h. Zeit kostet.
Damit lautet die Forderung an das Logging: „so viel wie nötig, aber so wenig wie möglich."
Einen Dienstaufruf durch Protokollierung zu klammern, erweist sich als sehr ineffizient, da dadurch auch prozeßlokale Dienstaufrufe, die auf einfache Funktionsaufrufe rückführbar sind, zu mindestens zwei, im asynchronen Fall zu vier Protokollaufrufen führen.[3]
Sinnvoller scheint es, die Ein- bzw. Austragung aus einer Nachrichtenwarteschlange zu protokollieren. Dadurch lassen sich die Strukturen zur Nachrichtenpufferung nach einem teilweisen oder vollständigen Ausfall wiederherstellen.

**Logging im prozeßlokalen Fall**

Für einen prozeßlokalen, synchronen Dienstaufruf findet kein Logging statt.

Für einen prozeßlokalen, asynchronen Dienstaufruf muß jeweils das Eintragen in die bzw. Austragen aus der prozeßlokalen Nachrichtenwarteschlange protokolliert werden.
Der Ablauf ist

- log_put_message_into_local_queue (Auftraggeberseite)
- log_fetch_message_from_local_queue (Auftragnehmerseite)
- log_put_message_into_local_queue (Auftragnehmerseite)
- log_fetch_message_from_local_queue (Auftraggeberseite)

**Logging im rechnerlokalen Fall**

Bei einem rechnerlokalen, synchronen Dienstaufruf fallen Protokollaufrufe für das Eintragen in die Mailbox bzw. das Austragen aus der Mailbox an:

- log_put_message_into_mailbox (Auftraggeberseite)
- log_fetch_message_from_mailbox (Auftragnehmerseite)
- log_put_message_into_mailbox (Auftragnehmerseite)
- log_fetch_message_from_mailbox (Auftraggeberseite)

Für rechnerlokale, asynchrone Dienstaufrufe ist der Ablauf für die Übertragung auf Basiskommunikationsebene völlig identisch zu einem synchronen Aufruf. Der einzige Unterschied ist, daß die Auftraggeberkomponente nicht die Antwort abwartet. Deshalb werden dieselben Protokollaufrufe wie im rechnerlokalen, synchronen Fall verwendet.

[3] Für einen prozeßlokalen, asynchronen Dienstaufruf sind zwei Aufrufe von *RM_call_async* notwendig (siehe dazu Kap. 3).

### Logging im entfernten Fall

Für den entfernten Fall gilt dasselbe wie für den rechnerlokalen Fall. Synchrone und asynchrone Aufrufe durchlaufen dieselben Stationen bei der Übertragung:

- log_put_message_to_postman (Auftraggeberseite)
- log_put_message_into_mailbox (Briefträger)
- log_fetch_message_from_mailbox (Auftragnehmerseite)
- log_put_message_to_postman (Auftragnehmerseite)
- log_put_message_into_mailbox (Briefträger)
- log_fetch_message_from_mailbox (Auftraggeberseite)

### Logging-Protokoll

Zu protokollieren ist dabei für alle geschilderten Fälle jeweils (mindestens) der Nachrichtenhenkel, die datenspezifischen Felder *data_type*, *message_length* und *message_buffer* der APRICOTS-Übertragungseinheit sowie die Identifikation der Komponente, an dem sich die Nachricht nach der Änderung befindet.
Aus diesen Angaben läßt sich die Nachricht rekonstruieren, da der Nachrichtenhenkel die Adreßangaben für Auftraggeberdienst und Auftragnehmerdienst enthält. Weiterhin sind im Nachrichtenhenkel der Verarbeitungszustand des Auftrags und der globale Transaktionsbezeichner festgehalten.[4]

## 6.3.2 Recovery

Das Wiederanlaufverfahren nach einem teilweisen oder vollständigen Systemzusammenbruch muß die Verwaltungsstrukturen zur Nachrichtenpufferung wieder in einen konsistenten Zustand bringen.
Dazu werden alle Nachrichteneinträge zu Aufträgen, deren Bearbeitung noch nicht abgeschlossen ist, aus der Protokolldatenbank erfragt. Erkennbar ist das am Eintrag „Verarbeitungszustand" im Nachrichtenhenkel.
Aus diesen „Treffern" muß derjenige Protokolleintrag zu einem Auftrag ausgewählt werden, der zuletzt eingetragen wurde. Das gemeinsame Merkmal aller Protokolleinträge eines einzigen Dienstaufrufs ist dabei das Paar

<*component_name*, *message_counter*>.

Der Protokolleintrag enthält den Namen der Komponente, die die Nachricht zuletzt erhalten hat. Dieser Komponente wird die Nachricht in die Mailbox eingetragen.
Das Verfahren stellt einen konsistenten Zustand der Nachrichtenwarteschlangen her:

[4] Der Aufbau eines Nachrichtenhenkels wird in Kap. 5 beschrieben.



- Jede vor einem Zusammenbruch adressierbare, weil aktive Komponente wird beim Wiederanlauf reaktiviert.
- Die Reduktion auf logische Namen von Diensten und Komponenten im Nachrichtenhenkel garantiert, daß beim Wiederanlauf nicht veraltete Adressen benützt werden, sondern durch eine Anfrage beim Name Service die aktuelle Adresse erfragt wird.[5]
- Wird ein Dienstaufruf in die Mailbox einer Komponente statt in deren prozeßlokale Nachrichtenwarteschlange eingetragen, so ändert das an der Adressierung der Antwort nichts.[6]

[5] Infolge von Rechnerausfällen kann es zu Migrationen von Komponenten kommen.

[6] Eine Nachricht in der prozeßlokalen Queue ist prozeßlokal und asynchron. Die Antwort erfolgt durch *RM_call_async*. Die Empfängerbestimmung ergibt dann Identität von Auftragnehmer- und Auftraggeberkomponente.

# Kapitel 7

# Resümee

## 7.1 Zusammenfassung

Ich will hier an dieser Stelle noch einmal die wichtigsten Eigenschaften des Resource Manager Calls zusammenfassend darstellen, indem ich sie gegenüber dem Konzept des Remote Procedure Calls abgrenze. Die folgenden Punkte werden vor allem unter dem Gesichtspunkt betrachtet, welche Konsequenzen das Modell des Resource Manager Calls für die Programmierung und Abwicklung langlebiger und verteilter Datenbankanwendungen hat.

**Ortstransparente Adressierung:** Ein RPC ist als **server call**, ein Dienstaufruf in APRICOTS jedoch als **service call** realisiert. Die Service-Server-Zuordnung wird dynamisch vom Kommunikationssystem durchgeführt.
Dadurch wird auf Systemseite eine flexible Prozeßkonfiguration unterstützt, die es erlaubt, Migrationen und Lastbalancierungsmaßnahmen transparent für die Anwendungen durchzuführen.
Auf der Anwendungsseite wird die Programmierung unabhängig von der tatsächlichen Konfiguration und damit entlastet.

**Service-Redundanz:** Dienste können in der APRICOTS-Konfiguration mehrfach realisiert werden. Durch die Ortstransparenz eines Dienstaufrufs kann das Kommunikationssystem beim Scheitern der adressierten Komponente eine Ersatzkomponente für die Dienstausführung auswählen und mit der Ausführung beauftragen, ohne daß die Anwendung dies erfährt.
Diese Fehlerredundanz erleichtert die Programmierung für Steps und Skript, weil die Fehlerbehandlung in das Kommunikationssystem integriert ist.

**Integration prozeßlokaler Aufrufe:** Remote Procedure Calls sind per definitionem ein Mittel zur Interprozeßkommunikation. Das Konzept des Resource Manager Calls ermöglicht prozeßlokale, rechnerlokale und entfernte Aufrufe. Dabei wird für jeden Typ ein effizienter Basiskommunikationsmechanismus für die Übertragung vom Sender zum Empfänger gewählt.





Erst die Möglichkeit, prozeßlokale Aufrufe über dieselbe Schnittstelle abzusetzen wie Aufrufe an andere Prozesse, realisiert die vollständige Ortstransparenz mit den oben beschriebenen Vorteilen.

