Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Klassische ACID[1]–Transaktionen wurden im Umfeld von Datenbanksystemen entwickelt. Sie haben sich bewährt, wenn Transaktionen kleine Einheiten sind, die wenige, einfach strukturierte Datenbankobjekte bearbeiten und deshalb nur eine kurze System–Verweildauer besitzen. Unter diesen Bedingungen stellen Transaktionen ein geeignetes Hilfsmittel dar, um atomare Zustandsübergänge zu gewährleisten.

Probleme ergeben sich allerdings bei sogenannten „Non-Standard“-Anwendungen wie Büroautomatisierung, Fabrikautomatisierung, CAD–Systeme etc. Die Vorgänge dieser Anwendungsgebieten zeichnen sich durch lange Transaktionen (im Vergleich zu oben skizzierten klassischen Transaktionen) aus, verwenden viele Objekte und haben komplexe Kontrollflußbeziehungen wie Migration von einem System auf ein anderes, Ausnahmebehandlung etc. Beispiele lassen sich im Bereich der Fertigungstechnik finden:

- MIGRATION
  Durch einen Ausfall muß ein Fertigungsvorgang von einer Maschine auf eine andere Maschine verlagert werden.

- AUSNAHMEBEHANDLUNG
  Die Verlagerung eines Vorgangs führt zur Unterbrechnung des Fertigungsprozesses. Der Vorgang kann im Sinne einer ACID-Transaktion nicht einfach „zurückgesetzt“ werden. Die Auswirkung des Bohrvorgangs — das gebohrte Loch — kann nicht einfach unwirksam gemacht werden. Zudem ist der Rohling kein Ausschuß, er kann weiterverarbeitet werden.

Das ConTract-Modell[2] [21] [23] [27] versucht eine formale Basis zur Kontrolle langlebiger, komplexer Abläufe zu definieren. ConTracts definieren eine Kontrollstruktur über ACID-Transaktionen und stellen ein Programmiermodell, das persistente Objekte, Konsistenz, Recovery, Synchronisation und Kooperation beinhaltet, zur Verfügung.

[1] Atomicity, Consistency, Isolation and Durability

[2] Controled activities, controled transactions compound

Zur Realisierung einer Umgebung für die Ausführung langlebiger, konsistenter und verteilten Anwendungen (ConTracts) wurde an der Universität Stuttgart das APRICOT[3]–System entwickelt. Es umfaßt verschiedene Komponenten zur Erfüllung der Anforderungen, die die Abwicklung eines ConTracts an eine Betriebsumgebung stellt. Diese sind:

- Verwaltung des Kontrollflusses
- Transaktionsverwaltung
- Synchronisation
- Recovery
- Transaktionsorientierte Kommunikation
- Resourcenverwaltung.

Zu den Aufgaben dieser Diplomarbeit gehört der Entwurf und die Realisierung eines geeigneten Kommunikationsmechanismus. Als Voraussetzung wird eine Beschreibung des Betriebsmodelles des APRICOT–Systems und eine Klassifikation des Kommunikationsbedarfs der einzelnen Komponenten benötigt.

Im den folgenden Abschnitten soll eine nähere Erläuterung des ConTract–Modells und eine Beschreibung des APRICOT–Systems erfolgen. Kapitel 2 spezifiziert das APRICOT–System und seine Komponenten. Mit den Abläufen im System und der Kommunikation der Komponenten untereinander befassen sich Kapitel 3 und 4.

Kapitel 5 definiert die Anforderungen an das Kommunikationsystem und eine Gegenüberstellung der wichtigsten Kommunikationskonzepte.

Im Kaiptel 6 werden das entwickelte Kommunikationskonzept, die G"runde die zu ihm geführt haben und die Realisierung des Communication System im APRICOT System beschrieben.

[3] A Prototyp Implementation of the ConTract System

1.2 ConTracts

Der Abschnitt über ConTracts ist nicht eigentlich Bestandteil der Diplomarbeit. Er versucht die Grundlagen zum Verständnis der Arbeit am APRICOT System zu schaffen. Aus diesem Grund ist dieser Abschnitt eine Zusamenfassung des Manuskriptes *The ConTract Model* von Helmut Wächter und Prof. Dr. Andreas Reuter, IPVR[4], Universität Stuttgart [27].

1.2.1 Problemstellung

Klassische Transaktion waren der erste anwendungsunabhägnige Kontrollmechanismus, der Teilaufgaben im Zusammenhang mit einer Datenbankanwendung unterstützte. Die Eigenschaften einer Transaktion (ACID–Prinzipien: Atomarität, Konsitenz, Isolation und Dauerhaftigkeit) werden in vielen Anwendungen, z.B. Reservierungssysteme, Bank–Buchungssysteme, Buchhaltung etc. erfolgreich eingesetzt. In anderen Bereichen stößt das Transaktionskonzept an seine Grenzen, da in „Non-Standard Applications" einige Voraussetzungen keine Gültigkeit mehr besitzen. Klassische Transaktionen modellieren:

- Kurze Ausführungseinheiten,
- die auf kleinen, einfach strukturierten Datenobjekten
- konkurrierend arbeiten.

Der gravierendste Nachteil im Zusammenhang mit langlebigen Applikationen ist die Annahme, daß ACID-Transaktionen unabhängige, konkurrierende Einheiten sind. Als Konsequenz ergibt sich hieraus, daß semantische Beziehungen zwischen Transaktionen nicht durch das (DB–)System sondern durch die Anwendung zu realisieren sind. (vgl. Transaktionssysteme CICS, UTM, etc. [31]).

Abbildung 1.1 zeigt Transaktionen, die durch semantischen Beziehungen verbunden sind:

> Transaktion T_1 muß ausgeführt werden. Danach können die Transaktionen T_2, T_3 und T_4 parallel gestartet werden. Nach deren erfolgreicher Beendigung kann T_5 abgearbeitet werden. Sollte eine der Transaktionen T_2, T_3 oder T_4 scheitern, müssen die beiden anderen ebenfalls abgebrochen werden. Zudem müssen die Auswirkungen von T_1 rückgängig gemacht werden.

Es ist in herkömmlichen Datenbanksystemen nicht möglich, die oben dargestellten Beziehungen zwischen Transaktionen mit Systemmitteln zu beschreiben. Weder die Beschreibung der Kontollflußbeziehungen noch die Notwendigkeit des Zurücksetzens der ersten Transaktion im Fehlerfall können in DBMS spezifiziert werden.

[4]Institut für Parallele und Verteilte Höchstleistungsrechner

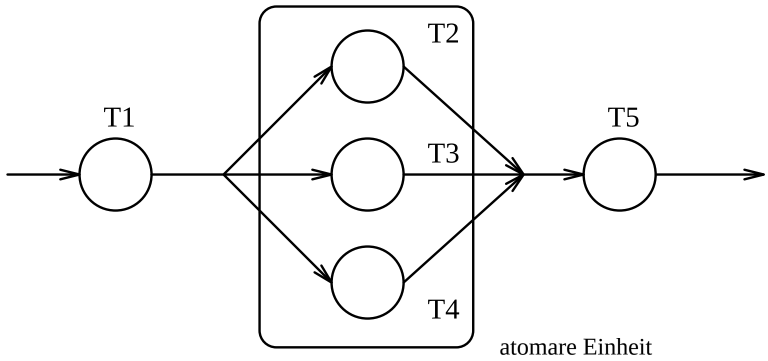

Abbildung 1.1: Beispiel für semantische Beziehungen zwischen Transaktionen

1.2.2 Anforderungen an das Verarbeitungsmodell

Welche Anforderungen ergeben sich an einen Mechanimus zur sicheren und korrekten Ausführung langlebiger, verteilter Anwendungen?

- PROGRAMMIERMODELL
  Anwendungen werden aus existierenden Programmen (Transaktionen) zusammengesetzt. Die Wiederverwendung vorhandener Software muß möglich sein.

- KONTROLLFLUSS
  Die Struktur langlebiger Aktionen muß beschrieben werden. Es müssen Möglichkeiten zur statischen und dynamischen Beschreibung von Kontrollflußbeziehungen vorhanden sein. Ein typischer Fall ist die Unterbrechung eines Vorgangs, um ihn an einen anderen Ort zu migrieren und dort fortzuführen (vgl. auch Beispiel auf Seite 1).

- FEHLERMODELL UND RECOVERY
  Das „alles-oder-nichts“–Prinzip ist unzureichend oder sogar unmöglich. Die Sprache zur Beschreibung des Kontrollflusses muß ein explizit und exakt definiertes Fehlermodell beinhalten. Die drei zentralen Anforderungen dabei sind:

  – Beim Zusammensetzen grösserer Anwendungen aus verschiedenen kleineren Einheiten müssen atomare Gruppen definiert und verwaltet werden (T_2, T_3, T_4 im Beispiel oben).

  – Ein Systemfehler muß nicht den ganzen Vorgang zerstören oder nutzlos machen („gebohrtes Loch“).

  – Im Gegensatz zu kurzen Transaktionen ist ein Ablauf „forward recoverable“, z.B. durch Reinstanzierung und Fortsetzung anhand der Kontrollflußspezifikationen (z.B. „versuche Aktion X n-mal auszuführen“).

- Kontextverwaltung
  Für ein „roll-forward“ muß nicht nur der Zustand der Datenbank, sondern auch der lokale Zustand der Anwendung wiederhergestellt werden.

- Referenzierung der Ausführungsreihenfolge
  Anwendungen mit einer langen Ausführungszeit benötigen manchmal Informationen über ihre Historie und ihren Ausführungspfad. Z.B. kann die Entscheidung über die Fortführung von vorangegangenen Aktionen abhängen[5]. In diesem Fall müssen sowohl die Ausführungsreihenfolge als auch vorausgegangene lokale Zustände referenziert werden.

- Externalisierung von Zwischenergebnissen
  In langlebigen Anwendungen müssen Zwischenergebnisse externalisiert werden bevor der eigentliche Vorgang abgeschlossen ist. Dies impliziert, daß ein Zurücksetzen nicht in einem Schritt möglich ist. Vielmehr müssen Kompensationsaktionen[6] als Teil der Kontrollflußinformation bekanntgegeben werden.

- Modell für konkurrierenden Zugriff und Konsistenz
  Aus denselben Gründen kann die Konsistenz nicht länger an der Serialisierbarkeit von Transaktionen festgemacht werden. Es müssen anwendungsbezogen Regeln zum synchronisierten Zugriff auf gemeinsame Objekte vorhanden sein.

- Konfliktbehandlung
  Im allgemeinen ist es nicht angebracht, eine Aktion warten zu lassen bis eine andere (langlebige) Anwendung beendet ist oder zurückgesetzt wurde. Aus diesem Grund muß in der Kontrollflußbeschreibung die Möglichkeit existieren, eine geeignete Konfliktbehandlung zu spezifizieren.

1.2.3 Das ConTract–Modell

Die obige Liste von Anforderungen führt zu einer Verallgemeinerung des klassiches Transaktionskonzeptes. Im folgenden soll nun der ConTract–Mechanismus vorgstellt werden, der eine Arbeitseinheit beschreibt, welche die angeführten Eigenschaften und Qualitäten besitzt.

Der Begriff *ConTract Manager* beschreibt dabei einen Systemdienst, der die Anforderungen für alle Arten von Anwendungen implementiert.

Bemerkungen: Ziel dieser Einführung ist es, die Grundlagen zum Verständnis der Anforderungen an das APRICOT System zu schaffen. Ausführliche Diskussionen der ConTract–Problematik findet sich in [21], [23] und [27].

[5] Ein gebohrtes Loch muß bei späterer Fortsetzung nicht noch einmal gebohrt werden.

[6] Kompensation ist das logische Zurücksetzen auf den Ausgangspunkt. Die Zubuchung eines Betrages auf ein Bankkonto kann kompensiert werden, indem der Betrag wieder abgebucht wird. In diesem Fall muß weder der Anfangskontostand bekannt sein („Before–After–Image“), noch das Konto in irgendeiner Art gesperrt werden.

1.2.4 ConTract Definition

Was ist ein ConTract*?*

Ein ConTract ist die unverwundbare und konsistente Ausführung einer beliebigen Folge vordefinierter Aktionen (*steps*) anhand einer explizit spezifizierten Kontrollflußbeschreibung (*script*).

Ein ConTract ist ein Programm, daß wie jede Programmierumgebung für parallele Anwendungen, Kontrollflußdefinitionen, persistente lokale Variablen, Zugriff auf gemeinsame Objekte, die durch einen anwendungsbezogenen Synchronisationsmechanismus koordiniert werden, und eine exakt definierte Fehlersemantik besitzt.

1.2.5 Kontrollflußbeschreibung

Als Beispiel–ConTract dient eine kombinierte Flug-, Hotel- und Mietwagenbuchungsanwendung. Sie ist ein typisches Fall für eine länger dauernde Anwendung, die i.A. mehr als eine Session benötigt und deshalb nicht als eine Transaktion abgewickelt werden kann. Abbildung 1.2 zeigt die graphische Repräsentation. Im Anhang A ab Seite 85 ist die textuelle Form zu finden.

script Ein **Script** beschreibt den Kontrollfluß und andere Ausführungsstrategien für langlebige Aktivitäten.

Die Beschreibung des Kontrollflusses kann dabei in jeder beliebigen Notation erfolgen. Die benutzerfreudlichste Alternative ist sicherlich eine graphische Repräsentation (siehe Abbildung 1.2). Da bei der Entwicklung des ConTract–Konzeptes kein besonderer Schwerpunkt auf die Benutzerschnittstelle gelegt wurde, wird eine textuelle, PASCAL-ähnliche Notation verwendet (siehe Beispiel in Anhang A).

Die Beziehungen zwischen Steps können mit herkömmlichen Elementen wie Sequenz, Verzweigung, Schleife und Konstrukturen für Parallelität beschrieben werden.

step Ein **Step** ist die elementare Ausführungseinheit des ConTract–Modells.

Jeder Step stellt eine Basisberechnung einer Anwendung dar, z.B. Buchung eines Fluges, Stornierung einer Reservierung etc. (siehe Abb. 1.3).

1.2.6 ConTract Programmiermodell

Beim ConTract Programmiermodell ist die Kodierung eines Steps getrennt von der Definition des Kontrollflusses einer Anwendung. Die Erstellung des Steps zur Flugreservierung und die Konkatenation der einzelnen Steps im Beispiel (Anhang A ab Seite 85) sind seperate Tätigkeiten, die sogar von verschiedenen Personen ausgeführt werden können.

Hintergrund dieser Trennung ist es, die Programmierumgebung so einfach wie möglich zu halten. Steps können erstellt werden, ohne Gesichtspunkte wie Asynchronität, Parallelität, Kommunikation, Resourcenverteilung (Lokalität), Synchronisation und Recovery im Fehlerfall zu berücksichtigen. Der Programmierer braucht auch nicht zu wissen, an welcher Stelle im Netz sein Step ausgeführt wird oder ob er in Verbindung mit anderen Steps als atomare ACID-Transaktion ausgeführt werden soll (z.B. (S_4, S_5) oder (S_8, S_9, S_{10})).

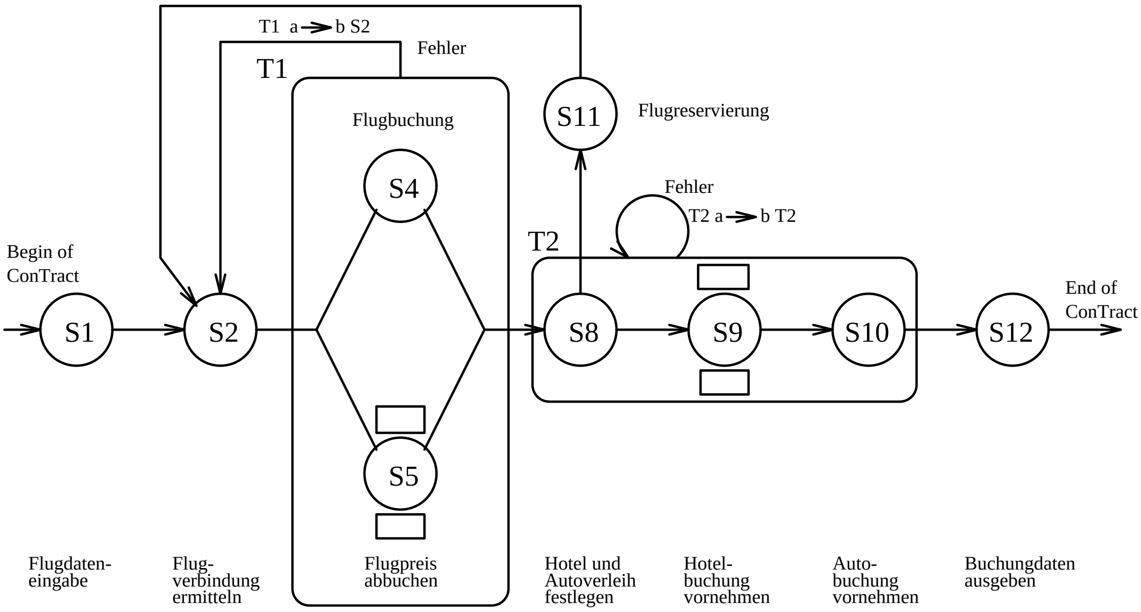

Abbildung 1.2: Beispielskript „Flug- und Hotelbuchung" — graphische Darstellung

1.2.7 Transaktionsmodell

ConTracts besitzen eine verfeinerte Menge von Konstrukten zur Kontrolle von Transaktionen. Diese sollen folgende Anforderungen erfüllen:

1. Flexibler Transaktionsmechanismus zur Strukturierung großer verteilter Anwendungen.

2. Unterstützung des Skriptprogrammierers durch Mechanismen, die einfach zu handhaben und zu verstehen sind.

Ein Step wird als klassische Transaktion realisiert, soweit im `TRANSACTIONS` Abschnitt nichts anderes spezifiziert ist. D.h., ein Step besitzt weiterhin alle ACID–Eigenschaften, die allerdings nur eine lokale Konsistenz der manipulierten Objekte gewährleistet.

Da ein ConTract keine ACID-Einheit darstellt, existieren einige Unterschiede zur Standarddefinition.

- Atomarität
  Ein ConTract ist auf Skriptebene nicht atomar. Dies ist der fundamentale Unterschied zum klassisches Modell. Ein ConTract kann explizit unterbrochen und durch den Benutzer nach einer beliebigen Zeit fortgesetzt werden. Viel wichtiger aber ist die

```
STEP Flight_Reservation

        DESCRIPTION:    Reserve n seats of a flight and pay for them ...

        IN              arline          :       STRING,
                        flight_no       :       STRING,
                        date            :       DATE,
                        seats           :       INTEGER,
                        ticket_price    :       MONEY;
        OUT             state           :       INTEGER;

FUNCTION flight_reservation() : INTEGER;
VAR
        flight_no               :       STRING;
        date                    :       DATE;
        seats                   :       INTEGER;
        ...
BEGIN
        ...
        EXEC SQL
                UPDATE Reservation
                        SET     seats_taken = seats_taken + :seats
                        WHERE   flight      = :flight_no
                          AND   date        = :date
                        ...
        END SQL
        ...
        state := ...;
        ...
END;
```

Abbildung 1.3: Auschnitt aus dem Quellcode eines Beispiel-Steps zur „Flugreservierung“

Tatsache, daß ein „Crash“ keine `rollback` auslöst. Das System führt ein vorwärtsgerichtetes Recovery eventuell entlang eines anderen Ausführungspfades aus.

- KONSISTENZ
  ConTracts beinhalten Systemintegrität und dies in einem sehr viel grösseren Maß als einzelne Transaktionen.

- ISOLIERUNG
  Ein ConTract ist eine langlebige Anwendungen. Isolierung gemeinsamer Daten durch herkömliche Sperrmethoden würde verheerende Folgen auf die Performance des Systems haben. ConTracts basieren daher auf semantischer Isolierung durch anwendungsspezifische Invarianten[7] (siehe auch Abschnitt 1.2.10 ab Seite 12).

[7] Invarianten sind Prädiakte auf gemeinsamen Daten. Z.B. Reisekostenbudget > Reisepreis.

- Dauerhaftigkeit
  Die globalen Effekte eines ConTract sind dauerhaft. Eine *undo* ist nur durch Ausführung eines gegenteiligen ConTracts (steps) möglich (analog dem klassischen Transaktionsmodell).

Zur Definition und Verwaltung von Transaktionen besitzt der Skriptprogrammierer einige Möglichkeiten.