**Wechselseitige Client-Server-Beziehungen:** Das RPC-Konzept erlaubt nur statisch festgelegte, einseitige Client-Server-Beziehungen. Implementierungen des Remote Procedure Calls wie beispielsweise diejenige unter SunOS ermöglichen zwar Duplex-Verbindungen, die Programmierung ist aber vergleichsweise aufwendig und erfordert eingehende Kenntnisse der Betriebssystemmechanismen.
Für die ConTract-Abwicklung sind jedoch wechselseitige Dienstaufrufe zur Ablaufsteuerung wichtig. Deshalb sind Resource Manager Calls für dynamisch veränderliche, geschachtelte und wechselseitige Client-Server-Beziehungen ausgelegt. Jede Komponente ist gleichberechtigt.
Die Skript- und die Step-Programmierung sind dadurch völlig unabhängig von der Berücksichtigung dynamisch entstehender Aufrufstrukturen bei der ConTract-Abwicklung.

**Ereignisorientierung:** Der Remote Procedure Call ist eine Erweiterung des prozeßlokalen Funktionsaufrufs und damit synchroner Natur. Es existieren zwar Implementierungen asynchroner Remote Procedure Calls, jedoch bedürfen diese eines blockierenden „*await_answer*" seitens des Aufrufers und erzwingen damit ein synchrones Warten auf das Ende des Aufrufs.
Der vorgestellte Kommunikationsmechanismus bietet einen vollständig asynchronen Dienstaufruf an, bei dem die Antwort des aufgerufenen Dienstes selbst wieder als Dienstaufruf an den ursprünglichen Aufrufer erfolgt. Ermöglicht wurde dies durch die wechselseitigen Client-Server-Beziehungen.
Die Konsequenzen sind zum einen die Vermeidung von Wartesituationen. Zum anderen wird dadurch das Event-Trigger-Konzept bei der ConTract-Abwicklung auf den Kommunikationsmechanismus übertragen. Ein beendeter Dienst informiert mit seinem Dienstaufruf den Aufrufer über das Ereignis „Dienst beendet".

**Transactional Remote Procedure Call:** Herkömmliche Konzepte der Interprozeßkommunikation bieten zwar auf der Transportschicht des OSI-7-Schichten-Referenzmodells Protokolle an, die Datenverlust bzw. -replikation ausschließen. Transaktionsschutz für Nachrichten, im Sinne von „recoverable service calls", ist jedoch nicht realisiert.
Der Resource Manager Call unterstützt das Transaktionsprotokoll und stellt durch Logging sicher, daß nach einem Systemzusammenbruch die Forward Recovery, die für langlebige Anwendungen erwünscht ist, auch auf der Ebene der Nachrichtenübertragung erfolgen kann.

Der Kommunikationsmechanismus orientiert sich in Entwurf und Realisierung an den spezifischen Anforderungen, die die ConTract-Abwicklung an die Synchronisation verteilter Datenbankanwendungen stellt und unterstützt die Sicherung langlebiger Aktionen.

| Mechanismus | durchschnittliche Zeit pro Aufruf | | | |
|---|---|---|---|---|
| | Durchlauf | | | Schnitt |
| | 1. | 2. | 3. | |
| Funktionsaufruf | 1,74 $\mu s$ | 1,74 $\mu s$ | 1,75 $\mu s$ | 1,74 $\mu s$ |
| prozeßlokaler RM-Call | 417 $\mu s$ | 357 $\mu s$ | 387 $\mu s$ | 387 $\mu s$ |
| rechnerlokaler RM-Call | 1,13 $ms$ | 1,29 $ms$ | 0,90 $ms$ | 1,11 $ms$ |
| rechnerlokaler RPC | 6,42 $ms$ | 6,62 $ms$ | 8,45 $ms$ | 7,16 $ms$ |

Tabelle 7.1: Leistungsmessung für das APRICOTS-Kommunikationssystem

In der Abstraktion von den physischen Gegebenheiten der Datenübertragung weit über dem Remote Procedure Call einzuordnen, implementiert der Resource Manager Call die Anwendungsebene des OSI-7-Schichten-Referenzmodells und ermöglicht dadurch eine von Kommunikationsaspekten entlastete Programmierung für die ConTract-Abwicklung.

## 7.2 Leistungsmessung

Jedes Konzept erweist sich erst dann als praktikabel und vernünftig, wenn seine Effizienz unter Beweis gestellt werden kann. Aus dieser Überlegung heraus ist das vorgestellte Kommunikationssystem einer Leistungsmessung unterzogen worden.
Die Ergebnisse basieren auf Messungen, die auf einer SPARCstation ELC vorgenommen wurden.
Dem Stand der Implementierung entsprechend wurden Resource Manager Calls für den prozeßlokalen und den rechnerlokalen Fall gemessen. Die Untersuchung war auf synchrone Aufrufe beschränkt, um sie adäquaten Vergleichsmechanismen gegenüberstellen zu können. Zum Vergleich wurden ein einfacher Funktionsaufruf und ein rechnerlokaler Remote Procedure Call herangezogen.
Übertragen bzw. übergeben wurde in allen vier Fällen eine vollständige APRICOTS-Übertragungseinheit, die 756 Bytes umfaßt und ohne weitere Bearbeitung sofort wieder an den Aufrufer zurückgegeben wurde.
Es wurden drei Messungen mit jeweils 1000 Aufrufen für jeden Mechanismus vorgenommen. Die Ergebnisse sind in der Tab. 7.1 zusammengestellt.

Der prozeßlokale Dienstaufruf benötigt durchschnittlich etwa 225-mal mehr Zeit als der direkte Funktionsaufruf. Das erscheint auf den ersten Blick sehr unbefriedigende, relativiert sich jedoch unter Berücksichtigung der Tatsache, daß in die bisher vorliegende Implementierung noch wenig Optimierungsmaßnahmen eingeflossen sind.
Weiterhin sind Zahlen in dieser Größenordnung nicht sehr aussagekräftig, wenn sie nicht den Zeiten für reale Aufrufe mit Ausführung tatsächlicher Aktivitäten gegenübergestellt werden.
Letztendlich muß angemerkt werden, daß der Vorteil einer einheitlichen Schnitt-



stelle für Aufrufe innerhalb eines Prozesses und zwischen verschiedenen Prozessen nicht ohne Nachteile in der Effizienz für prozeßlokale Aufrufe zu haben ist.

Bemerkenswert ist das Ergebnis für den rechnerlokalen Dienstaufruf, der mithilfe eines Mailboxmechanismus im gemeinsamen Speicherbereich implementiert wurde. Er benötigt nur knapp dreimal so viel Zeit wie der prozeßlokale Aufruf. Die Streuung der Meßergebnisse für den rechnerlokalen Aufruf hält sich im Rahmen derjenigen für den prozeßlokalen Aufruf. Der Mechanismus ist folglich in seiner Leistung relativ stabil, sofern diese kleine Meßreihe diese Prognose erlaubt.

Der Remote Procedure Call erweist sich mit dieser Messung als äußerst ineffizienter Übertragungsmechanismus für die rechnerlokale Kommunikation. Er benötigt etwa das Sechsfache eines rechnerlokalen Dienstaufrufes.
Die relativ hohe Streuung der Meßergebnisse läßt auch eine Abhängigkeit von Faktoren wie z. B. der Netzauslastung vermuten, die noch genauer zu untersuchen ist.
Bestätigt sich diese Annahme, so ist es auch zum Nutzen der Kommunikation innerhalb von APRICOTS, die Netzbelastung nicht unnötig zu erhöhen und nur im „Notfall" auf einen Remote Procedure Call als Übertragungsmechanismus zurückzugreifen.

## 7.3 Stand der Arbeit und Ausblick

Bis zum Abschluß dieser Diplomarbeit konnte nur eine Ein-Rechner-Mehr-Prozeß-Lösung implementiert werden, die auch als Basis der vorgestellten Leistungsmessungen gedient hat.
Der Entwurf für eine Mehr-Rechner-Lösung auf Sun-Workstations ist weitgehend abgeschlossen.

Darüberhinaus konnte noch ein *make*-File zur Erzeugung von ablauffähigen APRICOTS-Komponenten erstellt werden: *make_apricot_component* generiert aus einem oder mehreren Resource Manager Source-Files den Code für eine Komponente. Notwendig sind dazu die Definitionen der angebotenen Dienste der jeweiligen Resource Manager, die in Definitionsdateien dem *make*-File übergeben werden.
Das Manual für *make_apricot_component* wird in Bälde online verfügbar sein.