- Atomare Ausführungseinheiten
  Einzelne Steps können als atomare Einheit ausgewiesen werden. In Abbildung 1.2 ist dies durch den Rahmen um die Steps S_4, S_5 und S_8, S_9, S_{10} angedeutet. In der Textform entspricht dies der Definition:

  ```
  BEGIN TRANSACTIONS
    T1 ( S4, S5 )
    T2 ( S8, S9, S10 )
  END TRANSACTIONS
  ```
  [8]

- Event–Definition
  Durch die Definition von Events, die an die Ausgabe bzw. an der Aktivierung von Steps oder Transaktionen gebunden werden, besitzt der Skriptprogrammierer die Möglichkeit, den Ablauf zu kontrollieren. Er kann einen Step starten, der eine funktionale Alternative darstellt oder er kann versuchen, einen Fehler zu beheben, um eine Fortführung zu ermöglichen.

  Die Verwendung eines anderen Fluges in obigem Skript ist ein Beispiel für diesen Transaktionkontrollmechanismus. Die textuelle Darstellung lautet:

  $$\texttt{DEPENDENCY (} T_1 \texttt{ abort} \mapsto \texttt{begin } S_2 \texttt{)}$$

  Die Semantik dieser Abhängigkeit ist:

  Wenn T_1 abbricht, dann muß S_2 gestartet werden.

Aufgrund obiger Mechanismen zur Definition und Kombinierung atomarer Einheiten in einem ConTract müssen die Begriffe `begin-/end-of-transaction` und `rollback work` verfeinert werden, um den Beginn, das Ende und den Abbruch von

- der gerade aktiven Transaktion,
- der nächst höheren Kontrollsphäre,
- der obersten Ebene der Kontrollsphäre

und des gesamten ConTracts

zu unterscheiden.

[8] T_1 und T_2 sind logische Namen zur Referenzierung der atomaren Einheiten.

1.2.8 Benutzerschnittstelle zur Kontrolle der Anwendung

Es stehen zur Verwaltung von ConTracts als Ganzes drei wichtige Mechanismen zur Vefügung.

1. KONTROLLE LANGLEBIGE ANWENDUNGEN
   Die Ausführung eines ConTracts kann ausgesetzt (`suspend`) werden und nach einer beliebigen Zeit fortgeführt (`resume`) werden. In der Zwischenzeit muß der komplette Verarbeitungszustand stabil zwischengespeichert werden, um die Weiterführung zu einem beliebigen Zeitpunkt gewährleisten zu können.

   Im Beispiel–ConTract kann es vorkommen, daß ein Kunde einen Flug bucht, dann aber den Reservierungvorgang unterbricht, um ein Hotel und ein Auto vor der Reise (in einem anderen Vorgang) zu buchen.

2. KONTROLLE VERTEILTER ANWENDUNGEN
   Die Verwaltung verteilter Applikationen macht manchmal die Migration eines Vorgangs von einem Rechner auf einen anderen erforderlich. Dies ist z.B. der Fall, wenn ein Rechner „für immer“ ausfällt und der Benutzer dem System mitteilen muß, auf welchem Knoten der Vorgang weitergeführt werden soll.

3. ÜBERWACHUNG VERTEILTER ANWENDUNGEN
   Ein Monitor sollte dem Benutzer Funktionen zur Verfügung stellen, mit denen es möglich ist, den Verarbeitungzustand und den Ort eines ConTracts festzustellen oder der eine Protokollierung („trace“) der Berechnung erlaubt.

1.2.9 Recovery und Kontextverwaltung

1.2.9.1 Information zur Durchführung von Recovery–Maßnahmen

Systemrekonfigurationen, Kommunikationsfehler, Rechnerausfälle und andere Fehler sollten nicht dazu führen, daß Anwendungen undefiniert verlorengehen oder gar unerwünschte Auswirkungen hinterlassen. Ohne zusätzliche Applikationsprogrammierung verhalten sich normale Transaktionen aber in dieser Art und Weise:

- Ein normaler Betriebssystemprozeß wurde abgebrochen und nach dem Neustart des Systems vergessen. Der Benutzer muß wissen, welche Anwendung abgebrochen wurde und welchen Zustand das Programm hatte. Dies erfordert manuelles Recovery und manuellen Neustart.

- Ein Transaktionssystem stellt eine konsistente Datenbank durch das „rollback“ aller nicht abgeschlossenen (comitted) Transaktionen her. Eine Unterscheidung zwischen kurzen und langlebigen Transaktionen findet dabei nicht statt.

Ein zuverlässiges System für langlebige Aktivitäten würde alle nicht abgeschlossenen Vorgänge fortführen (automatisch nach dem Neustart des Systems oder auf Anforderung durch den Benutzer) und versuchen, den Verlust an Arbeit so gering wie möglich zu halten. Sollte im Beispiel–ConTract ein lokaler Rechnerfehler auftreten, möchte der Benutzer an ein anderes Terminal gehen und den unterbrochenen Buchungsvorgang ab dem letzten gültigen (konsistenten) ConTract-Zustand fortsetzen.

Der *ConTract Manager* versucht deshalb, Fehler zu berücksichtigen, indem er einen unterbrochenen ConTract durch das Erzeugen des letzten konsisteten Step–Zustands reinstanziert und anhand der im Script spezifizierten Strategie fortsetzt. Nur ein nicht beseitigbarer Fehler veranlasst das System den ConTract so weiterzuführen, daß alle externalisierten Effekte rückgängig gemacht werden (Kompensation aller abgeschlossenen Steps).

Dieses vorwärtgerichtete Recovery („forward recovery“) impliziert, daß alle Zustandsinformationen, auf die ein Steps aufbaut, „recoverable“sind. Die Menge der privat definierten, applikationsspezifischen Verarbeitungszustände wird **Kontext** genannt. Für die Reinstanzierung eines unterbrochenen ConTracts müssen folgende Informationen „recoverbale“ sein:

- Der globale Systemzustand, den alle Applikationen hatten, insbesondere die beteiligten Datenbanken.
- Den lokalen Zustand von Programmvariablen, Sessions, Windows, Dateideskriptoren, Cursor (SQL-Cursor) etc., die von mehr als einem Step verwendet wurden[9].
- Der gloable Verarbeitungszustand der betroffenen Anwendungen. Dies beinhaltet eine stabile „Buchhaltung“ des ConTract–Systems darüber, welches Ereignis „getriggered“ wurde, welcher Step ausgeführt (oder nicht ausgeführt) wurde etc.

1.2.9.2 Kontextverwaltung

Das ConTract–Modell verwendet persistente Kontextinformationen, um die relevanten lokalen Zustände einer langlebigen Anwendung stabil zu speichern. Die Kontextverwaltung ist zudem charakterisiert durch einen Bindemechanismus für Namen, welcher die Existenz und die Details der Parameterzuordnung von Kontextelementen zu Step–Parametern (und umgekehrt) transparent für den Step–Programmierer hält.

Jeder ConTract hat eine private Kontextdatenbank für die globalen Skriptvariablen, die durch die Steps erzeugt (Ausgabeparameter) oder verwendet (Eingabparameter) wurden. Bei der Skriptprogrammierung werden die Variablen und ihre Typen im `CONTEXT DECLARATION`–Abschnitt angegeben.

Für die Bindung von Kontextelementen gilt folgendes Konzept:

1. Step–Programmierung
   Der Programmierer eines Steps verwendet beliebige Namen für seine Ein- und Ausgabeparmeter, ohne Wissen über die zukünftige Ausführungsumgebung seines Steps, z.B. ohne zu wissen, ob die Werte über eine Eingabenachricht kommen, durch einen normalen Programmiersprachenmechanismus oder durch die Referenzierung von Kontextelementen eines ConTract. Die Trennung von Kontext und Step–Programmierung und die Verwendung eines Kontextbindemechanismus sind Voraussetzungen für die Wiederverwendbarkeit von Programmen.

[9] Diese globalen Variablen oder Zwischenergebnisse werden normalerweise nicht in der Datenbank gehalten. Trotzdem sind sie notwendig, um eine „forward recovery“ realisieren zu können.

2. Definition im Script
   Der Skriptprogrammierer definiert logische Namen für die Referenzierung von Kontextelementen. Bei jedem Step gibt er an, welche Kontextelemente an die Ein- und Ausgabeparameter gebunden werden sollen (siehe Beispiel–Skript ab Seite 85). Ein Parameter kann auch an eine Konstante oder an ein globales Objekt, daß z.B. durch ein SQL Select-Anweisung ein Tupel einer Relation referenziert, gebunden werden.

   Ohne die Verwendung von logischen Namen müßten alle Step–Programmierer für das gleiche Objekt eines ConTracts den gleichen Namen verwenden. Dies ist offensichtlich nicht sehr praktikabel.

3. ConTract Laufzeitsystem
   Da ein Step mehr als einmal aktiviert werden kann (z.B. in einer Schleife, im Rahmen einer Fehlerstrategie etc.), erfordert die eindeutige Identifiaktion eines Kontextelementes neben dem logischen Namen noch zusätzliche Informationen:

   - Einen ConTract–Identifier
   - einen Step–Identifier
   - Zeit und Datum der Erzeugung
   - einen Versionszähler, um verschiedene Aktivierungen unterscheiden zu können
   - und einen Zähler für parallele Aktivierungen eines Steps.

1.2.10 Konsistenz und Konfliktbehandlung

Konfliktbehandlung bei ACID–Transaktionen beruht auf der Isolierung atomarer Transaktionen gegeneinader mit der Abschicht der Serialisierung. Um dieses Ziel zu erreichen, verzögern fast alle Konfliktkontrollmechanismen Updates bis zum Ende der Transaktion. Für langlebige Transaktionen ist dieser Mechanismus nicht geeignet:

1. Ergebnis wäre ein drastischer Performance–Verlust aufgrund der lang gehaltenen Sperren, die andere Aktivitäten blockieren.

2. Die Rate der aufgrund eines Konfliktes oder eines Deadlock abgebrochenen Transaktionen wüde drastisch steigen. Die Wahrscheinlichkeit eines Deadlock steigt mit der vierten Potenz der Transaktionsgröße ([10]).

3. Serialisierbarkeit ist eine ausreichende aber keine notwendige Bedingung für die isolierte Ausführung.

ConTracts sind weder atomar noch kurz. Sie externalisieren einige ihrer Updates beim Ablauf z.B. dadurch, daß Sperren nach Ausfürhung eines Steps freigegeben werden. Es besteht aber noch die Möglichkeit, das diese Updates später zurückgesetzt („rolled back“) werden müssen. Zudem benutzt ein ConTract Daten, die ein anderer ConTract zuvor externalisiert hat. Es existieren zwei Arten von Konsistenzproblemen, die korrekt gehandhabt werden müssen:

1. Falls ein ConTract ruckgängig gemacht werden muß, kann dies nicht durch einfaches Zurückschreiben des „Before Image“ getan werden, vielmehr muß ein semantisches Undo durch Kompensation erreicht werden. In bestimmten Fällen können andere ConTracts von dieser Kompensation betroffen sein. Diese Situation muß entsprechend behandelt werden.

2. Das Freigeben von Sperren ohne weitere Konfliktmechanismen über Transaktionsgrenzen hinaus (wie im Saga–Modell [11]) kann zu Inkonsistenzen führen. Aufgrund dieses Risikos muß es für einen Step möglich sein, Isolierungsanforderungen zu spezifizieren.

1.2.10.1 Kompensation

Für die korrekte Anwendung der Kompensation müssen folgende Kriterien erfüllt sein:

- Für jeden Step muß genau ein gültiger Kompensations–Step exisitieren.

- Nach dem Ende eines Steps müssen alle Eingabedaten verfügbar sein. Wenn die momentanen Werte von globalen Objekten für die Kompensation benötigt werden, müssen sie ebenfalls im Kontext gesichert werden.

- Alle globalen Objekte, die ein Step verwendet hat und die für die Kompensation von Bedeutung sind, müssen bis zum Ende des ConTracts existieren. Z.B. müssen benutzte Relationen (Tupel) einer Datenbank ggf. durch „Existenzsperren“, die ein Löschen verhindern, geschützt werden. Wenn dies nicht garantiert wird, besteht die Möglichkeit, eine Relation zu löschen und neu anzulegen, welche denselben Namen und dieselbe Struktur besitzt, aber eine völlig andere Bedeutung besitzt. Dieser Fall ist eine Konsistenzverletzung und die Ausführung der Kompensation macht keinen Sinn.

- Für jeden zuvor ausgeführten Steps S_i, für den ein „committed“ Eintrag im Log existiert, muß C_i — der zugehörige Kompensations–Step — ausgeführt werden. Alle Kompensationstransaktionen müssen irgendwann im Zustand „committed“ sein. D.h. nicht, daß eine Kompensation nicht scheitern kann. Es ist durchaus erlaubt, eine abgebrochene Kompensation mehrfach zu starten, solange bis sie einmal „committed“ ist.

- Sollte die Ausführung einer Kompensationsaktion nicht erfolgreich beendet werden können, kann nicht weiter getan werden, denn eine Kompensation der Kompensation gibts es nicht (dies könnte eine Endlosschleife erzeugen). Die Strategie des ConTract Systems beim Eintretten eines solchen Fehlers ist es, einem Systemadministrator zu benachrichtigen und ihm durch einen „snapshot“ den aktuellen ConTract–Zustand zu vermitteln.

1.2.10.2 Synchronisation durch Invarianten

Zur Lösung der Synchronisationsaufgaben, die durch gleichzeitige Abarbeitung mehrerer ConTracts entstehen, kann die Verallgemeinerung einer Idee, die für eine spezielle Art von

„hot spots“[10] entwickelt wurde [20], verwendet werden. Die Synchronisation erfolgt nicht durch das Setzen von Sperren, sondern durch Prädikate, die auf (Daten-)Objekten gehalten werden. Aus Anwendungssicht betrachtet: Kein Programm braucht die Serialisierbarkeit oder kümmert sich darum, ob es serialisierbar ist. Das einzige Interesse besteht darin, die Datenbank frei von ungewollten Änderungen anderer zu halten. Zum Freihalten der Datenbank von ungewollten Änderungen müssen nicht Attribute, Tupel, etc. vor Änderung geschützt werden. In vielen Fällen reicht es z.B. aus, nur ein Tupel vor dem Löschen zu schützen, nur Attributwerte aus einem bestimmten Bereich zuzulassen oder zu überprüfen, ob nicht mehr als n Tupel eines bestimmten Typs existieren.

Um die Idee der Synchronisation durch Invarianten im ConTract System zu implementieren, sind zwei Dinge von Bedeutung:

1. Es muß möglich sein, Invarianten aus ConTract–Sicht zu spezifizieren, nachdem ein Step ausgeführt wurde. Diese Nachbedingung, die eine Isolierungsanforderung formuliert, wird **Ausgangsinvariante** (exit invariant) genannt. Das Etablieren dieser Nachbedingung heißt, die momentanen Werte gemeinsamer Objekte an Variablen des Prädiaktes zu binden.

   Eine beispielhafte Ausgangsinvariante ist dem Skript (ab Seite 85) zu entnehmen:

   ```
   budget >= limit
   ```

   Die „exit invariant“ von Step $S1$ formuliert, daß das Budget durch den Buchungsvorgang (`limit`) nicht überschritten werden darf.

2. Die Angabe, welche zuvor definierte Ausgangsinvariante erfüllt sein muß, um einen späteren Step ausführen zu können, ist der andere Punkt. Diese Vorbedingung wird **Eingangsinvariante** (entry invariant) genannt. Beispiele sind wiederum dem Skript zu entnehmen.

1.2.10.3 Konfliktbehandlung

Die spezifizierten Invarianten müssen durch den *ConTract Manager* und die Datenbank überwacht werden. Möglichkeiten hierfür sind:

- Um Änderungen zu verhindern, können Objekte wie bisher gesperrt werden. Der *ConTract Manager* und das DBMS müssen dann mit langen Sperren, die u.U. über Transaktionsgrenzen hinaus gehalten werden müssen, umgehen können.
- Eine alternative Technik ist das „escrowing“ [19]. Der Wert eines Attributes a darf sich bei dieser Methode innerhalb eines spezifizierten Intervalls u bewegen. Operationen, die eine solche Invariante verletzen würden, werden abgelehnt.
- Der liberalste Ansatz ist es, keine Sperren zu verwenden. Dies erfordert eine „check/revalidate“ Technik [20]. Nach dem Etablieren einer Ausgangsinvariante können die spezifizierten Datenbankobjekte durch andere ConTracts beliebig

[10] Gleichzeitiger Zugriff vieler Transaktionen auf eine kleine Menge von Tupel.

verändert werden. Zum Zeitpunkt, an dem ein Step mit dieser Invariante ausgeführt werden soll, wird sie einfach neu überprüft (revalidate). Wenn die Evaluierung erfolgreich ist, wurde die Isolierungsanforderung des ConTract aus seiner Sicht erfüllt. Diese Art der Konsistenzdefinition erlaubt nicht-serialisierbare Ausführungsfolgen von Transaktionen, erfüllt aber die durch die Applikation gesetzten Anforderungen.

Wenn die Evaluierung scheitert, kann der nächste Step nicht ausgeführt werden — im Beispiel könnte das Budget überschritten sein. ConTracts erlauben explizit über diese Konflikte zu kommunizieren und Lösungen zu definieren. Im Buchungsbeispiel muß jemand das Budget erhöhen, da nicht genügend Geld vorhanden ist, oder der ganze Buchungsvorgang muß abgebrochen werden.

1.3 Aufgabenstellung der Diplomarbeit

Ausgangspunkt für die Arbeit war das ConTract–Modell und Überlegungen, dieses in eine prototypische Ausführungsumgebung einzubetten.

Das APRICOT System besteht aus einer Reihe von Komponenten, die während einer ConTract–Abwicklung miteinander kommunizieren. Die dabei auftretenden Kommunikationsmuster lassen sich nicht ohne weiteres mit herkömmlichen Mechanismen (z.B. RPC) realisieren. Beispielsweise kann es vorkommen, daß zwei Prozesse geleichzeitig versuchen, sich gegenseitig Nachrichten zu senden. Die Aufgabe der Diplomarbeit ist es, die auftretenden Kommnikationsmuster zu erfassen, zu klassifizieren und ein geeignetes Communication System zu entwerfen und zu implementieren.

1. Beschreibung der Kommunikationsmuster
   Die Komponenten sollen Nachrichten auf einer Ebene miteinander austauschen, die von allen Aspekten der physischen Programm-zu-Programm Kommunikation abstrahiert. Die Abläufe im System sind zu erfassen und geeignet zu klassifizieren.

2. Entwurf des Kommunikationsmodells und des Adressierungschemas
   Die Kommunikation soll unabhängig von der Lokation in einem Netzwerk sein. Dazu soll auf Programmebene ein Mechanismus

   nachricht (*sender, empfänger, inhalt-der-nachricht, antwort*)

   existieren, der den logischen Namen der Empfängerkomponente in deren physische Adresse umsetzt (*name service*) und die Nachricht über den entsprechenden Kanal dorthin versendet.

3. Transaktionsorientierte Protokolle
   Die Interaktion der Komponenten soll in ein mit Transaktionsinformation angereichertes Protokoll eingebettet werden. Dazu müssen vorhandene Protokollvorschläge (etwa die *XA Sepzifikation „Distributed Transaction Processing“* von X/OPEN u.a.) entsprechend dem APRICOTS Betriebsmodell erweitert werden. Hinzu kommen u.a. transparente Aufrufe des Kommunikationsdienstes an den *Transaction Manager*, um die Ausbreitung einer Transaktion auf entfernte Knoten sowie dort anfallende Verwaltungsinformationen (Abhängigkeitsbeziehungen zwischen Transaktionen, etc.) zu erfassen.

Lokale Systemfehler, wie z.B. ein Rechnerausfall, dürfen die ConTract–Verarbeitung auf anderen Knoten nicht blockieren oder sonstwie beeinträchtigen.

Randbedingungen waren:

- Die Funktionsfähigkeit des implementierten Communication System soll an der Abwicklung von Beispielabläufen im APRICOT System demonstriert werden. Die Entwicklung geschieht im Rahmen des DFG–Projektes „Objektbanken für Experten“. Dieses umfaßt mehrere Diplom– und Studienarbeiten.
- Die Implementierung erfolgt unter dem Betriebssystem *SUN–OS* auf Workstations der Firma *SUN Microsystems*. Die erstelleten Programme sollen auch auf Rechnern anderer Hersteller zum, Einsatz kommen. Deshalb ist auf Portabilität zu achten.

Kapitel 2

Das APRICOT System

2.1 Anforderungen

Das APRICOT System[1] wird entwickelt um die Tauglichkeit der im ConTract–Modell vorgeschlagenen Mechanismen zu überprüfen. Im folgenden Abschnitt soll erläutert werden aufgrund welcher Anforderungen des ConTract–Verarbeitung und der Zielsetzung von ConTracts, welche Aufgaben zu erfüllen sind.