Einige anstehende Aufgaben bzw. noch offene Fragen seien hier kurz angeführt.
Die Fehlerbehandlung ist bislang nur auf der Basiskommunikationsebene realisiert worden.
Eine weitergehende Fehlerbehandlung erfordert eine Protokollierung und einen Entwurf für das Wiederanlaufverfahren nach einem Systemzusammenbruch.

Eventuell kann noch ein klassischer asynchroner Aufruf mit *await_answer()*-Funktionalität in das System integriert werden, um die Anwendbarkeit der

Kommunikationsmechanismen für asynchrone Aufrufe universeller zu gestalten.

Weiterhin sind noch einige Features sehr einfach implementiert und können noch ausgebaut werden.
So erfolgt die Empfängerbestimmung nicht nach Lastbalancierungskriterien, sondern gehorcht einem simplen Algorithmus der Art „nimm die erste verfügbare Komponente, die dem Aufrufer am nächsten ist".
Hier bietet es sich an, noch zusätzliche Komponenten zu modellieren, die in der Lage sind, eine „intelligente" Lastbalancierung vorzunehmen, um die ConTract-Abwicklung unter realen Bedingungen zu beschleunigen.

Ein ungelöstes Problem ist die Deadlock-Vermeidung.
Deadlocks bei der Kommunikation in einer verteilten Umgebung zu erkennen und zu beheben ist sehr komplex und aufwendig, da dazu die Verwaltung aller synchronen Wartesituationen im Netz notwendig ist und mit allen beteiligten Transaction Manager eine koordinierte Rücksetzung bestimmter Komponenten durchzuführen wäre.
Deshalb sind keine Deadlock-Erkennungsalgorithmen implementiert. Ursprünglich vorgesehene Restriktionen wie das Verbot synchroner, nichtprozeßlokaler Aufrufe schränken jedoch das Verarbeitungsmodell zu sehr ein.
Die Verantwortung für die Deadlockvermeidung liegt somit bei der Skriptprogrammiererung.

# Literaturverzeichnis


[1] A. L. Ananda, B. H. Tay, and E. K. Koh. Astra - an asynchronous remote procedure call facility. In *11. International Conference on Distributed Computing Systems*, Arlington, Mai 1991.

[2] M. J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, Englewood-Cliffs, 1986.

[3] Bernstein, W. P.A., Emberton, and V. Trehan. Decdta - digital's distributed transaction processing architecture. *Digital Technical Journal*, 3(1), Winter 1991.

[4] A. Birrell et al. Grapevine. an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[5] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):33–59, Februar 1984.

[6] A. P. Black and Y. Artsy. Implementing location independent invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119, Januar 1990.

[7] H. Burkert. Systemarchitektur und Betriebsmodell für das verteilte Transaktionssystem APRICOT. Diplomarbeit Nr. 812; Institut für Parallele und Verteilte Höchstleistungsrechner , Stuttgart, August 1991.

[8] R. N. Chang and C. V. Ravishankar. A service aquisition mechanism for the client/server model in cygnus. In *11. International Conference on Distributed Computing Systems*, Arlington, Mai 1991.

[9] DEC. *DEC*.

[10] R. P. Draves and D. S. Thomson. *Camelot and Avalon. A Distributed Transaction Facility*, chapter Interprocess Communication in Camelot, pages 111–118. Eppinger, J. L. and others, December 1989. Script.

[11] J. Gray and A. Reuter. *Transaction Processing Systems: Concepts and Techniques*. Morgan Kaufman Publishers, 1992. First Draft.

[12] F.-J. Kauffels. *Rechnernetzwerksystemarchitekturen und Datenkommunikation*. BI, Mannheim, 1989. kopieren 174-201,222-297,338-453.

[13] C. Koelbel, F. Lamaa, and B. Bhargava. Efficient implementation of modularity in raid. In *USENIX. Distributed And Multiprocessor Systems Workshop Proceedings*, pages 127–143, Fort Lauderdale, Oktober 1989.

[14] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN o88 Conference on Programming Language Design and Implementation*, Atlanta, Juni 1988.

[15] K. Meyer-Wegener. *Transaktionssysteme*. Teubner, Stuttgart, 1988.

[16] L. Richter. *Betriebssysteme*. Teubner, Stuttgart, 1985.

[17] M. Santifaller. *TCP/IP und NFS in Theorie und Praxis*. Addison-Wesley, Bonn München, 1990.

[18] P. D. Stout. *Camelot and Avalon. A Distributed Transaction Facility*, chapter The Design of the Camelot Communication Manager, pages 319–323. Eppinger, J. L. and others, December 1989. Script.

[19] A. Tevanian et al. Mach threads and the unix kernel: The battle for control. Technical Report CMU-CS-87-149, Carnegie Mellon University, August 1987.

[20] H. Wächter. Contracts. a means for improving reliability in distributed computing. In *IEEE Spring Computer Conference*, 1991.

[21] H. Wächter and A. Reuter. Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik Forschung und Entwicklung*, 1990(5):202–212, 1990.

[22] H. Wächter and A. Reuter. The contract model. Institut für Parallele und Verteilte Höchstleistungsrechner; Stuttgart, 1991.

[23] Wächter, H. et al. APRICOTS Produktblätter zum Design Review, September 1991. Institut für Parallele und Verteilte Höchstleistungsrechner; Stuttgart.

[24] G. Weck. *Prinzipien und Realisierung von Betriebssystemen*. Teubner, Stuttgart, 1985.

# Anhang A

# Schnittstellen: Anwendungsebene

Die folgenden Kapitel stellen das APRICOTS-Kommunikationssystem aus programmtechnischer Sicht dar. Sie wenden sich sowohl an NutzerInnen des Kommunikationssystems wie auch auch an ProgrammiererInnen. Deshalb ist dieser Teil der Arbeit in Form eines Handbuchs aufgebaut, das die Beschreibung der Schnittstelle, Arbeitsweise und Umgebung jeder Funktion umfaßt.
Das Handbuch ist in drei Abschnitte gegliedert. Der erste beschreibt die den Anwendungen zugänglichen Protokolle, danach folgen die Protokolle der einzelnen Schichten des Kommunikationssystems, wie sie im Kapitel „Ablaufschema eines RM_Calls" beschrieben sind. Abgeschlossen wird das Handbuch durch eine Beschreibung der Zugriffsprimitive für die Datenstrukturen des Kommunikationssystems.
Innerhalb jedes Eintrags erfolgt eine Gliederung nach

- Schnittstelle mit Parameterbeschreibung (Typ, Name, Ein- oder Ausgabeparameter, Erläuterung)
- Beschreibung der Funktionsweise
- Erläuterungen zur Parameterübergabe und weitere Anwendungshinweise
- Rückgabewerte und ihre Bedeutung
- Aufrufhierarchie
- Verweise und Nachweise

Es werden die Schnittstelle zu einem Resource Manager Call (*RM_call*) als Aufruf eines APRICOTS-Dienstes, sowie die Parametrisierung und korrekte Anwendung eines APRICOTS-Dienstes mit Rückkehr von diesem Dienst (*RM_call_return*) eingehend beschrieben. Darüberhinaus wird der Aufbau der Routine *main()*, die vom Kommunikationssystem gestellt und in den Quelltext einer APRICOTS-Komponente eingebunden wird, vorgestellt.

## A.1 main (CS)

### Name

**main** Hauptroutine in jedem Prozess des APRICOTS-Systems

### Aufruf

**int**
**main (argc, argv)**
**int argc (In)**
Argumentzähler
**char *argv[ ] (In)**
Argumentliste

### Beschreibung

*main* führt je nach Komponententyp unterschiedliche Funktionen aus.
Ist der gestartete Prozeß ein ConTract Processing Monitor, so werden die rechnerlokalen Datenstrukturen initialisiert. Der ConTract Processing Monitor generiert seine Mailbox und seinen „Briefträger"-Prozeß.[1]
Ansonsten führt *main* die Anmeldung beim zuständigen ConTract Processing Monitor durch und richtet die prozeßlokalen Datenstrukturen für den Zugriff auf das Kommunikationssystem ein.
Anschließend geht *main* für jeden Komponententyp in eine Endlosschleife, in der eintreffende Nachrichten bearbeitet und an den adressierten Empfängerdienst weitergeleitet werden.

### Anwendung

*main* ist eine Routine des APRICOTS Shared Library Systems und wird in jeden APRICOTS-Prozeß eingebunden.
Das Array *argv* enthält für Komponenten, die nicht als ConTract Processing Monitor gestartet werden, die folgenden Argumente:

**$argv_0$** Programmname der gestarteten Komponente

**$argv_1$** Komponentenname der gestarteten Komponente

**$argv_2$** Mailbox-Id der Komponente als String

**$argv_3$** Mailbox-Id des ConTract Processing Monitor als String

**$argv_4$** Name Service-Id als String

### Returncodes

*main* kehrt nicht mehr zurück außer durch Prozeßabbruch.

### Wird aufgerufen von:

—

### Ruft auf:

[1] Der „Briefträger"-Prozeß ist noch nicht implementiert, sondern liegt erst in einer Spezifikation vor.



cs_activate_service
cs_call_authenticate
cs_decode_tm_unit
cs_encode_tm_unit
cs_init_communication_system
cs_lookup_message
cs_return_message

### Verweise

RM_call_async (CS), RM_call_sync (CS), <service_name> (APRICOTS)

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## A.2 RM_call_async (CS)

### Name

**RM_call_async** Senden einer asynchronen Nachricht an einen spezifizierten Dienst innerhalb des APRICOTS-Systems

### Aufruf

**CS_RETURNCODE**
**RM_call_async (&msg_handle, &caller, &callee, gtrid, fct, callback_fct, rm_class, local, &data_type, &msg_len, &msg_buf)**

**CS_MESSAGE_HANDLE *msg_handle (Out)**
Zeiger auf den eindeutigen Henkel für die Nachricht

**CS_ADDRESS_PARAM caller (In)**
Adreßkontrollblock des Senders

**CS_ADDRESS_PARAM *callee (In/Out)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes

**_TA_ID gtrid (In)**
globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird

**CS_SERVICE_NAME fct (In)**
logischer Name des zu adressierenden Dienstes

**CS_SERVICE_NAME callback_fct (In)**
logischer Name der Funktion, an die der aufgerufene Dienst die asynchrone Antwort senden soll

**CS_RM_CLASS_NAME rm_class (In)**
logischer Name der RM-Klasse des zu adressierenden Dienstes

**CS_MSG_OPTION local (In)**
Option des Übertragungsmodus „Lokalität“

**CS_MSG_TYPE data_type (In)**
logischer Name für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN msg_len (In)**
Länge der eigentlichen Nachricht in Bytes

**char *msg_buf (In)**
Nachrichtenpuffer als Zeiger auf die eigentliche Nachricht(Bytestring)

### Beschreibung

Die Funktion *RM_call_async* überträgt die im Puffer *msg_buf* hinterlegte Nachricht an den Empfänger <*fct*,*rm_class*>.
Dabei überprüft sie die Berechtigung der aufrufenden Komponente, das APRICOTS-Kommunikationssystem zu benutzen.
Anschließend wird über die Parameter *fct* und *rm_class* sowie mithilfe der Option *local* der Adressat festgelegt und in die Struktur *callee* eingetragen.
Die im Nachrichtenpuffer *msg_buf* hinterlegte Nachricht des Datentyps *data_type* mit der Länge *msg_len* wird mitsamt den Adreßangaben und den Optionen in das APRICOTS-Übertragungsformat gebracht und schließlich mit einem Basiskommunikationsmechanismus an den Empfänger bzw. bei Übersendung zwischen Rechnern an den Briefträger auf dem Zielrechner übergeben.
Der aufgerufenene Dienst schickt nach Bearbeitung des Auftrags die Antwort an den spezifizierten Callback-Dienst innerhalb der Komponente des Auftraggebers.
Die aufrufende Komponente wartet demnach nicht die Bearbeitung des Auf-



Tabelle A.1: Belegungen des Parameters *local*

| **Belegung des Parameters** | **Wirkung** |
|---|---|
| local = CS_MSG_LOCAL | Bei der Bestimmung des Empfängerdienstes wird eine möglichst lokale Komponente ausgewählt (prozeßlokal bzw. rechnerlokal). |
| local = CS_MSG_REMOTE | Bei der Bestimmung des Empfängerdienstes darf keine prozeßlokale Komponente ausgewählt werden. |

trags ab, sondern kann anderweitig weiterarbeiten.

### Anwendung

Der Parameter *caller* muß aus der eigenen Parameterliste des aufrufenden APRICOTS-Dienstes übernommen werden, um eine korrekte Zusendung der Antwort zu ermöglichen.
Wird *callee = 0* übergeben, so erfolgt ein freies Scheduling. Wird in *callee* ein Zeiger auf eine bereits initialisierte Struktur übergeben, z. B. den Adreßkontrollblock eines früher aufgerufenen Resource Managers, so wird die dort spezifizierte Komponente als Empfängerkomponente übernommen.
Für die korrekte Abwicklung eines Auftrags muß der Parameter *fct* belegt sein, um den Empfängerdienst bestimmen zu können.
Wird *rm_class = 0* übergeben, so wird das Scheduling ohne Berücksichtigung der Resource Manager Klasse nur gemäß des Dienstnamen durchgeführt.
Der Parameter *callback_fct* bezeichnet den Dienst, an den der Callee die Antwort zurücksenden muß. Bei der Antwortnachricht wird dabei diejenige Instanz des Dienstes ausgewählt, die in derselben Komponente lokalisiert ist wie der Dienst, der *RM_call_async* aufgerufen hat. Ist *callback_fct* spezifiziert, wird diese Adresse übernommen, andernfalls geht die Antwortnachricht an den aufrufenden Dienst zurück.
Der Nachrichtenhenkel *msg_handle* dient der Zuordnung von Auftrag und Antwort. Ein Auftrag muß einen Nachrichtenhenkel, der auf NIL zeigt, an den *RM_call_async* übergeben. Ein Dienst, der eine Antwort auf einen erfolgten Auftrag sendet, benutzt den Nachrichtenhenkel aus der eigenen Parameterliste.
Der Parameter *gtrid* kann, muß aber nicht belegt werden. Bei einem Wert ungleich Null wird ein Logging gemäß der APRICOTS-Spezifikation durchgeführt.[2]
Die Belegung des Parameters *local* ist optional. Die möglichen Werte sind in Tabelle A.1 aufgezählt. Die Voreinstellung ist *CS_MSG_LOCAL*.

[2]Ein Logging ist noch nicht implementiert. Die Schnittstelle ist noch nicht spezifiziert.

### Returncodes

**CS_RC_SUCCESS** Die Übernahme durch das Kommunikationssystem war erfolgreich.

**CS_RC_AUTHORIZATION_DENIED** Die aufrufende Komponente besitzt nicht die Berechtigung, das Kommunikationssystem zu benützen (Noch nicht implementiert).

**CS_RC_SERVICE_UNKNOWN** Der angeforderte Dienst wird nicht von APRICOTS angeboten.

**CS_RC_RM_CLASS_UNKNOWN** Die angeforderte Resource Manager Klasse wird nicht von APRICOTS angeboten.

**CS_RC_DATATYPE_UNDEFINED** Der angegebene Datentyp wird nicht von APRICOTS unterstützt. (Noch nicht implementiert).

### Wird aufgerufen von:

Diensten der APRICOTS-Komponenten und der Step Computation Server
RM_call_return

### Ruft auf:

cs_call_authenticate
cs_decode_tm_unit
cs_encode_tm_unit
cs_schedule
cs_transmit_message

### Verweise

main (CS), RM_call_return (CS), RM_call_sync (CS), <service_name> (APRICOTS), cs_call_authenticate (CS), cs_decode_tm_unit (CS), cs_encode_tm_unit (CS), cs_schedule (CS), cs_transmit_message (CS)

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c



## A.3 RM_call_return (CS)

### Name

**RM_call_return** Beendigung eines APRICOTS-Dienstes und Rücksendung der Antwort

### Aufruf

**RM_call_return (&msg_handle, &data_type, &msg_len, &msg_buf)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel für den Auftrag

**CS_MSG_TYPE data_type (In)**
logischer Name für den Datentyp der Antwort

**CS_MSG_LEN msg_len (In)**
Länge der Antwort in Bytes

**char *msg_buf (In)**
Antwortpuffer als Zeiger auf die eigentliche Antwort (Bytestring)

### Beschreibung

*RM_call_return* ist ein Makro, der entsprechend des in *msg_handle* festgehaltenen Nachrichtenstatus die Rückkehr von einem Dienstaufruf durchführt und den Dienst beendet.