2.1.1 Ausgangspunkt für das ConTract–Modell

Bereits in der Aufgabenstellung, welche der Entwicklung der ConTract–Modells zugrunde lag, wurden einige Anforderungen formuliert:

- Verarbeitungsmodell für *langlebige, verteilte* Anwendungen.
- Programmiermodell mit hohem Abstraktionsniveau.
- Fortführungsmöglichkeit im Fehlerfall.
- Nutzung der Vorteile einer verteilen Verarbeitung.
- Kontrollmechanismen zur Unterbrechung, Migration und Fortsetzung einer Anwendung im Netzwerk.

2.1.2 Anforderungen durch die ConTract–Verarbeitung

Aus dem in Abschnitt 1.2 dargestellten Verarbeitungsmodell ergeben sich weitere Anforderungen an die Ausführungsumgebung:

- VERWALTUNG DER ConTract–AUSFÜHRUNG
  Die Ausführung eines ConTracts muß überwacht werden. Dazu ist ein robustes Laufzeitsystem erforderlich, daß folgende Hauptaufgaben hat:

[1] A Prototyp Implementation of the ConTract System

1. Eventverwaltung
   Events müssen stabil verwaltet werden.
2. Ablaufsteuerung
   Steps dürfen nicht scheitern, ohne das Maßnahmen zum Neustart (ggf. auf einem anderen Knoten), zur Recovery oder zur Kompensation eingeleitet werden.

- TRANSAKTIONSVERWALTUNG
  Bezüglich der Transaktionsverwaltung sind folgende Aspekte von Bedeutung:
  - Der Programmierer eines ConTract–Scripts kann einzelne Steps zu einer atomaren Einheit zusammenfassen. Der *ConTract Manager* benötigt eine allgemeine Schnittstelle zum *Transaction Manager*.
  - Zur Abarbeitung von Steps wird eine spezielle Art verschachtelter Transaktionen benötigt. Die Step–Ausführung teilt sich in verschiedene Subtransaktionen auf, den eigentlichen Stepcode und Pre- und Postprocessing (Evaluierung der Invarianten, Kontextbindemechanismus, etc.). Wenn ein Step nun Operationen auf einer Datenbank ausführt, kann die Commit–Entscheidung nicht nur vom DBMS getroffen werden. Deshalb ist eine Offenlegung der Commit–Protokolle und ein `prepare-to-commit`–Aufruf erforderlich.
  - Da ein ConTract System von einem Knoten auf einen anderen mirgrieren kann, muß ein Knoten als Commit–Koordinator bestimmt werden können.

- KOMPENSATION
  Sollte die ConTract–Ausführung scheitern (keine weitere Vorwärtsstrategie ist spezifiziert), wird der ConTract kompensiert. Die dazu benötigten Kompensations(trans)aktionen müssen bis zum Ende des ConTracts verfügbar sein.

- KONTEXTVERWALTUNG
  Die Aufgaben der Kontextverwaltung (vgl. Abschnitt 1.2.10.1 ab Seite 13) müssen von einer Instanz übernommen werden. Dies hängt mit der Aufgabe eines globalen Logs (nachfolgender Punkt) zusammen.

- LOGGING
  Die Realisierung einer robusten, verteilen Anwendung ist eng verbunden mit einem globalen, verteilten Logservice. Z.B. muß Loginformation auch bei Ausfall eines Knotens oder bei Migration einer Anwendung noch verfügbar sein. Das gleiche gilt für zuvor angesprochene Kontextverwaltung. Unterschiede bestehen in der Referenzierung der im Log gespeicherten Daten.

- RECOVERY
  Aufgabe ist die Gewährleistung korrekter Recovery–Maßnahmen im verteilten System.

- SYNCHRONISATION
  Zur Synchronisation mithilfe von Invarianten wird ein logisches Kalkül benötigt um die spezifizierten Vor– und Nachbedingungen zu evaluieren. Im Zusammenhang mit Datenbankanwendungen stellt SQL eine nutzbare Möglichkeit dar.

- KOMMUNIKATION
  Der Kommunikation in verteilten System kommt eine besondere Bedeutung zu. Fehler in einer Verbindung dürfen bei langlebigen Anwendungen nicht zum Abbruch führen. Hier müssen geeignete Mechanismen realisiert werden.

- RESOURCENVERWALTUNG
  Verteilte Systeme zeichnen sich u.A. dadurch aus, daß Dienste mehrfach vorhanden sind. Zur sinnvollen (gleichmäsigen, optimierten) Nutzung muß eine Verteilung der Aufträge — eine Lastbalancierung — vorgenommen werden.

2.2 Die Komponenten im APRICOT System

Zur Lösung der oben definierten Aufgaben werden im APRICOT System unten aufgeführte Komponenten eingesetzt. Es ist nicht Aufgabe dieser Arbeit, deren Aufgabe genau zu spezifizieren. Deshalb erfolgt nur eine kuze Beschreibung.

1. *ConTract Manager* (CM)
   Der *ConTract Manager* veranlaßt und überwacht die Ausführung der ConTracts (siehe [28]). Hierzu gehört die Interpretation des Skriptes, die Eventverwaltung, die Abbildung von Steps auf Transaktionen[2] etc.

2. *ConTract Client* (CC)
   Der *ConTract Client* realisiert die Schnittstelle (shell) des APRICOT System zum Benutzer. Die Minimalfunktionalität umfaßt das Starten, Abbrechen, Aussetzen, Migrieren und Fortsetzen eines ConTracts.

3. *Transaction Manager* (TM)
   Der TM verwaltet Transaktionen, überwacht deren Beziehungen untereinander und dient als Commit–Koordinator (siehe [17]).

4. *Compensation Manager* (COM)
   Kompensationsaktionen zu Steps werden im Skript definiert. Der COM übernimmt die Aufgabe die Kompensations(trans)aktionen bis zum Ende der ConTract–Ausführung sicher zu verwalten. Hierzu gehört u.A. das der Code, der die Aktion implementiert, nicht gelöscht wird. Nur wenn sichergestellt ist, daß die Kompensation ausgeführt werden kann ist die Unverwundbarkeit eines ConTracts gewährleistet.

5. *ConTract Processing Monitor* (PM)
   Aufgabe des PM ist die Verwaltung der Betriebsmittel, welche die Systemkomponenten des APRICOT System benötigen. Hierzu zählen:

   - SYSTEMKOMPONENTEN
     Die Systemkomponenten selbst sind Betriebsmittel. Für sie ergeben sich folgende Aufgaben:
     - „Erzeugen“ (Start) der Komponenten beim „Hochfahren“ des Systems.

[2] Mit dem TM kann nur über Transaktionen geredet werden.

  - Überwachung der Verfügbarkeit einer Komponente und ggf. Neustart.

  Optionale Betriebsmittel wie *Step Computation Server* oder *Object Manager* (siehe unten) werden beim *ConTract Processing Monitor* konfiguriert. Aufgrund der Konfigurationsinformation startet und überwacht der PM die Komponenten.

- LASTBALANCIERUNG
  SS werden beim PM angemeldet. Danach können ihre Dienste den Benutzern zugänglich gemachen werden. Aufgabe des PM ist es den Dienstleistungswunsch einer Anwendung (insbesondere eines Steps) einem der möglichen *Step Computation Server* zuzuteilen. Berücksichtigt werden muß der momentane Zustand (verfügbar, ausgefallen etc.), die Auslastung, die Kosten für die Übermittlung des Auftrages und weitere Kosten.

6. Communication System (CS)
   Das CS hat geeignete Kommunikationsmechanismen bereitzustellen.

7. *Global Log Manager* (LM)
   Der LM ist verantwortlich für einen robusten, verteilten, globalen Log–Service.

8. *Recovery Manager* (RC)
   Die Ausführung von Recovery–Maßnahmen obliegt dem RC.

9. *Application* (AP)
   Das Systemarchtiektur soll offen sein. Deshalb kann der allgemeine Typ *Application* am System partizipieren.

   Die *Applikation* stellt eine Einheit dar, die einzelne Komponenten des APRICOT System benutzt um ihre Aufgaben zur erfüllen. Z.B. kann mit Hilfe des Name–Service des PM eine Verbindung mit einer anderen Anwendung hergestellt werden.

Die oben aufgeführten (System-)Komponenten bilden die Basis des APRICOT Systems. Sie müssen aber noch um die eigentlichen Dienste ergänzt werden. Wie in anderen Systembeschreibungen ([32], [2]) werden diese allgemein als *Resource Manager* (RM) bezeichnet. Zur Ausführung von ConTracts werden jedoch zwei spezielle Arten von Resource Manager unterschieden:

1. *Step Computation Server* (SS)
   Ein *Step Computation Server* stellt dem Benutzer des ConTract Systems Services zur Stepausführung zur Verfügung. Beispiel hierfür ist ein Server, der Flugverbindungen ermittelt (vgl. Step $S2$ im Beispiel–Skript, Anhang A ab Seite 85).

   Step Computation Server arbeiten speziell ConTract–Steps ab, die unter Transaktionsschutz ablaufen. D.h., die von SS angebotenen Dienste stellen ACID–Transaktionen dar und müssen „recoverable"sein.

   Ein SS muß nicht selbst Recovery–fähig sein. Es kann zum Beispiel Aktionen auf einem Datenbestand ausführen, der durch ein DBMS geschützt ist.

2. *Object Manager* (OM)
   OM sind im Gegensatz zu SS einfacher aufgebaut. Sie arbeiten nur einen Step ab und werden bei Bedarf aktiviert. Bezüglich Transaktionen gilt das beim SS gesagte.

3. *Resource Manager* (RM)
   Alle anderen Dienstleistungsanbieter (nicht SS, OM) werden als *Resource Manager* bezeichnet. Die angebotenen Dienste können transaktionsgeschützt sein (Datenbanksystem), müssen es aber nicht (Filesystem). Für ein *verteiles Transaktionssystem* stehen aber RM mit Transaktionmechanismus im Vordergrund.

Bemerkungen:

- SS und OM sind spezielle RM. Sollte im folgenden eine Unterscheidung zwischen ihnen und allgemeinen RM nötig sein, werden diese explizit angeführt.
- Die Basis–Systemkomponenten (CC, CM, TM, PM, COM, LM, RC) könnten aus Benutzersicht ebenfalls als *Resource Manager* bezeichnet werden. Sie bilden aber vielmehr eine Ebene, die als „Transaction Processing Operating System (TPOS)“ bzw. als „Basic Operating System (BOS)“ [22] bezeichnet wird. Die Ebene der Anwendung wird in diesem Umfeld „Transaction Processing Services (TRAPS)“ genannt.
- Zur Beginn der Diplomarbeit waren zwar alle Komponenten angedacht, im Blickfeld der Realisierung standen aber nur ein Teil davon. Schwerpunkt der Untersuchung waren der CM, TM, SS und das CS.
- Aufgrund der engen Zusammenarbeit zwischen COM und CM wird im Prototyp keine expliziter *Compensation Manager* eingesetzt. Kompensation ist als Aufgabe beim *ConTract Manager* angegliedert.

2.3 Die Systemarchitektur

Die Liste der Komponenten ist nur ein Teil der Systembeschreibung. Im folgenden sollen weitere Aspekte erläutert werden.

Welche Aufgabenteilung besteht zwischen den Komponenten?
Wie groß ist der Geltungsbereich einzelner Komponenten?
Wieviele Komponenten einer Klassen können gleichzeitig existieren?

Ausgeführt werden sollen ConTracts in einer verteilten Umgebung. Dies impliziert, daß eine zentralistische Architektur — zumal in großen Netzen — wenig sinnvoll ist. Gründe sind u.A.:

- Verfügbarkeit wird durch zentrale Enheit bestimmt.
- Erhöhter Kommuniktionsaufwand mit Zentralrechner.
- Höherer Parallelitätsgrad in verteilten Umgebungen.

Deshalb sieht die APRICOT Systemarchitektur vor auf jedem Knoten eine komplette Umgebung zu installieren. Die meisten der nachfolgenden Angaben basieren deshalb auf der Annahme, daß die Basiskomponenten auf jedem Rechner verfügbar sind.

1. *ConTract Client*
   - Ein CC arbeitet für einen Benutzer. D.h., auf einem Knoten können mehrere CC aktiv sein.
   - Ein Benutzer kann mehrfach im System sein. Der CC muß dies ggf. erkennen und koordinieren. Über ein Protokoll einigen sich die beteiligten *ConTract Client*s, wer welche Aufgaben übernimmt.
   - Über den CC kann ein Benutzer einen ConTract Starten, Abbrechen, Ausetzen, Migrieren, Fortsetzen und Statusabfragen durchführen.
2. *ConTract Manager*
   - Der CM ist immer lokal vorhanden.
   - Ein CM bearbeitet immer die Aufträge/Anweisungen der lokalen *ConTract Clients*.
   - Der CM ist für alle lokalen CC's (und damit für alle lokalen Benutzer zuständig). D.h., er arbeitet als Server im Multiuser–Betrieb.
   - Durch Migration oder Fehlerfälle kann der CM zuständig für entfernte CC's oder CM werden.
3. *ConTract Processing Monitor*
   - Ein PM ist immer lokal vorhanden.
   - Ein PM überwacht den Zustand der lokalen Komponenten und veranlaßt ggf. den Neustart einer Komponente im Fehlerfall.
   - Ein PM überwacht den Zustand entfernter PM's und teilt eine Veränderung den anderen PM mit. Diese Informationen über die Verfügbarkeit sind Bestandteil des Algorithmus zur Lastbalancierung.
   - Die Verfügbarkeit neuer Dienste (SS, OM), oder das Ende der Verfügbarkeit von Diensten, muß ebenfalls im Netz bekanntgemacht werden.
4. Communication System
   - Alle Komponenten kommunizieren über das Communication System.
   - Die Adressierung der einzelnen Komponenten soll transparent sein. D.h., die Anwendung soll nichts über die Verteilung von Komponenten und Dienste im Netzwerk wissen.
5. *Transaction Manager*
   - Lokal exisitert ein TM.
   - Der TM ist für alle lokal laufenden Transaktion zuständig.
   - Ein TM kann Koordinator für Transaktionen werden, die auf verschiedenen Knoten ausgeführt werden bzw. noch nicht abgeschlossen (`prepare-to-commit`) sind.
6. *Step Computation Server*

- Es kann beliebig viele *Step Computation Server* auf einem Knoten geben.
- SS werden beim PM konfiguriert. Dieser startet und überwacht sie. Bei Bedarf erteilt der PM einem SS Aufträge.

7. *Object Manager*
   - Es kann beliebig viele OM auf einem Knoten geben.
   - OM's werden beim PM konfiguriert. Soll ein *Object Manager* aktiviert werden, so veranlasst dies der *ConTract Processing Monitor*.
   - Eine Anmeldung beim PM nach dem Aufruf ist nicht erforderlich (Ausführungsende eines Auftrags wird explizit gemeldet). Gleiches gilt für das Abmelden.
8. *Application*
   - Es kann beliebig viele *Applikationen* auf einem Knoten geben.
   - AP's unterliegen keinerlei Einschränkungen.
9. *Compensation Manager*
   - Lokal exisitert ein *Compensation Manager*.
   - Der COM ist, analog dem *ConTract Manager*, für lokale ConTracts bzw. für die Kompensationstransaktionen, zuständig.
   - Im Fehlerfall oder durch Migration kann ein COM Aufgaben entfernter COM übernehmen.
10. *Resource Manager*
    - Ein allgemeiner RM wird im Gegensatz zum SS und OM nicht vom *ConTract Processing Monitor* verwaltet. D.h., RM's melden sich nicht bei PM an. Wenn *Step Computation Server* oder *Object Manager* weitere *Resource Manager* benutzen, kommunizieren sie direkt, ohne Einschaltung des PM, miteinander und benutzen dabei i.A. die „native Schnittstelle“ des RM. Beispiel ist eine Datenbank. Die Anwendung kommuniziert mit dem DBMS durch direkte Funktionsaufrufe. Funktionen werden in Form von Bibiliothek (Library) dem Anwendungscode zur Verfügung gestellt.
    - Da der *ConTract Processing Monitor* den RM nicht explizit kennt, findet auf seine Dienste keine Scheduling statt.

2.4 Warum diese Architektur?

Welche Gründe sprechen für die gewählte Architektur?

- AUFGABENTEILUNG
  Die gewählte Komponenten repräsentieren die Gliederung der Gesamtaufgabe in kleinere logische Einheiten.

- VERTEILUNG
  Die Verteilung der Aufgaben auf mehrer Prozesse, Threads oder Knoten wird unterstutzt. Die Architektur ist modular und erweiterbar.

- NETZVERTEILUNG
  In großen verteilten Systemen kann nur ein kooperativer Verbund autarker Einzelsystemen die gewünschten Effekte (Fehlertoleranz, bessere Nutzung der Resourcen) erbringen.

- INTEGRATION
  „Das Rad sollte nicht neu erfunden werden.“ Deshalb wurden andere Ansätze und Architekturvorschläge berücksichtigt [13][32][2]. Sie alle definieren im Grundsatz dieselbe Architektur:

  - Applikationen nutzen mehrerer *Resource Manager* ihm Rahmen einer Transaktion.
  - Die Koordination von Transaktionen erfolgt durch einen *Transaction Manager*. Allerdings mithilfe unterschiedlicher Commit–Protokolle.
  - Ein Communication System (oder Communication Manager) übernimmt die Aufgabe der Integration anderer Systeme mit den zugehörigen Konvertierungen (Protokollkonvertierung etc.).

Anmerkungen:

- Obige Auflistung impliziert das jeder Komponente in einem eigenen Prozeß abläuft. Der erste Realisierungsansatz folgte auch dieser Implikation, nicht zuletzt aufgrund der einfacheren Aufgabenteilung. Ein andere Variante stellt aber ein „Ein–Prozeß–Ssystem“ dar.

  1. *Mehr–Prozeß–System*
     Die einzelnen Komponenten werden in getrenten Prozessen ausgeführt. Jede Kommunikation ist damit Interprozesskommunikation, d.h., bei sychronen Aufruf sind auf jeden Fall zwei Prozeßwechsel (Sender $\mapsto$ Empfänger $\mapsto$ Sender) erforderlich.

     Diese Art der Implementierung der zuvor beschriebenen Architektur war Ausgangspunkt der Überlegungen zur Realisierung eines Communication System.
  2. *Ein–Prozeß–System*
     Überlegung nach der ersten Implementierungsphase gehen dahin, wie in anderen Transaktionssystemen auch (CICS, UTM, siehe u.A. [31]), nur einen bzw. wenige Prozesse zu benutzen. Die Funktionalität der einzelnen Systemkomponenten wird hier in Form von Bibliotheken zum Code der ConTract–Instanz dazugebunden. Die Kommunikation kann dadurch auf einfache Prozeduraufrufe reduziert werden. Der Zugriff auf gemeinsame Daten muß durch geeigente Mittel (z.B. Semaphore) synchronisiert werden.

     Die Zusammenlegung in einen Prozeß läßt sich aber nicht ohne weiters auf alle Komponenten ausweiten. Allgemeine RM, wie z.B. ein Datenbanksystem, können i.A. nicht als ein Prozeß ausgeführt werden. In diesem Fall ist eine Aufteilung auf mehrere Prozesse unumgänglich. Als Konsequenz ergibt sich hieraus, daß eine Kommunikation zwischen verschiedenen Prozessen, auch auf einem Rechner" nicht vermeidbar ist. D.h., das Communication System für die Interprozeßkommunikation dient in diesem Fall als Basis und muß um weiteren „Features“ ergänzt werden.

2.5 Konsequenzen aus der Architektur

Welche Konsequenzen ergeben sich aus der Wahl der Architektur?

- KOMMUNIKATIONSAUFWAND
  Die Trennung der Aufgaben in verschiedene Komponenten bedingt einen erhOhten Kommunikationsaufwand zwischen diesen. Allgemein ist der Bedarf an Kommunikation in verteilten System erheblich größer.

- NEUE ANFORDERUNGEN
  Die Trennung von Kontrollfluß und Ausführung bei der ConTract–Verarbeitung führt zu neuen Anforderungen an einzelnen Komponenten.

  - *Event–Mechanismus*
    Der *ConTract Manager* muß seine Vewaltung auf einem Event–Mechanismus aufbauen ([28]), der sich auch auf die Schnittstelle zum *Transaction Manager* auswirkt (siehe hierzu `define_ta_event`–Aufruf des CM an den TM im nachfolgenden Kapitel über die Abläufe im APRICOT System, Seite 29).
  - *Step Computation Server*
    Die Abarbeitung eines ConTract–Steps erfolgt mithilfe spezieller „Server“. Sie haben zusätzliche Aufgaben wie Kontextbindung durchzuführen.
  - *ConTract Processing Monitor*
    Der PM muß eine globale und eine lokale Latsbalancierung durchführen.

- GLOBALE SICHTEN
  Verschiedene Systemkomponenten (CC, CM, PM, TM) benötigen für ihren Teilbereich eine globale Sicht auf das ganze System:

  - *CC*
    Ein Benutzer kann an verschiedenen Terminals angemeldet sein. Das System sollte ihm eine einheitliche Sicht bieten, die unabhängig von der Wahl des Terminals und des Rechenrs ist.
  - *CM*
    Für einen ConTract ist zu einem Zeitpunkt nur eine *ConTract Manager* verantwortlich. Den zuständigen CM zu einem ConTract zu ermitteln bedarf ebenfalls globaler Informationen.