### Anwendung

*msg_handle* muß derselbe Nachrichtenhenkel sein, mit dem der Dienst, der über *RM_call_return* seinen Kontrollfluß beendet, aktiviert worden ist.
Ist der aufrufende Dienst synchron aktiviert worden, so führt *RM_call_return* ein *return()* durch.
Ist der aufrufende Dienst asynchron aktiviert worden, so führt *RM_call_return*, falls der aufrufende Dienst Auftragnehmer ist, einen Resource Manager Call an die in *msg_handle* spezifizierte Callback-Adresse durch. Anschließend wird ein *return()* aufgerufen.

### Returncodes

**Kehrt nicht mehr zurück !!!**

### Wird aufgerufen von:

Von beliebigen Diensten des APRICOTS-Systems

### Ruft auf:

RM_call_async
return

### Verweise

RM_call_async (CS), RM_call_sync (CS), <service_name> (APRICOTS)

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.h

## A.4 RM_call_sync (CS)

### Name

**RM_call_sync** Senden einer synchronen Nachricht an einen spezifizierten Dienst innerhalb des APRICOTS-Systems

### Aufruf

**CS_RETURNCODE**
**RM_call_sync (&msg_handle, &caller, &callee, gtrid, fct, rm_class, local, &data_type, &msg_len, &msg_buf)**

**CS_MESSAGE_HANDLE *msg_handle (Out)**
Zeiger auf den eindeutigen Henkel für die Nachricht

**CS_ADDRESS_PARAM caller (In)**
Adreßkontrollblock des Senders

**CS_ADDRESS_PARAM *callee (In/Out)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes

**_TA_ID gtrid (In)**
globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird

**CS_SERVICE_NAME fct (In)**
logischer Name des zu adressierenden Dienstes

**CS_RM_CLASS_NAME rm_class (In)**
logischer Name der RM-Klasse des zu adressierenden Dienstes

**CS_MSG_OPTION local (In)**
Option des Übertragungsmodus „Lokalität"

**CS_MSG_TYPE data_type (In/Out)**
logischer Name für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN msg_len (In/Out)**
Länge der eigentlichen Nachricht in Bytes

**char *msg_buf (In/Out)**
Nachrichtenpuffer als Zeiger auf die eigentliche Nachricht(Bytestring)

### Beschreibung

Die Funktion *RM_call_sync* überträgt die im Puffer *msg_buf* hinterlegte Nachricht an den Empfänger *<fct,rm_class>*.
Dabei überprüft sie die Berechtigung der aufrufenden Komponente, das APRICOTS-Kommunikationssystem zu benutzen.
Anschließend wird über die Parameter *fct* und *rm_class* sowie mithilfe der Option *local* der Adressat festgelegt und in der Struktur *callee* eingetragen.
Die im Nachrichtenpuffer *msg_buf* hinterlegte Nachricht des Datentyps *data_type* mit der Länge *msg_len* wird mitsamt den Adreßangaben und den Optionen in das APRICOTS-Übertragungsformat gebracht und schließlich mit einem Basiskommunikationsmechanismus an den Empfänger bzw. bei Übersendung zwischen Rechnern an den Briefträger auf dem Zielrechner übergeben.
Die Antwort, die in den Parametern *data_type*, *msg_len* und *msg_buf* zurückgesendet wird, wird abgewartet und an den Aufrufer übergeben.

### Anwendung

Der Parameter *caller* muß aus der eigenen Parameterliste des aufrufenden APRICOTS-Dienstes übernommen werden, um eine korrekte Zusendung der Antwort zu ermöglichen.



Wird *callee = 0* übergeben, so erfolgt ein freies Scheduling. Wird in *callee* ein Zeiger auf eine bereits initialisierte Struktur übergeben, z. B. den Adreßkontrollblock eines früher aufgerufenen Resource Managers, so wird die dort spezifizierte Komponente als Empfängerkomponente übernommen.
Für die korrekte Abwicklung eines Auftrags muß der Parameter *fct* belegt sein, um den Empfängerdienst bestimmen zu können.
Wird *rm_class = 0* übergeben, so wird das Scheduling ohne Berücksichtigung der Resource Manager Klasse nur gemäß des Dienstnamen durchgeführt.
Der Nachrichtenhenkel *msg_handle* dient der Zuordnung von Auftrag und Antwort. Ein Auftrag muß einen Nachrichtenhenkel, der auf NIL zeigt, an den *RM_call_async* übergeben.
Der Parameter *gtrid* kann, muß aber nicht belegt werden. Bei einem Wert ungleich Null wird ein Logging gemäß der APRICOTS-Spezifikation durchgeführt.[3]
Die Belegung des Parameters *local* ist optional. Die möglichen Werte sind in Tabelle A.1 aufgezählt. Die Voreinstellung ist *CS_MSG_LOCAL*.
Die Parameter *data_type*, *msg_len* und *msg_buf* dienen zur Aufnahme der Antwort und müssen entsprechend belegt werden. *msg_buf* wird bei der Rückgabe an den Auftraggeber neu allokiert.

### Returncodes

**CS_RC_SUCCESS** Die Übernahme durch das Kommunikationssystem war erfolgreich. Die Antwort ist schon bei der Rückkehr vom Resource Manager Call im Puffer *msg_buf* verfügbar.

**CS_RC_AUTHORIZATION_DENIED** Die aufrufende Komponente besitzt nicht die Berechtigung, das Kommunikationssystem zu benützen (Noch nicht implementiert).

**CS_RC_SERVICE_UNKNOWN** Der angeforderte Dienst wird nicht von APRICOTS angeboten.

**CS_RC_RM_CLASS_UNKNOWN** Die angeforderte Resource Manager Klasse wird nicht von APRICOTS angeboten.

**CS_RC_DATATYPE_UNDEFINED** Der angegebene Datentyp wird nicht von APRICOTS unterstützt. (Noch nicht implementiert).

**CS_RC_REQUEST_REJECTED** Die Bearbeitung ist vom Empfänger verweigert worden.

### Wird aufgerufen von:

Diensten der APRICOTS-Komponenten und der Step Computation Server

### Ruft auf:

[3] Ein Logging ist noch nicht implementiert. Die Schnittstelle ist noch nicht spezifiziert.

cs_activate_service
cs_call_authenticate
cs_decode_tm_unit
cs_encode_tm_unit
cs_schedule
cs_transmit_message

### Verweise

main (CS), RM_call_async (CS), RM_call_return (CS), <service_name> (APRICOTS), cs_activate_service (CS), cs_call_authenticate (CS), cs_decode_tm_unit (CS), cs_encode_tm_unit (CS), cs_schedule (CS), cs_transmit_message (CS)

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c



## A.5 <service_name> (APRICOTS)

### Name

**<service_name>** APRICOTS-Dienst, der einen Resource Manager Call empfangen kann

### Aufruf

**CS_RETURNCODE**
**<service_name> (msg_handle, callee, gtrid, data_type, msg_len, msg_buf)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel der Nachricht

**CS_ADDRESS_PARAM *callee (In)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes

**_TA_ID *gtrid (In)**
Zeiger auf die globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird

**CS_MSG_TYPE *data_type (In/Out)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN *msg_len (In/Out)**
Zeiger auf die Länge der eigentlichen Nachricht in Bytes

**char **msg_buf (In/Out)**
Zeiger auf den Nachrichtenpuffer

### Beschreibung

Der APRICOTS-Dienst <service_name> erhält von der in der Struktur **caller* spezifizierten Komponente die mit dem eindeutigen Nachrichtenhenkel gekennzeichnete Nachricht in dem durch *msg_buf* adressierten Puffer.

### Anwendung

*callee* enthält die Adreßangaben des aufgerufenen Dienstes <service_name> und muß bei einem Resource Manager Call von <service_name> an der Parameterposition des Aufrufers, des Callers, übergeben werden.

Die durch *data_type*, *msg_len* und *msg_buf* spezifizierte Nachricht ist durch eine entsprechend dem Datentyp durchgeführte *cast*- Operation zu übernommen. *msg_len* verweist auf die Länge des Puffers. Es darf keine längere Datenstruktur in *msg_buf* als die ursprünglich übersandte Nachricht als Antwort zurückgegeben werden.

*msg_buf* kann bei der Rückgabe der Antwort auf einen anderen durch Allokation geschaffenen Puffer zeigen, sodaß auf bequeme Weise verschiedene Datentypen für Auftrag und Antwort benutzt werden können. Das sollte jedoch in *data_type* vermerkt werden. Außerdem darf natürlich nicht die maximale Länge des allokierten Puffers überschritten werden.

Ruft <service_name> selbst einen anderen Dienst auf, um einen Auftrag zu vergeben, so sind die Parameter wie in Tabelle A.2 aufgeführt zu belegen.

Die Beendigung eines APRICOTS-Dienstes erfolgt durch den Aufruf von *RM_call_return*. Die Belegung der Parameter erfolgt wie in Tabelle A.2 aufgeführt.

### Returncodes

CS_RC_SUCCESS Der Dienst <service> wurde erfolgreich ausgeführt.

Tabelle A.2: Parameterbelegung eines RM_call_return und eines RM_call

| **Parameter** | **Belegung beim Aufruf von RM_call_return** | **Belegung bei einem neuen Auftrag** |
|---|---|---|
| msg_handle | Nachrichtenhenkel aus der eigenen Parameterliste | Zeiger auf NIL |
| caller | — | callee aus der eigenen Parameterliste |
| callee | — | eine nichtinitialisierte Struktur bei freiem Scheduling bzw. eine initialisierte Struktur, falls Kontext erhalten bleiben soll |
| gtrid | — | die Variable, die die gültige TA-Id enthält |
| local | — | beliebig |
| msg_len, data_type | es muß derselbe Zeiger wie in der Parameterliste des Dienstes verwendet werden | es muß ein vom Zeiger in der Parameterliste verschiedener Zeiger verwendet werden |

**Wird aufgerufen von:**
cs_activate_service

**Ruft auf:**
RM_call_async
RM_call_return
RM_call_sync
komponentenspezifische Funktionen

**Verweise**
RM_call_async (CS), RM_call_return (CS), RM_call_sync (CS)

**Dateien**
Header: /az3/laube/apricot/source/cs.h

Source: in den jeweiligen Resource Manager Quelldateien

# Anhang B

# Schnittstellen: Schichten des CS

## B.1 cs_activate_service (CS)

### Name

**cs_activate_service** Aktivierung eines APRICOTS-Dienstes als Empfänger einer Nachricht

### Aufruf

**CS_RETURNCODE**
**cs_activate_service (msg_handle, callee, gtrid, data_type, msg_len, msg_buf)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel der Nachricht

**CS_ADDRESS_PARAM *callee (In)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes

**_TA_ID *gtrid (In)**
Zeiger auf die globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird

**CS_MSG_TYPE *data_type (In/Out)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN *msg_len (In/Out)**
Zeiger auf die Länge in Bytes der eigentlichen Nachricht

**char **msg_buf (In/Out)**
Zeiger auf die eigentliche Nachricht (als Bytestring)

### Beschreibung

Die Funktion *cs_activate_service* aktiviert den durch den Parameter *callee* spezifizierten Dienst. Dem Empfängerdienst werden alle Parameter von *cs_activate_service* übergeben.

### Anwendung

*cs_activate_service* wird bei prozeßlokalen, synchronen Nachrichten von *RM_call_sync* aufgerufen, um direkt den Empfängerdienst zu aktivieren. Im allgemeinen erfolgt der Aufruf jedoch von der Nachrichtenverteilerschleife in *main* aus, die mit diesem Aufruf die Nachricht aus den Eingabewarteschlangen an den Empfängerdienst weiterleitet.
Nach Beendigung der Verarbeitung durch den Dienst erhält *cs_activate_service* die Kontrolle durch das Makro *RM_call_return* zurück.

### Returncodes

**CS_RC_SUCCESS** Die Funktion ist erfolgreich ausgeführt worden.

**CS_RC_SERVICE_ADDRESS_UNDEFINED** Die im Parameter *callee* spezifizierte Einsprungadresse ist nicht korrekt.

### Wird aufgerufen von:

main
RM_call_sync

### Ruft auf:

cs_lookup_function_table
den in der Struktur *callee* durch die Felder Servicename, Resource Manager Klasse und Komponentenname spezifizierten Dienst

### Verweise



<service_name> (APRICOTS), RM_call_sync (CS)

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## B.2 cs_call_authenticate (CS)

### Name

**cs_call_authenticate** Überprüfung der Berechtigung eines Senders, eine Nachricht im APRICOTS-System an den Empfänger zu versenden

### Aufruf

**CS_RETURNCODE**
**cs_call_authenticate (caller)**
**CS_ADDRESS_PARAM *caller (In)**
Zeiger auf den Adreßkontrollblock des Senders

### Beschreibung

Die Funktion dient der Überprüfung der Berechtigung des Senders *caller*, das Kommunikationssystem zu benutzen.

### Anwendung

Es wird vorläufig nur die Schnittstelle implementiert. *cs_call_authenticate* liefert deshalb stets den Wert CS_RC_SUCCESS zurück.

### Returncodes

**CS_RC_SUCCESS** *caller* hat die Berechtigung, das APRICOTS- Kommunikationssystem zu benutzen.

**CS_RC_AUTHORIZATION_DENIED** *callee* hat keine Berechtigung, das APRICOTS-Kommunikationssystem zu benutzen

### Wird aufgerufen von:

main
RM_call_sync
RM_call_async

### Ruft auf:

—

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c



## B.3 cs_decode_tm_unit (CS)

### Name

**cs_decode_tm_unit** Übertragung einer übertragenen Nachricht aus dem APRICOTS- Übertragungsformat in die einzelnen Parameter

### Aufruf

**CS_RETURNCODE**
**cs_decode_tm_unit (message, &msg_handle, &caller, &callee, &gtrid, &sync, &data_type, &msg_len, &msg_buf)**
**CS_TRANSMISSION_UNIT *message (In)**
Zeiger auf die Nachricht im APRICOTS- Übertragungsformat
**CS_MESSAGE_HANDLE *msg_handle (Out)**
Zeiger auf den eindeutigen Henkel der Nachricht
**CS_ADDRESS_PARAM *caller (Out)**
Zeiger auf den Adreßkontrollblock des Senders
**CS_ADDRESS_PARAM *callee (Out)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes
**_TA_ID *gtrid (Out)**
Zeiger auf die globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird
**CS_MSG_OPTION *sync (Out)**
Zeiger auf die Option der Übertragungsart 'Synchronität'
**CS_MSG_TYPE *data_type (Out)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht
**CS_MSG_LEN *msg_len (Out)**
Zeiger auf die Länge in Bytes der eigentlichen Nachricht
**char *msg_buf (Out)**
eigentliche Nachricht (als Bytestring)

### Beschreibung

*cs_decode_tm_unit* zerlegt die Nachricht *message*, die im APRICOTS-Übertragungsformat vorliegt, in die Parameter, die zur Weiterleitung an den Empfängerdienst benötigt werden.
Die Ausgabeparameter werden durch ein *malloc()* erzeugt, weswegen die Adressen der Zeiger übergeben werden.