– *PM*
  Zur globale Lastbalancierung muß auch die entsprechende Information vorhanden sind. Hier kommt als weiteres Kriterium die Aktualität hinzu. Lastbalancierung ist nur auf halbwegs zuverlässlichen und damit akutellen Informationen möglich.
– *TM*
  Der *Transaction Manager* muß von der Ausweitung einer Transaktion auf andere Rechner erfahren und sich ggf. mit den dort ansässigen TM koordinieren.

Kapitel 3

Abläufe im APRICOT System

Im folgenden Abschnitt soll die Abwicklung eines ConTract aus Sicht des Kommunikationssystems, d.h., anhand der auftretenden Nachrichten zwischen den Komponenten, beschrieben werden. Diese Beschreibung dient als Basis zur Klassifikation der Kommunikationsmuster und der daraus resultierenden Anforderungen an das Communication System.

Bemerkungen zu den Schaubildern:

- In den Schaubilder wird jeweils nur eine ConTract–Ausführung betrachtet.
- Die in den Abbildungen eingezeichnenten Nachrichten sind danach zu unterschieden, ob sie einen Aufruf darstellen der ein Ergebnis zurückliefert (Hin- mit Zurückpfeil) oder nur eine Benachrichtigung (Message) des Empfängers (Einzelpfeil) sind.
- Gestrichelt eingezeichnete Zeiten lassen erkennen man eine Komponenten andere Tätigkeiten ausführen kann.
- Der *Global Log Manager* und der *Recovery Manager* wurden weggelassen. Einerseits um die Übersichtlichkeit zu bewahren, andererseits weil diese beiden Komponenten bei der Prototypentwicklung nur am Rande berücksichtigt wurden. Aufrufe an den LM können von jeder Komponente jederzeit erfolgen. Der RC wird speziell beim Wiederanlauf benötigt.

3.1 Grobablauf eines ConTracts mit lokaler Step-Ausführung

Abbildung 3.1 beschreibt die Vorgänge bei lokaler Abarbeitung. Dabei soll auf Aspekte wie verschachtelte RM–Aufrufe vorerst nicht eingegangen werden. Den Ablauf beim Commit einer Transaktion wird im Abschnitt 3.3 beschreiben.

Die Schritte im einzelnen:

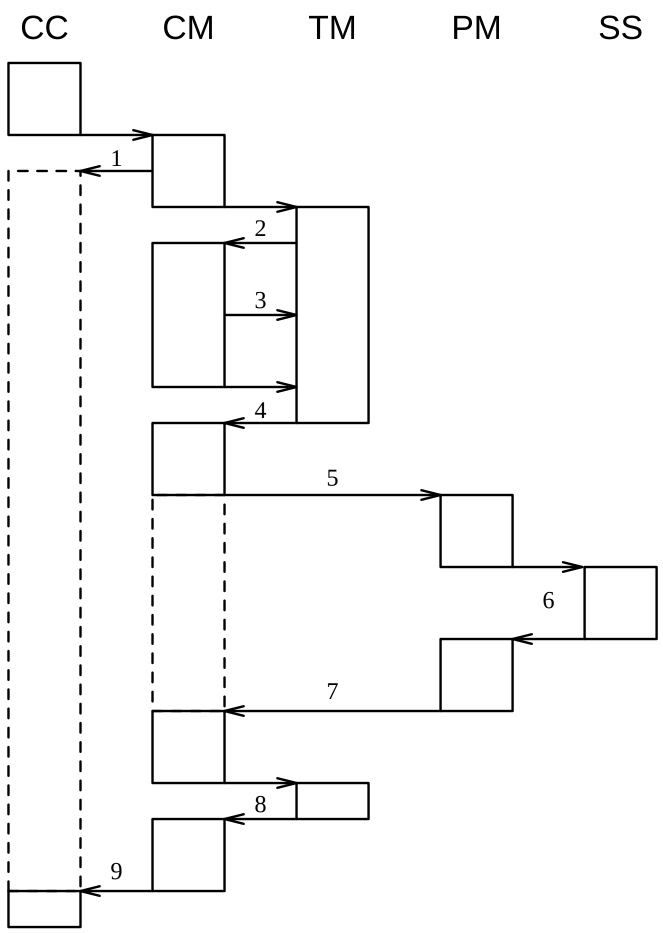

Abbildung 3.1: Grobablauf mit lokaler Step–Ausführung

1. *CC* ↔ *CM*: `activate_contract`
   Der Benutzer beauftragt den *ConTract Manager* über den CC mit der Ausführung eines ConTracts. Der CM lädt das entsprechende ConTract–Skript und führt diverse Initialisierungen durch [28]. Wenn einer oder mehrere Steps ausgeführt werden können kommt es zu nachfolgendem Ablauf. Ergebnis des Aufrufs ist der Identifier der ConTract–Instanzierung (cid).

2. *CM* ↔ *TM*: `get_gtrid_list`
   Der CM wendet sich an den TM um eine Liste von (globalen) Transaktions-Identifiern (gtrid) zu erhalten. Diese werden den gerade auszuführenden Steps zugeordnet um eine Abbildung von Steps auf Transaktionen zu erreichen. Der TM registiert die vergebenen gtrid's als neue Transaktionen im Zustand „installed“ (Bem.: Im X/Open XA–Modell [32] existiert dieser Zustand nicht. Die Transaktion ist gleich im Zustand

„active“).

3. *CM* → *TM*: `create_dependency`
   Sollten zwischen einzelnen Steps (genauer zwischen den zugeordneten Transaktionen) Abhängigkeiten bestehen, so werden diese dem *Transaction Manager* bekanntgegeben. Zur Identifikation der instanzierten Transaktionen dienen die durch Schritt 2 angeforderten gtrid's. Der Aufruf kann bei Bedarf mehrfach erfolgen, je nach Art und Anzahl der Beziehungen. Der TM verwaltet die Beziehungen im Transaktions–Beziehungs–Netz [17].

4. *CM* ↔ *TM*: `define_ta_event`
   Mit diesem Aufruf definiert der CM, daß er bei Eintritt eines Ereignisses (bzgl. einer Transaktion) benachrichtigt werden will. Ein Beispiel für die Verwendung dieses Aufruf ist dem Beispielskript zu entnehmen (grahpische Darstellung auf Seite 7):

   > Die Steps $S4$ und $S5$ sollen atomar ausgeführt werden. Der Name dieser zusammengefaßten Einheit ist $T1$. Bei Abbruch von $T1$ soll eine andere Flugverbindung ermittelt werden (durch Step $S2$). Der CM definiert ein `ta_event (`$T1$`, ‘aborted’)` um bei Abbruch erneut $S2$ ausführen zu können.

   Als Ergebnis liefert dieser Aufruf einen Event–Identifikator. Wie aus dem Buchungsbeispiel ersichtlich ist, wird der Aufruf nicht bei jedem Step benötigt.

5. *CM* → *PM*: `schedule_steps`
   Der *ConTract Manager* übergibt die ausführbaren Steps dem PM. Dieser bestimmt aus der angegebenen Service–Klasse einen *Step Computation Server* der die Ausführung übernimmt. Für diese Abbildung sind mehrere Schritte erforderlich:

   (a) LOGISCHER DIENSTNAME ↦ RECHNERLISTE
   Aus der geforderten Dienstklasse wird eine Liste mit Rechner bestimmt, auf denen „Server“ verfügbar sind.
   Beispiel:
   Dienstname `Flugverbindung` wird abgebildet auf die Rechnerliste $(\mathrm{Node}_1, \mathrm{Node}_2, \ldots)$.

   (b) RECHNERLISTE ↦ ZIELRECHNER
   Aus der Liste der Rechner, die für die Ausführung in Frage kommen, wird ein geeigneter ausgewählt. Gesichtspunkte können sein:
   - Auslastung des Rechners,
   - Ausführungszeit (geschätzt, statistische Mittel etc.),
   - Zuverlässigkeit,
   - Kosten für die Kommunikation,
   - Datenverteilung (sofern bekannt),
   - und weitere.

   Beispiel:
   Es wird der Rechner $Node_1$ ausgewählt, da die Auftragserteilung dorthin am schnellsten erfolgt.

(c) Zielrechner $\mapsto$ ausführender SS
Sind auf dem Zielrechner mehrere „Server“ vorhanden die den gewünschten Dinest anbieten, muß ein SS für die Ausführung bestimmt werden. Kriterien sind u.A.:
- Auslastung der SS,
- Ausführungszeit (geschätzt, statistische Mittel etc.),
- Datenverteilung (sofern bekannt),
- und weitere.

Beispiel:
Auf Knoten $Node_1$ sind Flugverbindungsdienste von *Lufthansa* und *Air France* vorhanden. Ausgewählt wird der Dienst von *Air France*, da der *Lufthansa*–„Server“ zur Zeit eine große Anzahl nicht bearbeiteter Aufträge besitzt.

6. *PM* $\leftrightarrow$ *SS*: `step_computation_request`
Der *ConTract Processing Monitor* beauftragt den ausgewählten SS mit der Ausführung des Steps. Ergebnis ist der Code der Step–Ausführung.

7. *PM* $\rightarrow$ *CM*: `step_finished`
Der PM meldet die Beendigung an den CM weiter.

8. *CM* $\leftrightarrow$ *TM*: `commit_work` oder `rollback_work`
Nach Abschluß aller Arbeiten[1] wird die Transaktion bestätigt oder abgebrochen. Hierzu sendet der CM dem TM eine entsprechende Nachricht.

- *TM :* Commit–Protokoll
Der TM führt mit allen an der Transaktion beteiligten RM ein 2-Phasen-Commit–Protokoll aus. Näheres im Abschnitt 3.3 ab Seite 32.

- Der Endzustand der Transaktion wird dem CM mitgeteilt (Ergebnis des Aufrufs aus Schritt 8).

- Der Zyklus von 2 bis 8 wird für jeden Step durchlaufen, bis der ConTract abgearbeitet ist.

9. *CM* $\rightarrow$ *CC*: `contract_terminated`
Die Beendigung, erfolgreich oder nicht, wird dem *ConTract Client* und damit dem Benutzer mitgeteilt.

3.2 Grobablauf eines ConTracts mit entfernter Step-Ausführung

Die Unterschiede des Standardablaufs im entfernten Fall sollen hier dargestellt werden (Abbildung 3.2). Deshalb wird nicht der gesamte Ablauf, sondern nur die Teile die sich verändert haben beschrieben.

Der Ablauf ergibt sich wie folgt:

[1] Die Transaktion kann sich über mehrere Steps erstrecken.

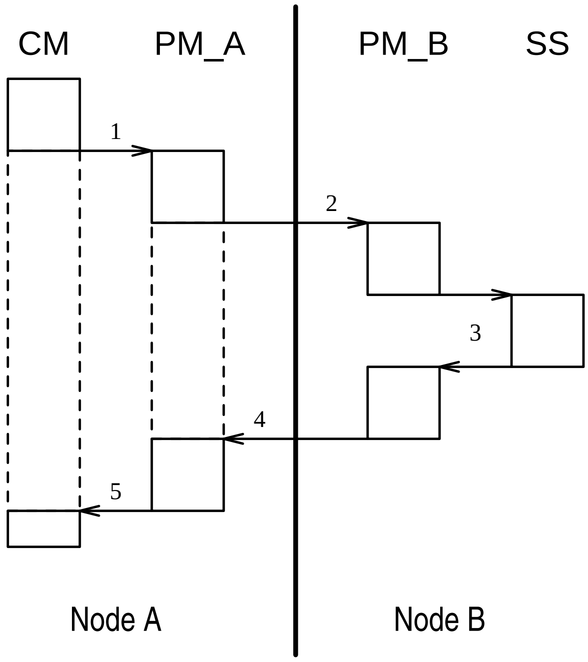

Abbildung 3.2: Grobablauf mit entfernter Step-Ausführung

1. $CM \rightarrow PM_A$: `schedule_steps`
   Der *ConTract Manager* übergibt die ausführbaren Steps dem PM. Dieser bestimmt wieder die in Frage kommenden *Step Computation Server*. Der Step soll auf einem anderen Rechner bearbeitet werden.

2. $PM_A \rightarrow PM_B$: `remote_request`
   Der PM_A beauftragt den PM_B des entfernten Rechners mit der Ausführung des Steps.

3. $PM_B \leftrightarrow SS$: `step_computation_request`
   Der entfernte PM_B wählt eine der vorhandenen lokale Verarbeitungseinheiten aus (lokale Lastbalancierung) und übergibt ihr den Auftrag. Nach Abarbeitung erhält der PM_B den Zustand der Ausführung zurück.

- Das Ergebnis der Step–Ausführung von wird dem aufrufenden PM_B zurückgeliefert.

4. $PM_B \rightarrow PM_A$: `remote_request_finished`
   Der entfernte *ConTract Processing Monitor* (PM_B) meldet dem Auftraggeber (PM_A) die Beendigung des Steps.

5. $PM_A \rightarrow CM$: `step_finished`
   Der PM_A gibt die Endemeldung wieder an den CM weiter.

3.3 Commit–Protokoll

Zur Bearbeitung von Transaktion gehört noch das Commit–Protokoll. In dieser Ausarbeitung soll das Protokoll im lokalen Fall beschrieben werden. Wie das genaue Protokoll im entfernten Fall aussieht war bei Beendigung dieser Arbeit noch nicht bekannt. Es ist Bestandteil von [17]. Angedacht war eine Lösung bei der Informationen aus dem Transaktions–Beziehungs–Netz bei Ausbreitung der Transaktion auf einen anderen Rechner diesem mitgeliefert werden.

Aus Sicht des Communication System bringt das genaue Commit–Protokoll (im entfernten Fall) keine weiteren Erkenntnisse. Es werden nur zusätzliche Aufrufe benötigt.

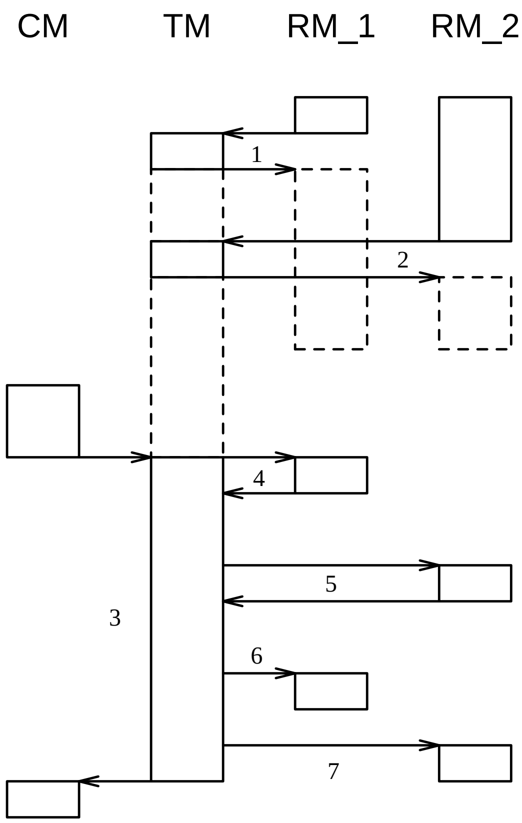

Abbildung 3.3: Commit–Protkoll mit zwei lokalen RM

Zur Illustration des Commit-Protokolls dient Abbildung 3.3. Es zeigt dabei die Verwendung von zwei RM.

- *SS*:
  Während der Abarbeitung des Steps benutzt der SS (u.U. mehrere) RM. Er ruft diese

über ihre „native" Schnitttselle, ohne direkte Nutzung des Communication System, auf[2].

1. *$RM_1 \leftrightarrow TM$*: `join_ta`
   RM_1 meldet dem TM, daß er an der Transaktion partipiziert.

2. *$RM_2 \leftrightarrow TM$*: `join_ta`
   Auch der zweite *Resource Manager* RM_2 meldet seine Teilnahme dem zuständigen TM.

3. *$CM \leftrightarrow TM$*: `commit_work` oder `rollback_work`
   Nach Abarbeitung des Steps erhält der CM die entsprechende Endemeldung und entscheidet über den Abschluß der Transaktion (`commit_work` oder `rollback_work` Nachricht an den TM).

- *TM :* COMMIT–PROTOKOLL
  Der TM führt mit allen an der Transaktion beteiligten RM ein 2-Phasen-Commit–Protokoll aus. In Phase 1 fragt er alle RM ob sie ein „Prepare-to-Commit" ausführen können. Ergebnis der Aufrufe sind die Commit–Entscheidungen der einzelnen RM.

4. *$TM \leftrightarrow RM_1$*: `precom_ta`

5. *$TM \leftrightarrow RM_2$*: `precom_ta`

- Nach Eingang der Votierungen der einzelnen RM entscheidet der TM über die Bestätigung oder den Abbruch der Transaktion. Die Entscheidung wird den beteiligten RM mitgeteilt (Phase 2 des Protokolls).

6. *$TM \rightarrow RM_1$*: `commit_ta` oder `abort_ta`

7. *$TM \rightarrow RM_2$*: `commit_ta` oder `abort_ta`

- Der Endzustand der Transaktion wird dem CM mitgeteilt (Ergebnis des Aufrufs aus Schritt 3).

  Bemerkungen:

  - Sollte der CM dem TM eine `rollback_work` Nachricht gesendet haben erübrigt sich ein 2-Phasen-Protokoll. Der TM benachrichtigt die beteiligten RM direkt mit einer `abort_ta` Nachricht (vgl. Schritt 6 bis 7).
  - Sollte der Commit–Wunsch des CM nicht positiv beantwortet werden, kann, je nach Kontrollflußspezifikation im Skript, eine Kompensation erforderlich werden.

3.4 Migration von ConTracts

Auf Wunsch des Benutzers kann ein ConTract von einem Rechner auf einen anderen migrieren. D.h., die Zuständigkeit und damit der Verarbeitungszustand muß an einen anderen *ConTract Manager* übergeben werden. Im APRICOT System muß zwischen zwei

[2] Die „Libraries" eines DBMS werden zum Objektcode des SS hinzugebunden.

Arten von Migration (`soft` und `hard`) unterschieden werden. Ausgelöst wird die Migration durch die Eingabe des entsprechenden Befehls durch den Benuzter, die der CC in eine `migrate_contract`–Nachricht an den CM umwandelt. Ein Parameter dieses Aufrufs bestimmt den Modus (`soft` oder `hard`).

3.4.1 Soft–Migration

Modus `soft` bedeutet, daß die gerade in Ausführung befindlichen Steps nicht abgebrochen werden. Vielmehr wird auf ihre Terminierung gewartet und keine neuen Steps mehr gestartet (bzgl. des ConTracts). Erst wenn alle aktuellen Steps beendet sind, veranlasst der CM die nötigen Aktivitäten um den ConTract zu migrieren. Dies sind:

1. *CM* $\rightarrow$ *PM*: `transfer_contract`
   Mit dieser Nachricht beauftragt der CM den PM den mitgelieferten Verarbeitungszustand dem neuen CM zu übermitteln.

2. Gleichzeitig vermerkt der PM die Migration, um zu gewährleisten, daß zu einem ConTract immer der verantwortliche *ConTract Manager* ermittelt werden kann. Diese Wissen muß global vorhanden sein. Deshalb wird der ConTract–Zustand über PM und nicht direkt von CM an CM weitergeleitet.

3.4.2 Hard–Migration

Im Gegensatz zur Soft–Migration, werden bei der Hard–Migration die gerade in Aufführung befindlichen Steps abgebrochen. Den zugehörigen Ablauf und die dabei auftretenden Fälle soll Abbildung 3.4 verdeutlichen.

1. *CC* $\rightarrow$ CM_A: `migrate_contract`
   Der Benutzer äußert gegenüber dem CC den Wunsch den ConTract auf einen anderen Rechner zu migrieren. Dies soll im Modus `hard` geschehen.

2. CM_A $\rightarrow$ *TM*: `abort_gtrid_list`
   Der *ConTract Manager* sendet dem *Transaction Manager* eine Liste mit den Transaktionen (Steps) die abgebrochen werden müssen. Der Modus `hard` hat zur Folge, daß die gerade ablaufenden Aktionen auf jeden Fall scheitern müssen, bzw. ihre Änderungen nicht permanent werden.

- Der CM wartet auf die `step_finished` Meldungen der abgebrochenen Steps.

3. *TM* $\rightarrow$ *SS*: `abort_ta`
   Die *Transaction Manager* benachrichtigen die betroffenen SS die gewünschten Transaktionen abzubrechen.

4. *TM* $\rightarrow$ *RM*: `abort_ta`
   Auch alle beteiligten RM werden aufgefordert die Transaktionen zu beenden.

- Der SS meldet das negative Ergebnis der Step–Ausführung dem PM (Antwort auf den `step_computation_request` Aufruf).

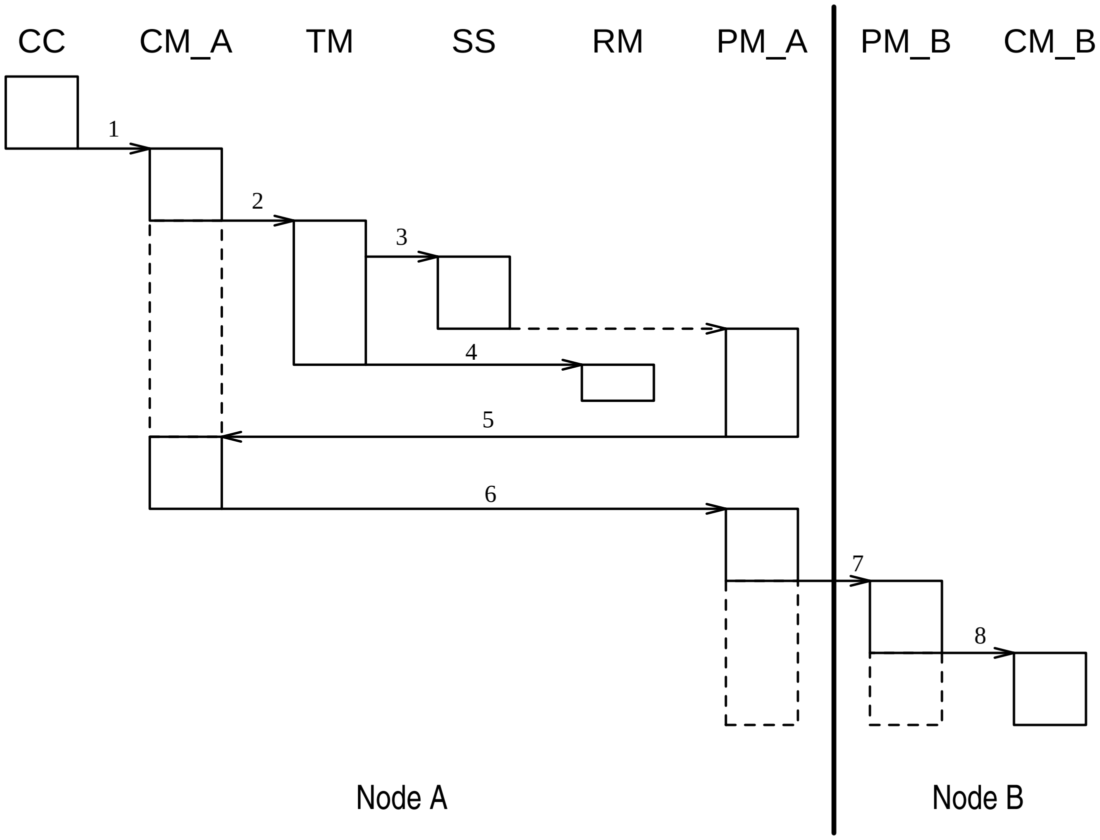

Abbildung 3.4: Ablauf für die Migration im Modus `hard`

5. $PM_A \rightarrow CM_A$: `step_finished`
   Das Ergebnis der Stepausführung geht an den CM.

6. $CM_A \rightarrow PM_A$: `migrate_contract`
   Der CM sendet, nachdem alle Endemeldungen eingegangen sind, dem PM_A die notwendigen Daten zur Zustandsbeschreibung des ConTracts und den neuen zuständigen *ConTract Manager* (CM_B).

7. $PM_A \leftrightarrow PM_B$: `transfer_contract`
   Der PM_A übermittelt die Daten an den neuen zuständigen PM_B.

   Die Information welcher CM für welchen ConTract verantworlich ist muß global bekannt sein. Aus diesem Grund erfolgt die Kommunikation über den PM.

8. $PM_B \leftrightarrow CM_B$: `contract_migration`
   Der PM leitet den Migrations-Auftrag an den neuen CM weiter.

3.5 Beobachtung aus den Abläufen

Anhand obiger Abläufe lassen sich für das Communication System einige Punkte erkennen:

- Die überwiegende Anzahl der Kommunikationswünsche bezieht sich auf lokale Kommunikation.
- Es findet viel Kommunikation zwischen den Systemkomponenten statt. Anwendungskomponenten erhalten in der Regel einen Auftrag und melden die Beendigung der Abarbeitung.
- Die Verwaltung der Transaktion erfordert den meisten Kommunikationsaufwand. Für die (lokale) Ausführung eines Steps sind 3 Nachrichten erforderlich:
    1. `schedule_steps`
    2. `step_computation_request`
    3. `step_finished`

  Zur Verwaltung der Transaktion sind es mindestes 7 (8) Nachrichten:
    1. `get_gtrid_list`
    2. `create_dependency`
    3. `join_ta`
    4. `commit_work` oder `rollback_work`
    5. `precom_ta`
    6. `commit_ta` oder `abort_ta`
    7. und ggf. `define_ta_event`
- Die Nachrichteninhalte sind bei den meisten Aufrufen klein (Nutzinformation kleiner 30 Byte).
    - `activate_contract`
      Name des ConTracts.
    - `create_dependency`
      Zwei gtrid's und Art der Abhängigkeit.
    - `join_ta`, `precom_ta`, commit_ta, ...
      Transaktionskennung (gtrid).
    - ...

Kapitel 4

Klassifikation der Kommunikation

4.1 Kriterien

Welche Gesichtspunkte klassifizieren die Kommunikation?
Welche Punkte sind im APRICOT *System von Bedeutung?*

Nach folgenden Kriterien soll der Kommunikationsbedarf der APRICOT–Komponenten untersucht werden:

- KOMMUNIKATIONSPARTNER
  Welche Komponente ist Empfänger einer Nachricht?

- TRANSPARENZ
  Ist die Kommunikation transparent?
  Wissen die Partner mit wem (mit welchem Prozeß, etc.) auf welchem Knoten sie kommunizieren?

- AUSFÜHRUNG
  Welche Arten der Ausführung existieren und wie sind sie zu unterscheiden?

- KOMMUNIKATIONSINITIATIVE
  Wer initiiert einen Informationsaustausch?

- KONTEXT
  Ist die Kommunikation mit Kontextinformation verbunden?

- ÜBERLAPPENDE AUFTRAGSERTEILUNG
  Kann oder muß eine Auftragserteilung mit anderen überlappend erfolgen?

- ERGEBNIS-/PARAMETERDARSTELLUNG
  Wie werden die Parameter bzw. das Ergebnis eines Aufruf/einer Nachricht repräsentiert?

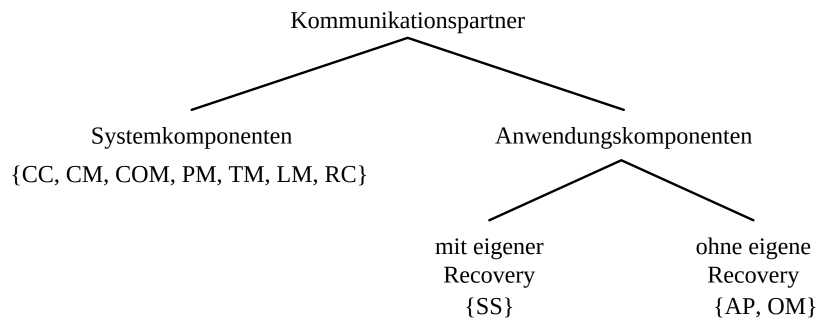

Abbildung 4.1: Klassifikation anhand des Partners

4.1.1 Kommunikationspartner

Wenn zwei Komponenten miteinander kommunizieren ist zu unterschieden ob Basiskomponenten (CM, TM, PM, CC, COM, LM ,RC) oder Anwendungskomponenten (SS, OM, AP) miteinander reden (siehe Abb. 4.1). Basiskomponenten müssen immer eigene Recovery–Fähigkeiten besitzen. Von den Anwendungskomponenten (SS, OM, AP) wird dies nicht unbedingt erwartet. Einzig ein *Step Computation Server* muß „recoverable“ sein. Erreicht er dies nicht durch Nutzung anderer RM (z.B. DBMS), muß er die Fähigkeit selbst implementieren.

4.1.2 Transparenz

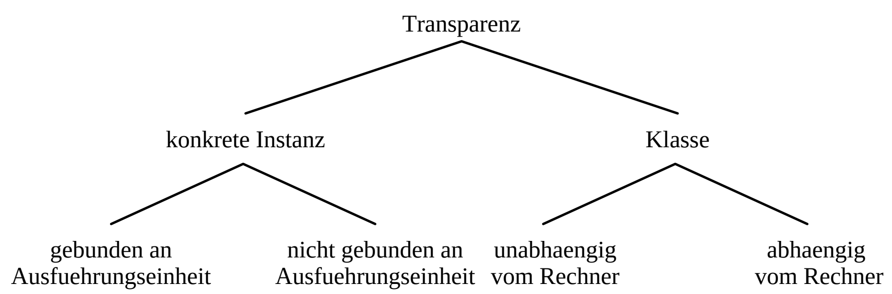

Abbildung 4.2: Unterscheidung durch die Transparenz

Abbildung 4.2 zeigt die Kriterien der Unterscheidung im Hinblick auf die Transparenz deri Adressierung. Erfolgt die Kommunikation mit einer konkreten Instanz oder mit einer Klasse? Der Erteilung eines Auftrages des PM an einen SS (`step_computation_request`)

ist Kommunikation mit einer konkreten Instanz. Der `join_ta`–Aufruf eines RM an den TM ist dagegen Kommunikation mit einer Klasse, da es dem RM egal ist welcher TM seinen Aufruf beantwortet.
Aufgrund der Systemarchitektur ist der Aufruf an den TM „rechnerabhängig“, denn der Empfänger ist immer auf dem lokal System. Der Auftrag an den SS ist „gebunden an eine Ausführungseinheit“, den speziellen *Step Computation Server*.

4.1.3 Ausführung

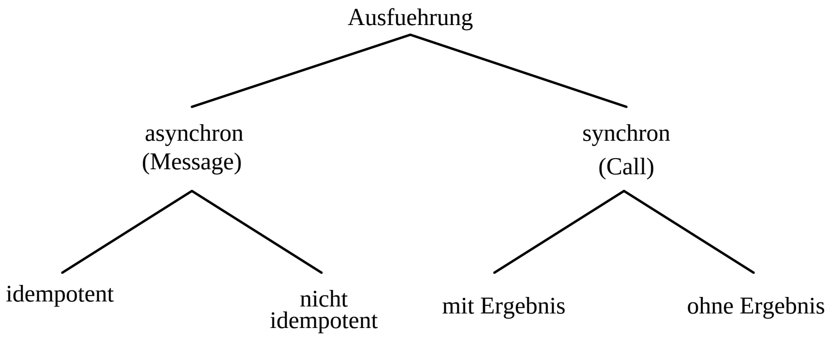

Abbildung 4.3: Art der Ausführung als Unterscheidungsmerkmal

Nach der Art der Ausführung muß ebenfalls unterschieden werden (Abb. 4.3). Wichtigstes Merkmal dabei ist die Trennung nach asynchroner „Message“ und synchronem „Call“. Nachrichten können idempotent sein, etwa der `abort_ta`–Aufruf des TM an einen RM. Dieser Aufruf kann beliebig oft wiederholt werden, ohne daß dies zu Inkonsistenzen führen würde.

Synchrone Call's können ein Ergebnis liefern. „Ohne Ergebnis“ bedeutet, daß die Rückkehr als Bestätigung (Acknowledge) ausreicht.

4.1.4 Kommunikationsinitiative

Abbildung 4.4 zeigt wie die Rolle der Kommunikationsparnter bzgl. der Initiative der Kommunikation festgelegt sein kann.

Als Beispiel für Komponenten, zwischen dennen die Initiative der Kommunikation nicht festgelegt ist, sind der *ConTract Manager* und der *ConTract Processing Monitor*. Nachrichtenaustausch kann von beiden Komponenten jederzeit angestoßen werden. (Bsp.: CM sendet dem PM `abort_gtrid_list` und gleichzeitig der PM dem CM `step_finished`). Der gleichzeitige Kommunikationsversuch kann, wenn dieser Gesichtspunkt nicht berücksichtigt wurde, zu einer Verklemmung führen, da beider Komponenten nach Anstoß der Nachrichtenaustausch auf die Entgegennahme durch den jeweils anderen warten.

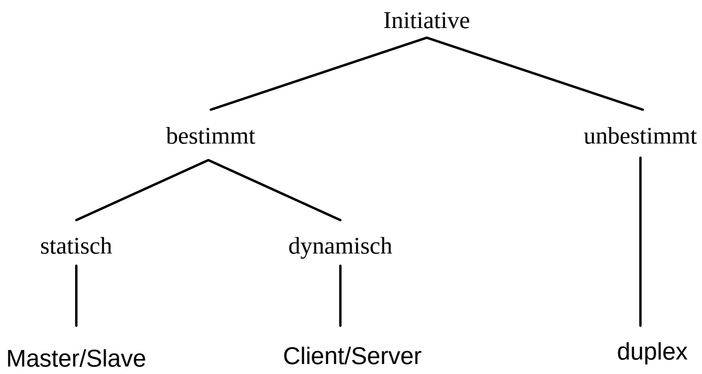

Abbildung 4.4: Klassifikation durch Unterscheidung der Initiative

4.1.5 Kontext

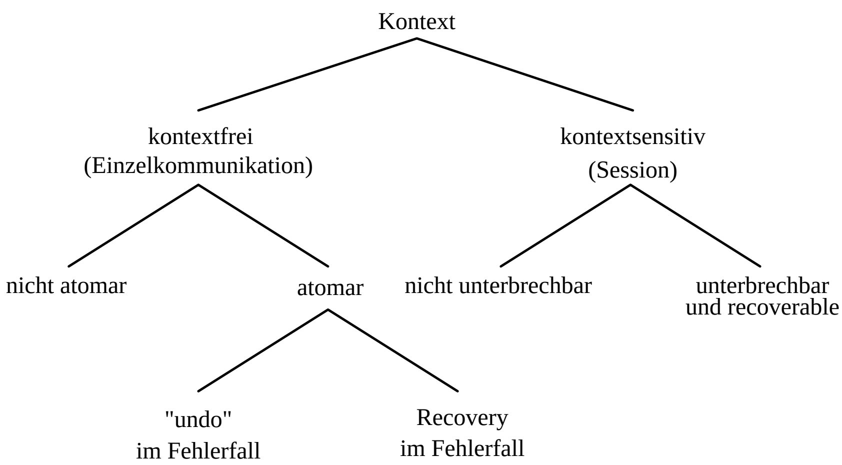

Abbildung 4.5: Kontext als Kriterium

Kommunikation kann kontextabhängig sein (siehe Abbildung 4.5). Kontextsensitiv ist Kommunikation, die über mehrere Schritte geht. D.h., die Partner tauschen mehrfach Nachrichten aus. Beispiel ist die Kommunikation des CM mit den PM . Sie geht über mindestens zwei Schritte (`schedule_steps` und `step_finished`). Sie ist unterbrechbar, denn sowohl CM als auch PM können ausfallen und nach dem Wiederanlauf weiterarbeiten, ohne daß der andere davon erfahren muß. Die Kommunikation zwischen OM und PM ist dagegen als atomar zu bezeichnen. Denn fällt der OM aus, muß der PM dies erfahren, um einen Ersatz einplannen zu können.

Einzelkommunikation kann atomar sein oder nicht. Atomar bedeutet, daß der Empfänger die Nachricht entgegenenhmen (quittieren) muß. Beispiel für nicht atomare Kommunikation ist das Versenden einer Nachricht,ohne daß bekannt ist, ob der Empfänger gerade erreichbar. Dies entspricht dem Versenden von Post. Wenn der CM dem CC eine `contract_terminated`–Meldung zukommen läßt, müßte der CC (bzw. der Benutzer) nicht vorhanden sein. Es reicht aus, wenn die „Post“ beim nächsten anmelden des Benutzers „zugestellt“ wird.

4.1.6 Überlappende Auftragserteilung

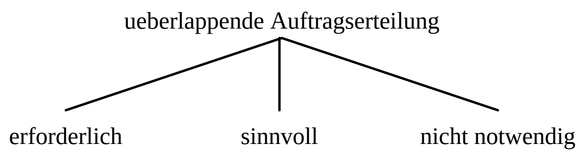

Abbildung 4.6: Überlappende Auftragserteilung

Die Erteilung von Aufträgen kann überlappend erfolgen oder nicht. Im APRICOT System muß dies möglich sein (PM) und sollte es möglich sein (TM), um einen höheren Parallelitätsgrad zu erreichen (siehe Abbildung 4.6).

- PM
  Der PM muß gleichzeitig mehrere *Step Computation Server* beauftragen und auf das Ergebnis der Ausführung warten (`step_computation_request`–Aufruf).
- TM
  Bei Abwicklung des Commit-Protokolls ist es sinnvoll die `Prepare-to-Commit` Anfragen an die beteiligten RM gleichzeitig abzusetzen und auf die Votierungen zu warten. Die sequentielle Ausführung der einzelnen (synchronen) Aufrufe ist ebenfalls logisch richtig aber in der Gesamtausführungszeit langsamer als die parallele Ausführung.

4.1.7 Parameter-/Ergebnisdarstellung

Für die Realisierung eines Communication System, insbesondere in einer heterogenen Rechnerumgebung, ist von Bedeutung, ob der strukturelle Aufbau der Aufrufparameter bzw. der Rückgabewerte statisch ist oder ob die Struktur erst zur Laufzeit bekannt gemacht wird (siehe Abbildung 4.7). Ähnliche Gesichtspunkte ergeben sich beim Austausch komplexer Strukturen wie Listen. Beim Einsatz in verteilten Umgebungen ist darauf zu achten, ob der Datenautausch ohne Schwierigkeiten zwischen den Rechnern der verschiedenen Hersteller (genauer: unterschiedlicher Prozessorarchitekturen) möglich ist. Sollte dies nicht der Fall

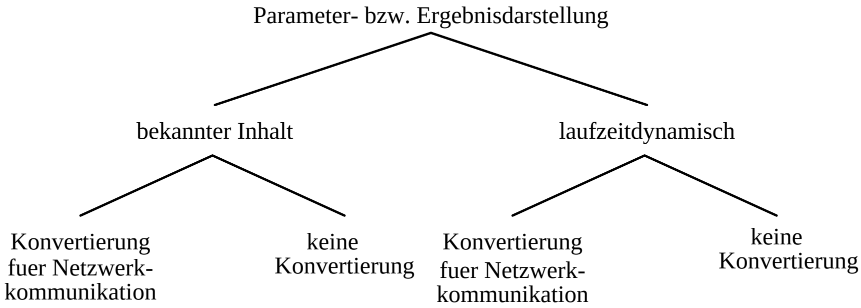

Abbildung 4.7: Darstellung des Ergebnisses bzw. der Parameter

sein, muß eine konvertible Darstellung verwendet werden, oder beim Übertragen zuerst eine Konvertierung stattfinden. Ein Protokoll, daß dieses Problem lößt ist das XDR–Protokoll von SUN Microsystems [18].

4.2 Kommunikationsklassen

Welche Kombinationen von Eigenschaften existieren im APRICOT *System?*

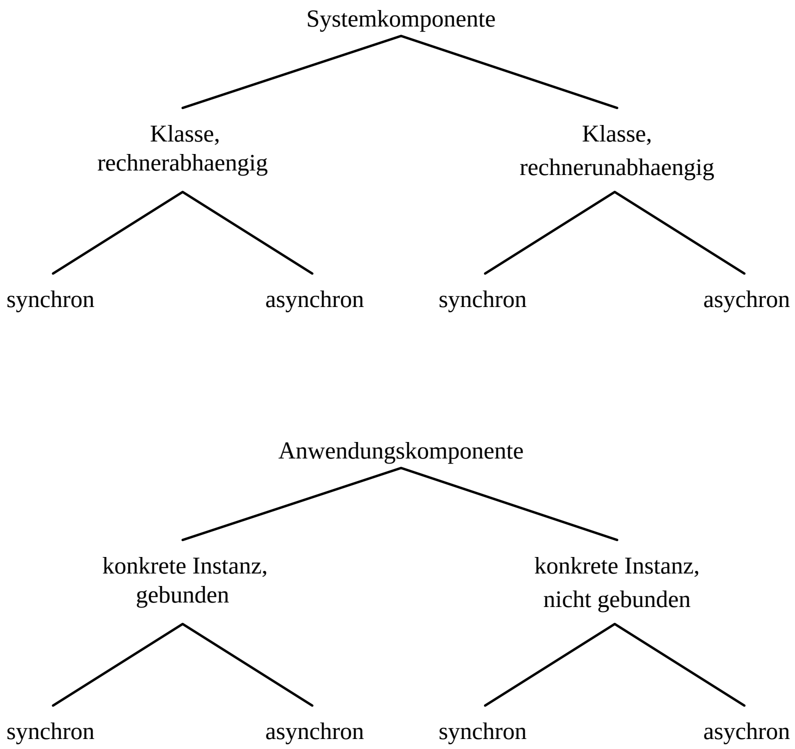

Abbildung 4.8: Kommunikationsklassen im APRICOT System

Abbildung 4.8 zeigt die zwei Hauptgruppen nach denen Kommunikation im APRICOT System unterschieden werden kann. Nicht alle Aufrufe fallen in diese zwei Gruppen, sie stellen nur den Schwerpunkt dar.