### Returncodes

**CS_RC_SUCCESS** Die Dekodierung der APRICOTS-Übertragungseinheit war erfolgreich.

**CS_RC_DATATYPE_UNDEFINED** Die Dekodierung ist an einem undefinierten Datentyp gescheitert. (nicht implementiert)

### Wird aufgerufen von:

main
RM_Call_sync
RM_call_async

### Ruft auf:

malloc
cs_copy_msg_handle
cs_copy_address_param

### Verweise

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c



## B.4 cs_encode_tm_unit (CS)

### Name

**cs_encode_tm_unit** Übertragung der Nachricht und ihrer Spezifikationen in das APRICOTS-Übertragungsformat

### Aufruf

**CS_RETURNCODE**
**cs_encode_tm_unit (msg_handle, caller, callee, gtrid, sync, data_type, msg_len, msg_buf, &message)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel der Nachricht

**CS_ADDRESS_PARAM *caller (In)**
Zeiger auf den Adreßkontrollblock des Senders

**CS_ADDRESS_PARAM *callee (In)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes

**_TA_ID *gtrid (In)**
Zeiger auf die globale Id der Transaktion, unter deren Schutz die Nachricht versendet wird

**CS_MSG_OPTION *sync (In)**
Zeiger auf die Option der Übertragungsart 'Synchronität'

**CS_MSG_TYPE *data_type (In)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN *msg_len (In)**
Zeiger auf die Länge in Bytes der eigentlichen Nachricht

**char **msg_buf (In)**
Zeiger auf die eigentliche Nachricht (als Bytestring)

**CS_TRANSMISSION_UNIT *message (Out)**
Zeiger auf die Nachricht im APRICOTS-Übertragungsformat

### Beschreibung

Die Funktion *cs_encode_tm_unit* überträgt die Eingabeparameter in die neu zu allokierende Struktur **message*.

### Returncodes

**CS_RC_SUCCESS** Die Übertragung in das APRICOTS-Übertragungsformat war erfolgreich.

**CS_RC_DATATYPE_UNDEFINED** Die Enkodierung ist an einem undefinierten Datentyp gescheitert. (nicht implementiert)

### Wird aufgerufen von:

main
RM_call_sync
RM_call_async

### Ruft auf:

malloc
cs_copy_msg_handle
cs_copy_address_param

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## B.5 cs_init_communication_system (CS)

### Name

**cs_init_communication_system** Initialisierung der prozeßlokalen Datenverwaltung für das Kommunikationssystem

### Aufruf

**CS_RETURNCODE**
**cs_init_communication_system (argc, argv)**
**int argc (In)**
Argumentzähler
**char *argv[ ] (In)**
Argumentliste

### Beschreibung

*cs_init_communication_system* initialisiert zunächst den prozeßlokalen Cache und die prozeßlokale Nachrichtenwarteschlange und belegt dann prozeßglobale Daten.
Im nächsten Schritt wird überprüft, ob die gestartete Komponente ein ConTract Processing Monitor ist.
Wenn dies der Fall ist, so generiert *cs_init_communication_system* den Komponentennamen, initialisiert die prozeßlokale Einsprungadressentabelle, installiert seine Mailbox und bindet sie in den eigenen Adreßraum ein. Danach wird die Komponentendefinitionstabelle eingelesen und an den Name Service zur Eintragung weitergegeben.
Alle übrigen APRICOTS-Komponenten initialisieren allein die prozeßlokale Einsprungadressentabelle und binden die vom ConTract Processing Monitor generierten Mailboxen, deren Nummer in den Prozeßparametern übergeben werden, in ihren Adreßraum ein.

### Anwendung

Für einen ConTract Processing Monitor enthält nur *argv[0]* ein Argument, nämlich den Programmnamen. Ansonsten sind die Parameter belegt, wie es bei *main() (CS)* beschrieben ist.

### Returncodes

**CS_RC_SUCCESS** Der Prozeß ist beim ConTract Processing Monitor angemeldet und die Datenstrukturen sind initialisiert.

**CS_RC_NO_QUOTA** Das Betriebssystem hat die Zuteilung von Resourcen verweigert.

### Wird aufgerufen von:

main

### Ruft auf:

cs_connect_mailbox
cs_destroy_mailbox
cs_init_function_table



cs_init_local_cache
cs_install_mailbox
cs_name_service_connect
cs_name_service_destroy
cs_name_service_insert_comp
cs_name_service_install
gethostname
malloc

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## B.6 cs_lookup_message (CS)

### Name

**cs_lookup_message** Lesen einer Nachricht aus den Eingabepuffern einer Komponente

### Aufruf

**CS_RETURNCODE**
**cs_lookup_message (&message)**
**CS_TRANSMISSION_UNIT *message (Out)**
Zeiger auf die Nachricht im APRICOTS-Übertragungsformat

### Beschreibung

*cs_lookup_message* liest zunächst von der prozeßlokalen Nachrichtenwarteschlange. Liegt dort keine Nachricht an, so wird versucht, eine Nachricht von der Mailbox zu lesen.

### Returncodes

**CS_RC_SUCCESS** Eine Nachricht aus dem Eingabepuffer wird mit dem Parameter *message* zurückgegeben.

**CS_RC_NO_MESSAGE** Es liegt keine Nachricht in den Eingabepuffern.

### Wird aufgerufen von:

main

### Ruft auf:

cs_get_local_queue
cs_read_message_from_mailbox

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c



## B.7 cs_return_message (CS)

### Name

**cs_return_message** Zurücksenden einer nicht prozeßlokalen Antwort auf eine synchrone Nachricht

### Aufruf

**CS_RETURNCODE**
**cs_return_message (message)**
**CS_TRANSMISSION_UNIT *message (In)**
Zeiger auf die Antwort im APRICOTS-Übertragungsformat

### Beschreibung

Wenn ein Dienst synchron von einem Sender eines anderen Prozesses aktiviert wurde, so wird die Antwort vom Nachrichtenverteiler des Empfängers direkt an den Sender zurückgeschickt.

### Anwendung

Diese Funktion wird nur bei nicht prozeßlokalen synchronen Nachrichten von *main* aufgerufen.

### Returncodes

**CS_RC_SUCCESS** Die Antwort ist erfolgreich an den Auftraggeber gesendet worden.

**CS_RC_RETURN_ADDRESS_UNDEFINED** Die in Adreßkontrollblock *caller* angegebene Adresse ist ungültig.

### Wird aufgerufen von:

main

### Ruft auf:

cs_write_message_into_mailbox

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## B.8 cs_schedule (CS)

### Name

**cs_schedule** Bestimmung der Adressierungsangaben eines Empfängerdienstes

### Aufruf

**CS_RETURNCODE**
**cs_schedule (&msg_handle, caller, &callee, fct, callback, rm_class, sync, local)**

**CS_MESSAGE_HANDLE *msg_handle (In/Out)**
Zeiger auf den eindeutigen Henkel der Nachricht
**CS_ADDRESS_PARAM caller (In)**
Adreßkontrollblock des Senders
**CS_ADDRESS_PARAM *callee (In/Out)**
Zeiger auf den Adreßkontrollblock des Empfängerdienstes
**CS_SERVICE_NAME fct (In)**
logischer Name des zu adressierenden Dienstes
**CS_SERVICE_NAME callback (In)**
logischer Name des Callback-Dienstes
**CS_RM_CLASS_NAME rm_class (In)**
logischer Name der Resource Manager Klasse des zu adressierenden Dienstes
**CS_MSG_OPTION sync (In)**
Option der Übertragungsart 'Synchronität'
**CS_MSG_OPTION local (In)**
Option des Übertragungsmodus 'Lokalität'

### Beschreibung

*cs_schedule* geht bei der Adreßbestimmung in mehreren Schritten vor.
Zunächst wird überprüft, ob die Nachricht ein Auftrag oder eine Antwort auf einen Auftrag ist. Handelt es sich um eine Antwort, wird die *msg_handle* aktualisiert und die Adreßbestimmung ist beendet. Andernfalls wird eine *msg_handle* allokiert.
Im nächsten Schritt wird überprüft, ob der Adreßkontrollblock, auf den *callee* verweist, bereits belegt ist. Ist dies der Fall, wird außer Aktualisierung des Dienst- und Callback-Namens keine weitere Adreßbestimmung durchgeführt. Andernfalls wird die Struktur *callee* neu allokiert.
Im nächsten Schritt beginnt das eigentliche Scheduling. Entsprechend der Belegung von *local* werden die Einsprungadressentabelle und der prozeßlokale Cache durchsucht.
Ist die Suche ohne Erfolg, wird eine Suche im Name Service vorgenommen und das Ergebnis bei Erfolg in den prozeßlokalen Cache eingetragen. Danach wird dieser noch einmal durchsucht.
Suchalgorithmus: „Nimm die Zielkomponente, die am nächsten liegt und zu der schon eine Verbindung aufgebaut worden ist."