1. KOMMUNIKATIONSPARTNER
   Alle Kommunikation mit den Systemkomponenten stellt immer Kommunikation mit einer Klasse dar, es interessiert niemals die konkrete Instanz (z.B. der konkrete TM oder der spezielle CM). Die Anwendungskomponenten werden hingegen immer als konkrete Instanz angesprochen (der eine SS oder die spezielle *Applikation*). Weiter Besonderheiten sind:

   - Bei der Kommunikation mit einem SS ist nicht unbegdingt eine Bindung an die Ausführungseinheit (Prozeß) notwendig. Ist der SS „recoverable“ so kann die Auftragserteilung an Prozeß n gehen, während Prozeß m die Endemeldung absetzt. Das System (oder besser der PM) behandelt aber Prozeß n und den Nachfolgerprozeß m als einen speziellen SS[1].

- Die Kommunikation mit den Komponenten LM und RC ist rechnerunabhängig. Bei ihr kann davon ausgegangen werden, daß diese Komponenten im Netzwerk verteilt sind. Kommunikation mit PM, CM, TM ist aufgrund der Systemarchitektur immer lokal[2].

2. AUSFÜHRUNG
Zur Ausführung der Aufrufe der einzelnen Gruppen ist sowohl synchrone als auch asynchrone Kommunikation erforderlich. Die überwiegende Anzahl von Aufrufen ist nicht idempotent (einzige Ausnahme `abort_ta`–Nachricht) und die Ergebnis-/Paramterdarstellung ist bis auf wenige Ausnahmen (Listenfunktionen wie `get_gtrid_list` oder `abort_gtrid_list` und der `transfer_contract`–Funktionen) fest und bekannt (zur Übersetzungszeit der Komponenten).

Die Unterscheidungsmerkmale *Initiative* und *Kontext* besitzen im APRICOT System nur untergeordnete Bedeutung:

- INITIATIVE
Die Kommunikationsinitiative bei Abarbeitung eines ConTracts ist meist als „Client/Server"–Kommunikation zu klassifizieren, wobei die Rollenverteilung nicht fest ist. Beispiel ist die Kombination PM, SS. Der SS wartet auf Aufträge, bearbeitet diese und liefert das Ergebnis zurück. Zur Ausführung der Aufträge wendet er sich an dem TM. Bei dieser Kombination ist der SS der Auftraggeber und der TM der Empfänger.

  Die Rollenverteilung zwischen CM und PM ist allerdings unbestimmt. Der PM kann dem CM eine `step_finished` Nachricht senden wähend der CM zur selben Zeit eine `schedule_steps` Nachricht übermitteln will (Verklemmungsgefahr).

- ÜBERLAPPENDE AUFTRAGSERTEILUNG
Für die Arbeit des PM und des TM ist bei Abarbeitung nur eines ConTracts bereits eine überlappende Auftragserteilung erforderlich (PM: mehrere `step_computation_request`–Nachrichten zur gleichen Zeit) bzw. sinnvoll (TM: `precom_ta`–Aufruf an alle beteiligten RM parallel).Erweitert man das Blickfeld auf die gleichzeitige Ausführung mehrerer ConTracts, so stellt man fest, daß überlappende Kommunikation der Komponenten untereinander möglich sein muß (ein ConTract pro APRICOT System ist eindeutig zu wenig).

- KONTEXT
In verteilten Systemen kann eine Komponente zwischenzeitlich nicht verfügbar sein. D.h., im Falle der kontextfreien Kommunikation (Message) interessiert den Absender einer Nachricht nicht unbedingt ob sein Gegenüber gerade ansprechbar ist oder nicht. Es reicht aus, wenn die Nachricht zugestellt wird.

[1] Dieses Verfahren ist z.Zt. nicht vorgesehen, wäre aber denkbar.

[2] Nicht lokale Kommunikation findet im APRICOT System nur zwischen PM–PM und zwischen TM–TM statt.

Ähnliche Überlegungen gellten für „kontextsensitive“ Kommunikation. Sie kann nur „unterbechbar und recoverbale“sein.

Kapitel 5

Das Communication System

5.1 Anforderungen an das Kommunikationssystem

Welche Anforderungen haben die APRICOT *Komponenten an das CS?*
Welche Besonderheiten ergeben sich aus der Systemarchitektur und der ConTract*–Abwicklung?*

- TRANSPARENZ
  Das APRICOT System ist zum Einsatz in verteilten Rechnersystemen konzipiert. Deshalb wird eine transparente Kommunikation gefordert, d.h., Anwendungen sollen nicht wissen, auf welchem Rechner ihre Kommunikationspartner sind.
  Beispiel:

  > Aufrufe an den *Global Log Manager*. Der Aufrufer will nicht wissen, wo der LM ansässig ist.

- KLASSENKOMMUNIKATION
  Die Kommunikation erfolgt meistens nicht mit einer konkreten Instanz, sondern mit Klassen (Kommunikation mit Systemkomponenten). Hinzu kommt, daß zwei Aufrufe an eine Dienstklasse nicht zwangsweise an den gleichen Prozeß gehen (Kontextunabhängigkeit).
  Beispiel:

  > Der `get_gtrid_list`–Aufruf des *ConTract Manager* und der anschliessende `create_dependency` Call müssen nicht an den gleichen TM oder an denselben Prozeß erfolgen. Entscheidend ist, daß eine konsistente Zustandsbeschreibung der Abhängigkeiten zwischen den Transaktionen vorhanden ist.

- ABLÄUFE BEI DER ConTract–ABWICKLUNG
  Die Abläufe im System sind vorgegeben. Daraus ergeben sich zwei Besonderheiten bezüglich der Schnittstelle des Communication System zu den Anwendungen (Komponenten):

1. *Bekannte Funktionalität*
   Art und Umfang der Funktionalität der Kommunikationswünsche ist bekannt. Durch die eindeutige Definition der Schnittstellen kann der Anwendung bzw. dem Programmierer eine eindeutig definierte Menge von Kommunikationsfunktionen zur Verfügung gestellt werden (analog den `gtrid_` und `xa_` Funktionen in der X/Open DTP Umgebung [32]).
   Beispiel:

   `join_ta`, `abort_ta`, etc. sind Funktionen zum Umgang mit Transaktionen.

2. *Implizite Adressierung*
   Ein weiterer Vorteil der eindeutig definierten Schnittstelle der Komponenten untereinander und der Systemarchitektur besteht in der Vereinfachung der Adressierung. Aus der Art des Aufrufs ergibt sich der Empfänger, und aufgrund der Architektur ist bekannt, ob der Empfänger lokal oder entfernt ist. Für viele Kommunikationsaufrufe ist die explizite Angabe des Empfängers nicht notwendig.
   Beispiel:

   Empfänger des `join_ta`–Aufrufs ist immer der (lokale) TM.
   `schedule_steps` geht an den PM.

- VERFÜGBARKEIT
  Die Verarbeitung soll auch im Fehlerfall (Ausfall eines Rechnerknotens, Ausfall einer einzelnen Komponente usw.) soweit wie möglich fortgesetzt werden. Daraus folgt, daß, wenn der Kommunkationswunsch es zuläßt, eine direkte Verbindung zwischen zwei Ausführungseinheiten nur dann eingegangen werden soll, wenn durch eine Ausfall keine unnötigen Auswirkungen auf den Sender entstehen.
  Beispiel:

  Die `remote_request_finished` Nachricht geht an einen entfernten PM. Dieser kann zur Zeit nicht erreichbar sein. Den PM braucht dies aber nicht zu interessieren. Es genügt ihm, wenn nach Wiederverfügbarkeit, die Nachricht zugestellt wird. Deshalb sollte die zwischenzeitliche Unerreichbarkeit des entfernten PM keine zusätzliche Aktionen des lokalen PM erfordern (z.B.: PM muß die Nachricht stabil speichern und selbst dafür sorgen, daß nach einiger Zeit ein neuer Versuch unternommen wird die Nachricht zu versenden).

- ROBUSTHEIT
  Kommunizierern zwei Anwendungen wiederholt miteinander und sollte zwischen zwei Aufrufen eine der beiden Komponenten kurzfristig nicht verfügbar sein, so muß dies keine Auswirkungen auf den Sender haben. Dabei ist zwischen Fehlern in der Kommunikationsverbindung und dem Ausfall einer der beiden Partner zu unterscheiden:

  – Verbindungsfehler
    Tritt kurzfristig ein Fehler in der Verbindung auf (ohne daß die Leitung benutzt wird), darf dies keine Auswirkung auf beide Partner haben. Sie können ohne Störung weiterarbeiten.

  - Ausfall eines Partner
    In diesem Fall muß unterschieden werden, ob der ausgefallene Partner eine Systemkomponente oder eine Anwendungskomponente ist. Eine Systemkomponente muß ihren Verarbeitungszustand stabil verwalten (z.B. mithilfe des LM). Nach Wiederanlauf hat sie den ursprünglichen Zustand wieder errreicht, der Ausfall darf keine Auswirkungen haben.

    Der Ausfall einer Anwendungskomponente unterscheidet sich von dem einer Systemkomponente. Anwendungskomponenten werden immer als konkrete Ausführungseinheit betrachtet. Ihr Ausfall hat Auswirkungen auf den Ablauf, denn es muß Ersatz beschafft werden (ein anderer SS oder ein neuer OM).

  Beispiel:

  > Auf einem entfernten Rechner weden Steps ausgeführt. Der Rechner ist aufgrund eines Fehlers im Netzwerk nicht mehr erreichbar. Diese Situation muß nicht zum Abbruch der Anwendung führen, es kann auf die Behebung des Fehler gewartet werden[1].

- Komunikationsinitiative
  Es muß möglich sein, daß zwei Komponenten gleichzeitig miteinander kommunizieren. Dies darf nicht zu einer Blockierung (Deadlock) führen. Ebenso muß der Nachrichtenversand möglich sein, wenn der Empfänger gerade mit einer dritten Komponente (synchron) kommuniziert.
  Beispiel:

  > CM sendet dem PM eine `schedule_steps` Nachricht und gleichzeitig der PM dem CM eine `step_finished`–Meldung.

- Art der Ausführung
  Die Kommunikation besteht aus synchronen Aufrufen und asynchronen Nachrichten. Logisch synchrone Aufrufe müssen aber nicht zwangsweise auch synchron ausgeführt werden. Bei Durchführung des 2–Phasen–Commit–Protokolls des TM mit den beteiligten RM (bzw. entfernten TM) fragt der TM der Reihe nach die einzelnen RM nach ihrer Votierung. Diese Aufrufe müssen nicht synchron ausgeführt werden. Sie sollten es auch nicht, denn es wird eine höherer Grad an Parallelität erreicht, wenn der TM alle Anfragen asynchron absetzten kann und auf die Ergebnisse der einzelnen RM wartet. Ein weiteres Problem bei der Art der Ausführung besteht darin, daß die Systemkomponenten mehrere Aufträge (ConTracts) bearbeiten. Während dem Warten auf das Ergebnis eines synchronen Aufrufes können und müssen andere Tätigkeiten ausgeführt werden. Das Communication System muß einen Mechanismus zur aynchronen Ausführung synchroner Aufrufe zur Verfügung stellen, um den Grad der Parallelität zu erhöhen und die Ausführungszeit zu reduzieren.
  Beispiel:

  > 2PC–Protkoll im Beispielablauf ab Seite 32.

[1] Sollten Zeitrestriktionen bestehen, kann auch ein Abbruch erfolgen und eine Ersatzkomponente eingeplant werden. In langen Vorgängen (z.B. mehrere Tage bei einem Fertigungs- oder Bürovorgang) ist das Warten auf die Wiederverfügbarkeit aber durchaus akzeptabel oder gar unumgänglich.

- NACHRICHTENREIHENFOLGE
  Beim Nachrichtenversand zwischen zwei Komponenten (auch Klassen) muß die Reihenfolge beim Senden der Reihenfolge beim Empfangen entsprechen. Das Communication System hat für die Einhaltung der Reihenfolge des Nachrichtenverkehrs zu sorgen.

- NACHRICHTENVERLUST
  Die Nutzer des Kommunikationsdienstes dürfen mit dem Problem des möglichen Verlustes einer Nachricht nicht konfrontiert werden. Dieses Problem ist durch das Communication System zu lösen.
  Beispiel:

  > Der Verlust einer `schedule_steps` Nachricht des CM an den PM, ohne da s der CM dies bemerkt, führt zum Stillstand des ConTracts.

- REALISIERUNGSUNABHÄNGIGKEIT
  In verteilten, goßen Netzen werden Rechner unterschiedlicher Hersteller und verschiedene Netzwerke eingesetzt. Der Kommunikationsmechanismus muß von technischen Aspekten und Implementierungseigenschaften wie unterschiedlichen Übertragungsprotokollen, Unzuverlässigkeit der Verbindungen, Geschwindigkeiten etc. abstrahieren. Gleiches gilt für die Unabhängigkeit von den verschiedenen Betriebssystemen.
  Beispiel:

  > Die Funktionalität des APRICOT System darf nicht davon abhängen, ob als Netz ein LAN oder ein WAN verwendet wird.

- DARSTELLUNG VON WERTEN
  Aufgrund der Systemarchitektur sind die meisten Aufrufe lokal, da die entsprechenden Komponenten lokal vorhanden sind. Für diese Aufrufe muß keine netzwerkgerechte Konvertierung der Datendarstellung erfolgen. Für die Aufrufe über das Netz gilt die Umkehrung. Im Communication System soll dies berücksichtigt sein.
  Beispiel:

  > Für alle Remote–Aufrufe muß ggf. eine Datenkonvertierung durchgeführt werden. Sind Kenntnisse über den Zielrechner bekannt, braucht eine Konvertierung nur bei Bedarf vorgenommen werden.

- UNTERSTÜTZUNG VON RECOVERY–MASSNAHMEN
  Von fast allen Komponenten des APRICOT System wird erwartet, daß sie Fähigkeiten zur Recovery besitzen. Das Communication System sollte diese Anforderung an die Komponenten unterstützen, um die Implementierung dieser zu vereinfachen.
  Beispiel für eine Realsierung:

  > Der Eingang eines Auftrags muß vermerkt werden. Liest eine Komponente einen Auftrag und fällt danach aus, muß sichergestellt werden, daß der Auftrag nicht verlorengeht. Ein Mechanismus könnte z.B. sicherstellen, daß eine Nachricht gelesen werden kann und die Annahme bestätigt werden muß. Dazwischen kann die Komponenten den Auftrag stabil vermerken.

- PERFORMANCE
  Die Kommunikation soll in der Leistungsfähigkeit (Zeit- und Resourcenbedarf) mit

ähnlichen Kommunikationsmechanismen existierender Betriebssysteme (RPC, Socket based IPC etc.) vergleichbar sein.

5.2 Alternative Konzepte

Welche Alternative Kommunikationskonzepte existieren und wie ist ihre Eignung für das APRICOT *System zu bewerten?*

Folgenden drei Komunikationskonzepte sollen auf ihre Eignung hin untersucht und bewertet werden:

1. Remote–Procedure–Call (RPC)
2. Punkt–zu–Punkt Verbindung
3. Mailbox Kommunikation

Im weiteren werden die wichtigsten Eigenschaften der Konzepte kurz vorgestellt. Die Bewertung anhand der im vorigen Abschnitt angeführten Kriterien erfolgt mithilfe einer Tabelle und zusätzlichen Erläuterungen.

5.2.1 Beschreibung der einzelnen Konzepte

5.2.1.1 Remote Procedure Call

RPCs dienen zur Realisierung eines Client-Server–Konzeptes. Das Senden des Auftrags und der Empfang des Ergebnisses erscheinen dem „Client" wie ein gewöhnlicher Funktionsaufruf in einem Programm. In beiden Fällen wartet der Aufrufer auf die Beendung und die Verfügbarkeit des Ergebnisses. Die eigentliche Ausführung des RPC erfolgt lokal oder auf einem anderen Rechner. Der Aufrufer braucht sich dessen aber nicht bewußt sein.

Im Unterschied zu normalen Funktionsaufrufen sind RPCs betroffen, wenn Nachrichten verlorengehen, der „Server" oder der „Client" ausfallen. Zur Definition der Semantik eines RPCs existieren deshalb folgende Möglichkeiten:

- GENAU EINMAL (EXACTLY ONCE)
  Der Aufruf wurde genau einmal durchgeführt.
- HÖCHSTENS EINMAL (AT MOST ONCE)
  Die Operation wurde höchstens einmal ausgeführt (einmal oder keinmal).
- MINDESTES EINMAL (AT LEAT ONCE)
  Wenn der Aufrufer die Kontrolle zurückerhält, weiß er, daß die Operation mindestens einmal ausgeführt wurde. Bei idempotenten Aufrufen ist dies eine geeignete Lösung.

5.2.1.2 Punkt–zu–Punkt Verbindung

Unter diesem Konzept wird hier folgendes Verstanden:

- DIREKTE VERBINDUNG
  Die Kommuniktion zwischen zwei Partnern erfolgt über eine direkte Verbindung der beiden Ausführungseinheiten (Prozesse).

- INDIREKTION DURCH PORTS
  Der Aufbau einer Verbindung erfolgt über Ports. Dadurch ist es möglich, daß eine Dienstklasse (repräsentiert durch einen Port) angesprochen wird und der Auftrag von einem Prozeß der Klasse beantwortet wird.

5.2.1.3 Mailbox

Der Mailbox–Mechanismus orientiert sich an den Gegebenheiten der „Brief–Post".

- ANWESENHEIT
  Der Empfänger muß nicht anwesend sein. Der Brief wird entweder in den Briefkasten eingeworfen oder der Briefträger kommt weitere Male vorbei.
- ASYNCHRONITÄT
  Nach dem Einwurf eines Briefes kann der Absender anderen Tätigkeiten nachgehen.
- QUITTIERUNG
  Der Absender kann eine Quittierung des Briefes verlangen (Einschreiben).
- PARALLELITÄT
  Es ist möglich, mehrere Briefe gleichzeitig anzunehmen und zu sichten bzw. mehrere Briefe gleichzeitig abzusenden. Dies kann von unterschiedlichen Personen (Prozessen) getan werden.

5.2.2 Gegenüberstellung der Konzepte

Als Vergleichskriterien werden die im vorhergehenden Abschnitt angeführten Kriterien verwendet. Tabelle 5.1 zeigt die Gegenüberstellung der drei Mechanismen. Folgende Beurteilungen ergeben sich:

1. RPC
   Ein RPC ist das am meisten verwendete Kommunikationskonzept. Für den Einsatz im APRICOT System ergeben sich aber einige Nachteile:
   - Auf konzeptueller Ebene sind nur synchrone Aufruf möglich. Die Nutzung asynchroner RPCs erfordert die Nutzung tieferer Ebenen der RPC–Realisierung. Im Betriebssystem SUN OS sind solche Möglichkeiten zwar vorgesehen ([18]), eine Nutzung verbietet sich aber aufgrund der Randbedingung, daß verschiedene Betriebssysteme zusammen genutzt werden sollen.
   - Die GENAU EINMAL Semantik eines Aufrufs hängt vom verwendeten Protokoll ab (z.B. TCP oder UDP).
   - Der Kommunikationspartner muß verfügbar sein.
   - Ein Vorteil des RPC ist, daß er auf den Systemenm, die ihn anbieten, sehr schnell ist.
2. PUNKT–ZU–PUNKT VERBINDUNG

- Im Gegensatz zu RPCs ist i.A. Asynchronität vorgesehen.
- Der Kommunikationspartner muß ebenfalls verfügbar sein.
- Analog zum RPC ist die Realisierung und Funktionalität abhängig vom Protokoll und vom Betriebssystem.

3. MAILBOX MECHANISMUS

   - Asynchronität stellt keine Schwiergkeit dar.
   - Vorteile sind, daß der Partner nicht verfügbar sein muß und die größere Robustheit gegenüber Fehlern.
   - Ein Problem der Mailbox ist die Ausführungszeit. Durch die (stabile) Verwaltung der Nachrichten entsteht zwangsweise ein Overhead.

Bemerkungen:

- Voraussetzung für die Implementierungen im Rahmen dieser Diplomarbeit war, daß das ConTract System auf UNIX–Rechnern verschiedener Hersteller (SUN, Digital Equipment) und ggf. auf einem Computer der Firma Tandem unter Guardian eingesetzt werden soll. Aufgrund der Abhänigigkeiten der Kommunikation vom Betriebssystem habe ich deshalb eine Mailbox Lösung bevorzugt (siehe auch folgendes Kapitel).

- Um die Leistungsfähigkeit des ConTract Systems, und damit des gesamten APRICOT Systems, nicht unnötig durch einen zeit- und resourcenvebrauchenden Kommunikationsmechanismus herabzusetzen, habe ich einen Mailbox–Mechanismus auf Basis von *Shared Memory* realisiert.

- Bei Beginn der Diplomarbeit wurde bzgl. der Architektur von einem Mehrprozeßsystem ausgegangen. Diese Prämisse betrifft viele der vorgestellten Überlegungen und die Realisierung. Eine Implementierung in einem Ein–Prozeß–System (bzw. mithilfe weniger Prozesse) ist angedacht. Hier werden sich die Gewichtungen einzelner Kriterien des Kommunikationsbedarfs ändern und ggf. andere Lösungen sinnvoller erscheinen lassen. Kriterien wie z.B. Überlappende Ausführung habe dort keine Bedeutung, da der entsprechende Code zu den einzelnen ConTract–Instanzen hinzugebunden ist.

Kriterium	RPC	Punkt–zu–Punkt	Mailbox
Transparente Adressierung	$\surd$	$\surd$	Ist, wenn nicht vorhanden, zu realisieren.
Klassenkommunikation	$\surd$	$\surd$ Nach einem Ausfall ist ein „rebind" erforderlich.	$\surd$ Nach einem Ausfall ist ein neuer Verbindungsaufbau notwendig.
Verfügbarkeit	— Partner muß verfügbar sein.	— Partner muß verfügbar sein.	$\surd$ Partner muß nicht verfügbar sein.
Robustheit Verbindungsfehler Ausfall	 $\surd$ Erkennung und Wiederaufbau der Verbindung erforderlich.	 $\surd$ Erkennung und Wiederaufbau der Verbindung erforderlich.	 $\surd$ $\surd$
Kommunikationsinitiative	—	$\surd$ Möglich, wenn Asynchronität vorgesehen ist.	$\surd$
Art der Ausführung	— Nur synchrone Aufrufe möglich.	$\surd$ Möglich, wenn Asynchronität vorgesehen ist.	$\surd$
Überlappende Auftragserteilung	—	$\surd$ Nur bei verschiedenen Empfängern (Verbindungen) möglich.	$\surd$
Reihenfolge	— Im Konzept nicht vorgesehen. Hängt von der Implementierung bzw. vom verwendeten Netzprotokoll ab.	— Abhängig von der Imple- mentierung bzw. vom Netzwerkprotokoll.	$\surd$ Ist vorhanden oder leicht zu realisieren, z.B. durch einen „timestamp".
Verlust	— Protokollabhängig	— Protokollabhängig	— Abhängig von der Implementierung.
Unabhängigkeit	Z.Zt. abhängig von Netzwerk und Betriebssystem.	Z.Zt. abhängig von Netzwerk und Betriebssystem.	Protokolle und Realisierung auch für tiefere Ebene vorhanden.
Darstellung von Werten	Herstellerabhängig bzw. eigene Realisierung erforderlich.	Herstellerabhängig bzw. eigene Realisierung erforderlich.	Herstellerabhängig bzw. eigene Realisierung erforderlich.
Recovery–Unterstützung	—	—	$\surd$ Ist zu realisieren.

Tabelle 5.1: Vergeich der Kommunikationsmechanismen

Kapitel 6

Realisierung des ConTract System im Prototypen

6.1 Allgemeine Bemerkungen

6.1.1 Lösungsansatz

Als Konzept für die Realiserung wurde eine Mailbox–Mechanismus gewählt. Kriterien für die Wahl waren:

- KLASSENKOMMUNIKATION
  Sie ist sehr einfach möglich, da jeder Klasse einen eigenen Briefkasten bekommt. Hiermit ist sowohl die eine Indirektion bzgl. den kommunizierenden Prozessen als auch der Zugriff mehrerer Instanzen einer Klasse auf eine Mailbox möglich.

- VERFÜGBARKEIT
  Die Mailboxen sind unabhängig von der Verfügbarkeit der zuständigen Komponenten ansprechbar. Dies erhöht die Parallelität der Abarbeitung (es muß nicht auf die Verfügbarkeit gewartet werden) und macht das System fehlertoleranter. Der letzte Punkt hat zudem Einfluß auf die Erstellung und Wartung der Programme, da im Code für die Komponenten weniger Fehlsituationen behandelt werden müssen.

- KOMMUNIKATIONSINITITIVE
  Die bereits öffters angesprochende Gefahr der Verklemmung wenn zwei Partner gleichzeitig versuchen zu Kommunizieren exisitert bei der Kommunkation über Briefkästen nicht.

- ART DER AUSFÜHRUNG
  Asynchrone und synchrone Kommunikation ist möglich. Synchrone Aufrufe können bei Bedarf auf zwei asynchrone Nachrichten (Anfrage und Antwort) reduziert werden (siehe auch Abschnitt 6.2.5 ab Seite 6.2.5).

Das allgemeine Kommunikationskonzept auf Anwendungsebene, das je nach den Mitteln des Betriebssystems entsprechend umgesetzt werden muß, soll die nötige Unabhängigkeit von den physischen Eigenschaften der Verbindung und dem Betriebssystem ermöglicht. Es läßt sich wie folgt charakterisieren:

- Das Konzept beinhaltet die Zusicherung der *sicheren Kommunikation*:

  > **Sichere Kommunikation** bedeutet, daß durch die Entgegennahme eines Kommunkationsauftrages durch das Communication System, welcher Art der Auftrag auch ist, dieser garantiert ausgeführt wird . Diese Zusicherung beinhaltet die garantierte Ausführung zu einem frühestmöglichen Zeitpunkt unter Einhaltung der Aufrufreihenfolge in einem undefinierten (zeitlich nicht begrenzten) Zeitraum. Bei Ausfall des Rechners oder des Netzwerkes wird der Kommunikationsauftrag nicht vergessen.

- Kommunikation ertfolgt in der Anwednungsebene über eine genau spezifierte Menge von Funktonen. In diesen ist, aufgrund der Semantik der einzelnen Aufrufen, zusätzliche Information wie Empfänger, Art der Ausführung sowie Anzahl und Typen der Aufruf- und Rückgabeparameter als Information enthalten.

- Benutzer der Kommunikation wissen, daß diese über Mailboxen stattfindet und jeder Komponentenklasse eine Mailbox zugeordnet ist[1]. Die Mailboxen sind unabhängig von der Verfügbarkeit der Ausführungseinheiten ansprechbar.

Für die Realisierung des Communication System unter dem Betriebssystem UNIX (SUN OS) gilt weiter:

- Die Briefkästen sind über *Shared Memory* realisiert. Die einzelnen Bereiche sind nach der Initialsierung Bestandteil des Adressraums der Komponenten(prozesse).

- Der Zugriff auf den gemeinsamen Mailbox–Speicherbereich wird durch Semaphore synchronisiert.

- Um einen konsisteten Zustand der Mailbox sicherzustellen, erfolgen die Manipulationen als Transaktion. Änderungen werden auf einer Kopie der Mailbox ausgeführt und erst nach erfolgreicher Beendigung freigegeben. Andernfalls könnte eine aufgrund eines Fehlers nicht komplett ausgführte Änderung einer Komponente die Fehlfunktion anderer Komponenten zur Folge haben.

Bemerkung:

- In dem Ansatz der Firma Digital Equipment ([2]) wurde eine „reliable Queue“–Mechanismus implementiert ([3]). Dieser Ansatz hat große Ähnlichkeit mit dem hier vorgestellten Mailbox–Prinzip.

6.1.2 Schichtenaufbau

Zur Abstraktion von den verwendeten Kommunikationshilfsmittel des Betriebssystems liegt es nahe, das von der ISO definierte OSI (Open Systems Interconnection) Reference Model zu berücksichtigen. Dieses Modell definiert die Gliederung einer Netzwerkarchitektur in sieben Schichten. Die einzelnen Schichten und deren Aufgaben bzw. Realisierung im APRICOT System verdeutlicht Abbildung 6.1.

[1] Eine Ausnahme bildet die Klasse der *Step Computation Server*. Jeder SS hat eine eigene Mailbox.

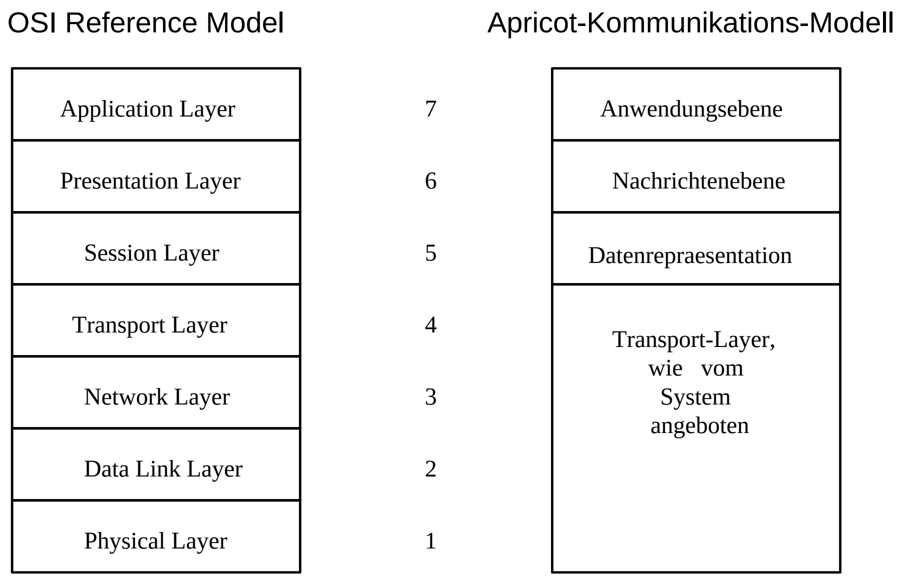

Abbildung 6.1: Gegenüberstellung der Schichten

Die Ebenen im einzelnen:

7 ANWENDUNGSEBENE
Dem Benutzer werden die benötigten synchronen und asynchronen Aufrufe als Funktionen zur Verfügung gestellt. Synchrone Aufrufe wirken blockierend, eine Fortsetzung ist erst beim Eintreffen der Rückantwort möglich. Asynchrone Aufrufe werden, nachdem sie vom Communication System entgegengenommen wurden, unter Systemgarantie beim Empfänger abgeliefert[2]. Der Nachrichtenversand ist immer nicht blockierend.

Zum Empfangen von Nachrichten wird ein Briefkastenkonstrukt zur Verfügung gestellt. Der Zugriff (Emfang von Nachrichten/Ergebnissen) auf den „Briefkasten“ kann in zwei verschiedenen Modi erfolgen.

1. Die Anwendung kann (blockierend) auf das Eintreffen einer Nachricht warten oder
2. mithilfe einer Status–Funktion feststellen, ob eine Benachrichtigung vorliegt.

Die Verwendung einer Mailbox impliziert, daß die Anwendung, die einem Briefkasten zugeordnet ist, exisitert. Die Verarbeitung kann verzögert erfolgen. Aufgabe des

[2] Die Systemarchitektur definiert, daß die Systemkomponenten immer vorhanden sein müssen bzw. daß der *ConTract Processing Monitor* daür zu sorgen hat, daß eine Überwachung und ein Wiederanlauf nach einer Fehlersituation stattfindet

ConTract Processing Monitor ist es, dafür zu sorgen, daß die benötigten Briefkästen erzeugt werden und verfügbar sind. Sollte eine Komponente aufgrund eines Fehlers (oder beim Neustart des Systems) ausgefallen sein, muß der PM einen „restart“ initiieren.

Die Adressierung des Empfängers einer Nachricht geschieht auf dieser Ebene implizit durch die entsprechende Auswahl des Kommunikationsaufrufes. Z.B. existiert ein Aufruf vom *ConTract Manager* an den *Transaction Manager* um eine Liste von „Transactionsidentifiern“ zu erhalten. Dieser Aufruf wird durch die Funktion `TM_get_gtrid_list` realisiert. Der Empfänger (TM) ist im Funktionsnamen enthalten.

6 NACHRICHTENEBENE
Die Aufrufe der Anwendungsebene werden alle auf einen Basis–Nachrichten–Mechanismus abgebildet. Dieser stellt eine Funktion zur Verfügung, mit der Nachrichten (asynchron) versendet werden können. Die andere Hauptaufgabe dieser Ebene ist die Realisierung der Mailbox. Sie ist auf dieser Ebene angesiedelt.

Auch ist hier eine explizite Adressierung möglich. Es wird hierfür aber nicht die physikalische Adresse sondern ein logisches Adessierungsschema verwendet. Eine Unterscheidung in lokale und entfernte Kommunikation ist zwar impilzit möglich[3], aber nicht erforderlich.

5 DATENREPRÄSENTATION
Diese Ebene ist in der Lage, den Zielrechner zu bestimmen. Aufgrund dieses Wissens wird ggf. eine Konvertierung der rechnerspezifischen Datendarstellung in eine neutrale Darstellung vorgenommen. Von Bedeutung ist dies insbesondere dann, wenn das APRICOT System in einer heterogenen Rechnerumgebung abläuft.

4 TRANSPORT-EBENE
Aufgabe der Ebene 4 ist es, die Übertragung der Daten zum Empfänger sicherzustellen. D.h., lokale Kommunikation oder Kommunikation über ein Netzwerk wird hier realisiert. Diese Ebene wird vom Betriebssystem zur Verfügung gestellt. Dabei erfolgt die Kommunikation über ein Netzwerk aufgrund des im OSI/ISO-Schichtenmodells definierten Transport Layer Interface (TLI) oder muß sich auf Netzwerkdienste der Ebene 3 (Sockets, Ports etc.) abstützen. Die lokale Kommunikation erfolgt ggf. mit den Hilfsmitteln zur Interprozess–Kommunikation des Betriebssystems.

6.1.3 Bemerkungen zur Spezifikation

- In den nachfolgenden zwei Abschnitten wird die Anwendungsebene (7) und die Nachrichtenebene (6) spezifiziert. Sie sind besonders wichtig, da sie das logische Konzept definieren mit dem die Komponenten im APRICOT System kommunizieren. Die Implementierung der tieferen Ebene ist anhängig von den Mitteln, die das Betriebssystem bietet, und wird hier nicht weiter erläutert[4].

[3] Bedingt durch die Systemarchitektur sind auf einem Knoten z.B. immer ein CM und ein TM vorhanden. D.h. ein Aufruf des CM an den TM ist immer lokal, ein Aufruf des CM an einen anderen CM immer entfernt.

[4] Die Remote–Kommunikation findet in der ersten Realisierung über RPCs statt.

- Die Deklaration der Funktionen erfolgt in C–Syntax. Für die Deklaration werden selbsterklärende Parameternamen verwendet. Rückgabeparameter lassen sich durch den vorangestellten Adressoperator (&) erkennen.

- Alle Funktionen (in Sinne von Funktionsaufrufen in C) liefern als primäres Funktionsergebniss einen Fehlercode (Typ `AS_RETURN_CODE`) zurück. Dadurch wird der Rufer über den Verlauf der Funktion informiert. Ergebnisse aufgrund des Aufrufes werden getrennt, über Zeigervariable, zurückgeliefert.

- Das Präfix „`cs_`" kennzeichnet Funktionen, Variablen bzw. Parameter, die direkt mit dem Kommunikationssystem zusammenhängen.

- Zur Illustration der Vorgehensweise und der zur Verfügung gestellten Funktionen wird der bereits erwähnte `get_gtrid_list`–Aufruf „*get global Transaction Identifier*" des *ConTract Manager* (CM) an den *Transaction Manager* (TM) verwendet. Der CM verwendet diesen synchronen Aufruf, um für die gerade zur Ausführung anstehenden Steps neue Transaktion–Id's zu erhalten.

6.2 Anwendungsebene

6.2.1 Schnittstelle nach oben bzw. gegenüber der Anwendung

6.2.1.1 Gruppen von Funktionen

Die Funktionalität der Anwendungsebene läßt sich in 3 Gruppen einteilen:

1. Funktionen zum Senden
   Zum synchronen Aufruf entfernter Funktionen (vgl. Remote Procedure Call), zum asynchronen Nachrichtenversand und zum Zurücksenden von Ergebnissen werden die Funktionen dieser Gruppe benötigt.

2. Funktionen zum Empfangen
   Die Daten, die mit den Funktionen der ersten Gruppe übertragen werden, müssen beim Empfänger entgegengenommen werden. Die Gruppe stellt die entsprechende Funktionalität zur Verfügung.

3. Sonstige Funktionen
   Dieser Gruppe besteht aus Funktionen, mit der die Anwendung sich beim Communication System anmelden muß und den Zustand eines Aufrufes feststellen kann, falls dieser asynchron ausgeführt wurde.

   (a) *Initialisierung und Anmeldung*
   Diese Funktion wird zu Beginn der Ausführung einer Anwendung aufgerufen. Die Funktion überprüft die Systemumgebung (vorhandensein des PM und der eigenen Mailbox etc.). Als Ergebnis der Funktion wird der Anwendung mitgeteilt, ob sie neu gestartet wurde oder ein 'restart' nach einem Fehler vorliegt. Im letzteren Fall muß die Anwendung Recovery–Aktionen durchführen.

   (b) *Statusfunktion für Nachrichten*
   Mithilfe der Statusfunktion besteht die Möglichkeit, festzustellen, ob eine Nachricht beim Empfänger abgeliefert wurde, ob dieser sie quittiert hat und ob ggf. die Rückantwort abgesendet wurde.

   (c) *Abmeldung*
   Analog dem Anmelden erfolgt das Abmelden mit einer getrennten Funktion. Diese meldet die Anwendung beim PM ab und gibt die für die Kommunikation belegten Resourcen frei.

6.2.1.2 Namensgebung für Funktionen

Für die Ermittlung des Names von Sende- und Empfangsfunktionen gilt untenstehendes Schema. Der Name besteht aus der Abkürzung für den Empfänger, genauer der Klasse in der der Empfänger ist, einer Typangabe bzgl. des Zweck des Aufrufes und des logischen Namens der auszuführenden Funktion[5], z.B. `get_gtrid_list`.

[5] Zur Realisierung einer logischen Funktion des Betriebsmodells werden für die Programmierung mehrere Funktionsaufrufe benötigt.

`XX_YYY_`*funktion*			
	`XX`		Abkürzung des Empfängers (bei Aufrufen oder beim Nachrichtenversand) oder der ausführenden Komponente (beim Lesen von Parameter, Initialisierung etc.).
	`YYY`		Art des Aufrufes:
		`SYN` ...	synchroner Aufruf
		`ASY` ...	asynchroner Aufruf
		`REC` ...	Funktion zum Lesen des Nachrichteninhaltes (Lesen des Aufrufs durch den „Server“ [receive]).
		`REP` ...	Funktion zum Senden des Ergebnisses an den Aufrufer (reply).
	funktion		Logische Funktion, für die der Aufruf benötigt wird.

Bemerkung:

Es muß nicht notwendigerweise jeder Aufruf in einer synchronen und einer asynchronen Version implementiert sein. Insbesondere sind die Aufrufe, die logisch als Nachricht gelten, nur als asynchrone Funktion verfügbar.

6.2.1.3 Senden

Der Benutzer des APRICOT System, respektive die Anwendungen welche die Wünsche des Anwenders umsetzen, benötigen verschiedene synchrone und asynchrone Aufrufe. Das Konzept sieht vor, den Anwendungen alle gewünschten Kommunikationsaufrufe als Funktionen zur Verfügung zu stellen. Zu bemerken ist, daß die meisten Systemkomponenten ihre Aufgaben als „Server“ erfüllen. Dies bedeutet, daß ein einzelner Aufruf nicht die gesamte Arbeit des „Servers“ blockieren darf. Aus diesem Grund werden einige logisch synchrone Aufrufe auch als asynchrone Funktionsaufrufe angeboten. Die Verwaltung der Asynchronität obliegt dann den jeweiligen Anwendungen.

Die Verbindung zwischen zusammengehörenden Kommunikationsaufrufen geschieht über einen Indentifikator, die „**handle**“. Mit ihr muß

- die Anwendung bei einem asynchronen Aufruf einer logisch synchronen Funktion die Verbindung zwischen dem Aufruf und der Ergebnisnachricht

und

- der „Server“ die Verbindung zwischen eingegangenem Auftrag und der Nachicht mit dem Ergebnis

herstellen.