### Anwendung

Ist *msg_handle* = 0, so handelt es sich bei der Nachricht um einen Auftrag, andernfalls um die Antwort auf einen Auftrag.
Ist *callee* = 0, so wird ein freies Scheduling durchgeführt, andernfalls werden die in **callee* spezifizierten Angaben übernommen.



Ist *local* = *CS_MSG_ANYWAY*, so wird in allen Adreßtabellen gesucht. Bei *local* = *CS_MSG_LOCAL* wird nur in der Einsprungadressentabelle, bei *local* = *CS_MSG_REMOTE* in allen Tabellen außer dieser gesucht.

### Returncodes

**CS_RC_SUCCESS** Es wurde ein Dienst gefunden, der den spezifizierten Angaben genügt.

**CS_RC_SERVICE_INVALID** Es wurde kein Dienst gefunden, der den spezifizierten Angaben genügt.

### Wird aufgerufen von:

RM_call_sync
RM_call_async

### Ruft auf:

cs_init_msg_handle
cs_insert_local_cache
cs_lookup_function_table
cs_lookup_local_cache
cs_name_service_lookup

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

## B.9 cs_transmit_message (CS)

### Name

**cs_transmit_message** Übertragung einer Nachricht vom Sender zum Empfänger

### Aufruf

**CS_RETURNCODE**
**cs_transmit_message (&message)**
**CS_TRANSMISSION_UNIT *message (In/out)**
Zeiger auf die Nachricht im APRICOTS- Übertragungsformat

### Beschreibung

Die Nachricht wird im asynchronen, prozeßlokalen Fall [1] in die prozeßlokale Nachrichtenwarteschlange eingetragen.
Im nichtprozeßlokalen Fall wird die Nachricht in die Mailbox des Empfängers eingetragen. Ist die Nachricht dann synchron, so wird die Antwort in *message* zurückgesendet.

### Returncodes

**CS_RC_SUCCESS** Die Nachricht ist – im asynchronen Fall – erfolgreich bei der Zielkomponente eingetragen worden. Im synchronen Fall liegt zusätzlich die Antwort schon vor.

### Wird aufgerufen von:

RM_call_sync
RM_call_async

### Ruft auf:

cs_connect_mailbox
cs_put_local_queue
cs_write_message_into_mailbox

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cs.c

[1] Der synchrone, prozeßlokale Fall wird direkt von RM_call_sync bearbeitet.

# Anhang C

# Schnittstellen: Dienste des CPM

Die Dienste, die der ConTract Processing Monitor anbietet, haben dieselbe Schnittstelle wie alle anderen Dienste in APRICOTS, das beim Eintrag <service_name> beschrieben worden ist.

## C.1 create_component (APRICOTS)

### Name

**create_component** Erzeugung eines Resource Manager unter Kontrolle des ConTract Processing Monitor

### Aufruf

**CS_RETURNCODE**
**create_component (msg_handle, cpm_address, gtrid, data_type, msg_len, msg)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel der Nachricht
**CS_ADDRESS_PARAM *cpm_address (In)**
Zeiger auf den Adreßkontrollblock des empfangenden ConTract Processing Monitor
**_TA_ID *gtrid (In)**
Zeiger auf die globale Id der Transkation, unter deren Schutz die Nachricht versendet wird
**CS_MSG_TYPE *data_type (In/Out)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht
**CS_MSG_LEN *msg_len (In/Out)**
Zeiger auf die Länge der eigentlichen Nachricht in Bytes
**char **msg (In/Out)**
Zeiger auf den String mit dem Namen der zu startenden Komponente

### Beschreibung

Die Funktion *create_component* erzeugt die im Nachrichtenpuffer *msg* durch *program_name* spezifizierte Komponente. Dabei wird zunächst eine Mailbox für die zu erzeugende Komponente eingerichtet. Anschließend wird der Prozeß für diese Komponente durch ein *fork()* erzeugt und es werden die von *program_name* angebotenen Dienste in den Name Service eingetragen. Danach wird mit einem *execv()* die Komponente gestartet.
Bei Erfolg gibt die Funktion *CS_RC_SUCCESS* zurück und *msg* zeigt auf den logischen Prozeßnamen der gestarteten Komponente.
Treten Fehler auf, so zeigt *msg* auf einen leeren String "".

### Anwendung

Der Parameter *msg* enthält den Zeiger auf den Namen eines ausführbaren Programmes. Um die Komponente installieren zu können, benötigt der ConTract Processing Monitor zusätzlich eine Definitionsdatei mit dem Namen *<program_name>.cpm_def*. Diese Datei wird von *make_apricot_process* beim Compilieren von *program_name* erzeugt und enthält eine Liste der angebotenen Dienste, die vom ConTract Processing Monitor in die Name Service Tabelle übernommen wird.
Bei der Rückgabe enthält *msg* einen Zeiger auf den logischen Komponentennamen, mit dem die Komponente vom ConTract Processing Monitor versehen worden ist.

### Returncodes

**CS_RC_SUCCESS** Die Komponente ist erfolgreich gestartet worden.

**CS_RC_NO_QUOTA** Die Komponente istaufgrund von Betriebsmittelverweigerungen nicht gestartet worden.



**CS_RC_NO_DEF_FILE** Die Komponente ist wegen einer fehlenden Definitionsdatei nicht gestartet worden.

### Wird aufgerufen von:

main

### Ruft auf:

cs_connect_mailbox
cs_delete_local_cache
cs_destroy_mailbox
cs_disconnect_mailbox
cs_insert_local_cache
cs_install_mailbox
cs_name_service_delete_comp
cs_name_service_insert_comp
execv
fork
malloc

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cpm.h

*typedef struct {*
*long stop_mode ;*
*CS_LOGICAL_NAME component ;*
*} STOP_COMPONENT_STRUCT ;*

Tabelle C.1: Datentyp STOP_COMPONENT_STRUCT

## C.2 stop_component (APRICOTS)

### Name

**stop_component** Anhalten eines Resource Manager in APRICOTS

### Aufruf

**CS_RETURNCODE**
**stop_component (msg_handle, cpm_address, gtrid, data_type, msg_len, msg)**

**CS_MESSAGE_HANDLE *msg_handle (In)**
Zeiger auf den eindeutigen Henkel der Nachricht

**CS_ADDRESS_PARAM *cpm_address (In)**
Zeiger auf den Adreßkontrollblock des empfangenden ConTract Processing Monitor

**_TA_ID *gtrid (In)**
Zeiger auf die globale Id der Transkation, unter deren Schutz die Nachricht versendet wird

**CS_MSG_TYPE *data_type (In/Out)**
Zeiger auf den logischen Namen für den Datentyp der eigentlichen Nachricht

**CS_MSG_LEN *msg_len (In/Out)**
Zeiger auf die Länge der eigentlichen Nachricht in Bytes

**char **msg (In/Out)**
Zeiger auf eine Struktur, die Komponentennamen und Stopmodus enthält

### Beschreibung

Der Parameter *msg* weist auf eine Struktur des Typs *STOP_COMPONENT_STRUCT* (vgl. Tab. C.1) mit dem logischen Namen einer Komponente *component* und einem Stopmodus *mode*.
Der durch *component* spezifizierte Prozeß wird gestoppt und bei *mode* = *STOP_HARD* aus dem APRICOTS-System entfernt. Dabei werden die Nachrichten, die in der Mailbox des Prozesses liegen, erhalten und müssen von den jeweiligen Sender durch ein Timeout storniert werden.
Der ConTract Processing Monitor aktualisiert die Name Service Tabelle.

### Anwendung

Das Feld *component* in der Struktur, auf die *msg* verweist, enthält den logischen Prozeßnamen der Komponente, die gestoppt werden soll. Es ist der Name, der durch den vorhergehenden Aufruf von *create_component* zurückgeliefert wurde.

### Returncodes

**CS_RC_SUCCESS** Die spezifizierte Komponente ist angehalten und – bei Modus = *STOP_HARD* – aus dem Prozeßsystem entfernt worden.



**CS_RC_COMPONENT_INVALID** Der Komponentenname ist ungültig.

### Dateien

Header: /az3/laube/apricot/source/cs.h

Source: /az3/laube/apricot/source/cpm.h