Zur Illustration des Namensschemas und der vorhandenen Programmfunktionen zu einem logischen Aufruf dient nachfolgendes Beispiel. Der `get_gtrid_list` Aufruf ist logisch synchron. Der CM arbeitet allerdings als „Server“. Deshalb stehen eine synchrone und eine asynchrone Funktion zur Verfügung. Für den zeitlichen Ablauf im CM und im TM ergeben sich folgende Funktionsaufrufe:

- Aufruf durch den CM:

  `TM_SYN_get_gtrid_list ()`

  oder

  `TM_ASY_get_gtrid_list ()`

- Empfang der Aufrufparameter durch den TM:

  `TM_REC_get_gtrid_list ()`

- Absenden des Ergebnisses durch den TM an den CM:

  `CM_REP_get_gtrid_list ()`

- Entgegennehmen des Ergebnisses durch den CM beim asynchronem Aufruf der Funktion:

  `CM_REC_get_gtrid_list ()`

  Im synchronen Fall erhält die Anwendung das Ergebnis durch die Rückgabeparameter des Funktionsaufrufes.

Bemerkungen:

- Für den gesamten Ablauf in der Anwendung sind insgesamt noch weitere Funktionen notwendig (Empfangen der Nachrichten). Das Beispiel am Ende des Abschnittes über die Anwendungebene beschreibt einen vollständigen Ablauf (Abschnitt 6.2.5 ab Seite 73).

6.2.1.4 Empfangen

Das Empfangen von Nachrichten (Auftrag oder Rückantwort) geschieht in drei Schritten.

1. Warten auf eine Nachricht
   Dieser Vorgang kann blockierend oder nicht blockierend erfolgen. D.h., die Anwendung kann solange warten, bis eine Nachricht eintrifft oder sie kann durch die Funktion feststellen, ob „Messages“ vorliegen. Letztere Möglichkeit entspricht dem Abfragen des Status des Briefkastens bezüglich einer oder mehrerer Nachrichtentypen.

   Durch die Angabe eines entsprechenden Parameters kann der Aufrufer bestimmen, ob er auf eine beliebige Nachricht oder eine bestimmte (bzw. einer Teilmenge aller möglichen) warten will.

   Sollte ein gewünschter Nachrichtentyp angekommen oder bereits im Briefkasten sein, liefert die Funktion auch hier eine „handle“ zurück. Die „handle“ wird für die Zuordnung der Ergebnisnachricht zum Aufruf benötigt.

2. LESEN DER NACHRICHT
   Im zweiten Schritt liest die Anwendung den Inhalt der Nachricht. Das Lesen erfolgt, analog dem Senden, mit entsprechenden Funktionen, die als Eingabeparameter u.a. die zuvor erhaltene „handle“ erwarten.

   Durch das Lesen wird die Nachricht markiert und gesperrt. Die Sperre bedeutet, daß bei einem weiteren Aufruf der Warte-Funktion (die meisten Komponenten arbeiten als „Server“) diese Nachricht nicht ein weiteres Mal zurückgeliefert wird. Die Anwendung kann nach dem Lesen den Erhalt der Nachricht zu einem beliebigen Zeitpunkt quittieren. Bei einer positiven Bestätigung wird die markierte Nachricht endgültig aus dem Briefkasten entfernt. Eine negative Quittierung gibt die Sperre frei. Die Nachricht kann beim nächsten Aufruf der Warte–Funktion wieder gelesen werden.

   Die Markierung und anschliessende Quittierung einer Nachricht erfolgt aus folgenden Gründen:

   - Die Anwendung kann nach dem Lesen der Nachricht diese wieder zurückstellen.
   - Die Realisierung einer „recoverable“ Anwendung wird erleichtert. Sollte zwischen dem Lesen und dem Quittieren einer Nachricht ein Fehler auftreten, der zum Ausfall der Komponenten führt, so werden alle Sperren auf Nachrichten ausgehoben. D.h., ruft die Anwendung nach dem „restart“ die Warte-Funktion auf, erhält sie alle nicht quittierten Nachrichten von Neuem.

     Damit muß die Anwendung das Lesen einer Nachricht nicht selbst sicher protokollieren. Sie kann die Nachricht lesen und muß erst später, z.B. nach einem Savepoint oder wenn ein Zurücknehmen ausgeführter Aktion unmöglich ist, den Erhalt quittieren.

3. QUITTIERUNG DER NACHRICHT
   Hat die Anwendung die Nachricht gelesen und möchte, daß sie aus der Mailbox entfernt wird, so muß sie diese mit der `acknowlegde_message`–Funktion quittieren. Nach einer positiven Quittierung wird die Nachricht aus dem Briefkasten entfernt. Eine negative Quittierung bewirkt, daß die Nachricht wieder zurückgestellt wird, d.h., sie gilt als nicht verarbeitet und kann bei erneuten Lesen des Briefkastens wieder erhalten werden (vgl. Freigeben einer exclusiven Sperre auf ein (Daten-)Objekt)

6.2.2 Aufgaben der Anwendungsebene

Zu den Aufgaben gehört es, die nötigen Aufrufe der Anwendungen auf einen Basisaufruf abzubilden. Dieser dient zum asynchronen Versenden von Nachrichten.

Eine weitere Aufgabe der Anwendungsebene ist die Umsetzung der Parameterdarstellung die in der Programmiersprache der Anwendung verwendet wird, in die von den Basisaufrufen verwendete Form. Die Anzahl der Parameter der Funktionsaufrufe in der Anwendungsebene variiert bei den einzelnen Aufrufen. Aufrufparameter müssen zu einer Struktur zusammengefaßt werden, da die Basis–Nachrichten-Funktion eine feste Anzahl von Parametern hat. Zusätzliche Bedeutung erhält dieser Punkt bei Verwendung von Programmiersprachen, die eine Definition von Strukturen nicht zulassen (z.B. FORTRAN).

6.2.3 Schnittstelle zur nächsten Ebene

Die Schnittstelle nach unten ist „leer“, denn die Schicht 6 hat keine Möglichkeit mit der Anwendung zu kommunizieren. Alle Aktivitäten gehen von der Anwendung aus.

6.2.4 Spezifikation der oberen Schnittstelle

Die obere Schnittstelle besteht aus nachstehenden Funktionen. Bei den Sendefunktionen werden nur die logischen Funktionen des Betriebsmodells angeführt.

6.2.4.1 Sendefunktionen

In diesem Abschnitt erfolgt nicht die Spezifikation aller verfügbaren Funktionen. Vielmehr werden die im Zusammenhang mit dem `get_gtrid_list`-Aufruf zur Verfügung gestellten Sendefunktion beispielhaft spezifiziert.

- `get_gtrid_list` als synchroner Aufruf

```
TM_SYN_get_gtrid_list(number_gtrids, &function_code, &gtrid_list)
```

Parameter	C-Typ	Erklärung
number_gtrids	long	Anzahl der Transaktions–Identifier, die benötigt werden.
function_code	FUNCTION_CODE	Kennung, die den Verlauf der Funktionsausführung beschreibt.
gtrid_list	GTRID_LIST	Liste mit den „Transaction Identifier“.

- `get_gtrid_list` als asynchroner Aufruf

`TM_ASY_get_gtrid_list(&cs_handle, cs_mode, number_gtrids)`

Parameter	C-Typ	Erklärung
`cs_mode`	CS_MODE	Modus der Funktionsausführung. Folgende Modi sind zulässig: `CS_MODE_NNO`[a], `CS_MODE_NOR`[b], `CS_MODE_NOP`[c]. oder `CS_MODE_NOA`[d]. Der Modus bestimmt, wann der Erfolg der Kommunikation gemeldet werden soll. Sofort nach Abnahme der Auftrages durch das Communication System (NNO), nach dem Eintragen der Nachricht im Empfängerbriefkasten (NOR), nach der Quittierung durch den Empfänger (NOP) oder nach Eintreffen der Rückantwort (NOA).
`number_gtrids`	long	Anzahl der Transaktions–Identifier, die benötigt werden.
`cs_handle`	CS_HANDLE	Handle zur Verbindung des Aufrufs mit der Ergebnisnachricht.

[a]NNO ... No Notification
[b]NOR ... Notification of Receipt
[c]NOP ... Notification of Processing
[d]NOA ... Notification of Answer

Bemerkungen:

- Wenn der Erfolg einer Kommunikation erst durch das Eintreffen der Antwort erfolgen soll (Modus `CS_MODE_NOA`), dann entspricht dies einem synchronen Aufruf der Funktion.
- Bei Aufrufen, die im Betriebsmodell nicht als Aufruf sondern als Nachrichten klassifiziert wurden, ist der Modus `CS_MODE_NOA` unzulässig, da keine Ergebnisnachricht existiert.

- `get_gtrid_list` Ergebnis zurücksenden

```
TM_REP_get_gtrid_list(cs_handle, cs_mode, function_code, gtrid_list)
```

Parameter	C-Typ	Erklärung
`cs_handle`	CS_HANDLE	Handle, die angibt, welchem Auftrag dieses Ergebnis zuzuordnen ist.
`cs_mode`	CS_MODE	Modus der Funktionsausführung. Folgende Modi sind zulässig: `CS_MODE_NNO`, `CS_MODE_NOR` oder `CS_MODE_NOP`.
`function_code`	FUNCTION_CODE	Kennung, die den Verlauf der Funktionsausführung beschreibt.
`gtrid_list`	GTRID_LIST	Liste mit den erzeugten Transaktions–Id's

6.2.4.2 Empfangsfunktionen

- `lookup_message` — Warten auf eine Nachricht

```
CS_lookup_message (&cs_handle, &cs_msg_typ, cs_mode, cs_handle_list, cs_msgtyp_list)
```

Parameter	C-Typ	Erklärung
`cs_mode`	CS_MODE	Modus der Funktionsausführung. Folgende Modi sind zulässig: `CS_MODE_WAIT` oder `CS_MODE_NOWAIT`. Bei `WAIT` kehrt der Aufruf erst zurück, wenn eine entsprechende Nachricht vorhanden ist. Im Modus `NOWAIT` kehrt die Funktion nach Sichtung der Nachrichten immer zurück. Sollte keine passende „Message“ vorhanden sein, enthält `cs_handle` die NULL–Handle `CS_NULL_HANDLE`.
`cs_handle_list`	CS_HANDLE_LIST *	Liste mit „handles“, auf die gewartet werden soll. Soll auf eine beliebige gewartet werden, kann die NULL-Liste `CS_NULL_LIST` angegeben werden.
`cs_msgtyp_list`	CS_MSGTYP_LIST *	Liste mit Typen von Nachrichten, auf die gewartet werden soll. Soll auf eine beliebige gewartet werden, kann die NULL-Liste `CS_NULL_LIST` angegeben werden.
`cs_handle`	CS_HANDLE	Handle zur Identifikation der Nachricht und zur Zuordnung eines eventuellen Ergebnisses.
`cs_msg_typ`	CS_MSGTYP	Typ der Nachricht, die ausgewählt wurde.

Bemerkungen:

– Die Suche nach einer passenden „handle“ hat Vorrang vor der Suche nach einem entsprechenden Nachrichtentyp.

- `get_gtrid_list` Parameter lesen

`TM_REC_get_gtrid_list (cs_handle, cs_mode, &number_gtrids)`

Parameter	C-Typ	Erklärung
`cs_handle`	CS_HANDLE	Handle, die beim `lookup`–Aufruf als Identifikator erhalten wurde.
`cs_mode`	CS_MODE	Modus bezüglich der Quittierung. Zulässige Werte sind: `CS_MODE_NOACK` oder `CS_MODE_ACK`. Bei `NOACK` will die Anwendung die Nachricht getrennt quittieren. Ein Aufruf mit Modus `ACK` beinhaltet eine automatische Quittierung.
`number_gtrids`	long	Anzahl der Transaktions–Identifier, die benötigt werden.

- `acknowledge_message` — Nachricht quittieren

`CS_acknowlegde_message (cs_handle)`

Parameter	C-Typ	Erklärung
`cs_handle`	CS_HANDLE	Handle, die beim `lookup`–Aufruf als Identifikator erhalten wurde.

- `get_gtrid_list` Ergebnis lesen

`CM_REC_get_gtrid_list (cs_handle, cs_mode, &function_code, &gtrid_list)`

Parameter	C-Typ	Erklärung
`cs_handle`	CS_HANDLE	Handle, die beim `lookup`–Aufruf als Identifikator erhalten wurde.
`cs_mode`	CS_MODE	Modus bezüglich der Quittierung. Zulässige Werte sind: `CS_MODE_NOACK` oder `CS_MODE_ACK`. Bei `NOACK` will die Anwendung die Nachricht getrennt quittieren. Ein Aufruf mit Modus `ACK` beinhaltet eine automatische Quittierung.
`function_code`	FUNCTION_CODE	Kennung, die den Verlauf der Funktionsausführung beschreibt.
`gtrid_list`	GTRID_LIST	Liste mit den Transaktions–Identifier.

6.2.4.3 Sonstige Funktionen

- INITIALISIERUNG UND ANMELDUNG
  Nach dem Start einer Komponente muß sich diese beim *Compensation Manager* anmelden. Bei dieser Anmeldung werden die notwendigen Kommmunikationsvariablen und -parameter initialisiert. Komponenten, die nur einmal auf einem lokalem System vorhanden sein können (CC, CM, COM, PM, TM), benutzen zur Anmeldung Version 1 des Aufrufes. Die anderen Komponenten (AP, SS, RM) die Version 2.

  1. *Version 1*

     `XX_initialization (&cs_startup_mode)`

     $XX \in \{CC, CM, COM, PM, TM, LM, RC\}$

Parameter	C-Typ	Erklärung
`cs_startup_mode`	CS_STARTUP_MODE	Modus, in dem die Komponente gestartet worden ist. Mögliche Werte sind: `CS_STARTUP_NEW`, `CS_STARTUP_RECOVERY` oder `CS_STARTUP_NOCONNECT`. Aus dieser Modusangabe kann die Anwendung erkennen, ob sie neu gestartet wurde (Modus `CS_STARTUP_NEW`), ob ein Ausfall vorlag und deshalb Recovery–Maßnahmen durchzuführen sind (Modus `CS_STARTUP_RECOVERY`) oder keine Anmeldeerlaubnis besteht. Dies kann der Fall sein, wenn eine Komponente versucht sich anzumelden, obwohl eine andere aktive Instanz dies bereits getan hat.

2. *Version 2*

`YY_initialization(&cs_startup_mode, cs_connect_id)`

$YY \in \{AP, OM, SS\}$

Parameter	C-Typ	Erklärung
`cs_connect_id`	CS_CONNECT_ID	Kennung, die eine Instanz identifiziert. Sind von einer Komponente mehrere Instanzen vorhanden, z.B. mehrere SS, dann müssen diese unterschieden werden, insbesondere wenn durch einen Fehler ein „restart“ erforderlich wird. Diese Unterscheidung erfolgt durch die `connect_id`, welche von den Komponenten selbst zu verwalten ist.
`cs_startup_mode`	CS_STARTUP_MODE	Modus, in dem die Komponente gestartet worden ist. Mögliche Werte sind: `CS_STARTUP_NEW`, `CS_STARTUP_RECOVERY` oder `CS_STARTUP_NOCONNECT`. Aus dieser Modusangabe kann die Anwendung erkennen, ob sie neu gestartet wurde (Modus `CS_STARTUP_NEW`), ob ein Ausfall vorlag und deshalb Recovery–Maßnahmen durchzuführen sind (Modus `CS_STARTUP_RECOVERY`) oder keine Anmeldeerlaubnis besteht. Dies kann der Fall sein, wenn eine Komponente versucht sich anzumelden, obwohl eine andere aktive Instanz dies bereits getan hat.

Bemerkungen:

- *Step Computation Server* müssen mit einem zusätzlichen Aufruf (`ss_registration`) dem *ConTract Processing Monitor* mitteilen, welche Dienste sie den Benutzern anbieten. Erst durch diese Information ist der PM in der Lage, den SS in das Scheduling miteinzubeziehen und somit dafür zu sorgen, daß der SS Aufträge erhält.
- Eine *Application* erhält immer den Modus `NEW`, da von ihr keine Recovery–Fähigkeiten erwartet werden.

- STATUSFUNKTION FÜR NACHRICHTEN
  Bei den Sendefunktionen besitzt der Sender die Möglichkeit, beim Aufruf anzugeben, wann ihm der Erfolg eines Aufrufs gemeldet werden soll (genauer: wann die Kommunikationsfunktion zurückkehren soll). Diese Angabe erfolgt durch entsprechende Wahl einer Modusangabe bei `cs_mode`. Folgende Angabe sind dabei zulässig:

Modus	Bedeutung
CS_MODE_NNO	*No Notification* Der Aufruf kehrt nach Annahme des Auftrages durch das Communication System sofort zum Aufrufer zurück.
CS_MODE_NOR	*Notification of Receipt* Der Aufruf kehrt erst zurück, wenn der Auftrag im Empfängerbriefkasten angelangt ist.
CS_MODE_NOP	*Notification of Processing* Der Aufruf kehrt erst zurück, wenn der Empfänger die Nachricht (positiv) quittiert hat.
CS_MODE_NOA	*Notification of Answer* Der Aufruf kehrt erst zurück, wenn der Empfänger auf die Nachricht ein Ergebnis gesendet hat.

Ein andere Möglichkeit, den Zustand einer Nachricht festzustellen, ist der Aufruf nachfolgend definierter Statusfunktion. `CS_get_msg_state`.

```
CS_get_msg_state (cs_handle, &cs_msg_state)
```

Parameter	C-Typ	Erklärung
cs_handle	CS_HANDLE	Handle, die beim Senden als Identifikator zurückgeliefert wurde.
cs_msg_state	CS_MODE	Zustand der nachgefragten Nachricht. Mögliche Werte sind: CS_MSGSTATE_NOR, CS_MSGSTATE_NOP oder CS_MSGSTATE_NOA

Bemerkungen:

- Der Zustand `CS_MODE_NOA` kann nur zurückgeliefert werden, wenn der Sendeaufruf ein „Call“ war und somit überhaupt eine Rückantwort existieren kann.
- `CS_MODE_UNKOWNHANDLE` (*unkown message*) wird als Kennung für eine ungültige Handle verwendet.

- ABMELDUNG
  Analog dem Anmelden muß sich eine Komponente auch beim Communication System abmelden. Sollte eine Anwendung sich beenden, ohne eine Abmeldung durchzuführen, wird sie beim nächsten Aufruf der Initialisierungsfunktion den Modus `RECOVERY` zurückerhalten.

```
CS_cleanup ()
```

Wie mit den nicht verarbeiteten Nachrichten beim Aufruf der `cleanup`–Funktion verfahren wird, hängt von der Komponente ab. Zu unterscheiden sind:

– **CC, CM, COM, PM, TM, LM, RC**:
Diese Komponenten müssen aufgrund der Systemarchitektur immer vorhanden sein. D.h., nach dem Aufruf der `cleanup`-Funktion werden noch nicht verarbeitete und noch eingehende Nachrichten zwischengespeichert. Die Komponente erhält diese Nachrichten nach dem nächsten Anlauf. Gleiches gilt für Nachrichten, die gelesen aber nicht quittiert wurden. Sie werden beim nächsten Anlauf erneut durch die `lookup`–Funktion bereitgestellt.

– **AP, SS, OM**:
Nicht verarbeitet und nicht quittierte Nachrichten werden bei einem erneuten Anlauf nicht wieder zurückgeliefert, sie gehen verloren. Bevor die Nachrichten verworfen werden, wird allerdings folgendes Verfahren angewendet:

  * Komponenten, die einen „Call“ abgesetzt haben und auf die Rückantwort warten, erhalten eine Ergebnisnachricht, die durch das Communication System erzeugt wird und den negativen Verlauf des Aufrufes bekanntgibt.
  * „Messages“ aufgrund von *Nachrichten* werden ohne weitere Aktionen verworfen. Will ein Absender wissen, ob seine Nachricht verarbeitet wurde, muß er sie mit dem entsprechenden Modus (`NOP, NOR`) absenden oder mit der Statusfunktion `cs_get_msg_state` den Verlauf überprüfen.

Bemerkungen:

  * *Step Computation Server* müssen mit dem `ss_registration`–Aufruf dem *ConTract Processing Monitor* mitteilen, welche Dienste sie den Benutzern anbieten. Erst dann wird der SS in das Scheduling miteinbezogen und erhält Aufträge. Am Ende der Arbeit eines *Step Computation Server* muß dieser sich entsprechend beim PM abmelden (`ss_unregistration`), um keine weiteren Aufträge zu erhalten. Danach kann er sein bereits angenommen Auftäge zu Ende führen und sich mit `cs_cleanup` beenden. Dies stellt den ordnungsgemäßen Ablauf zum Beenden eines SS dar und somit erhält keine Anwendung eine Fehlermeldung, die das Nichtvorhandensein einer Komponenten angibt.

6.2.5 Beispiele für die Funktionalität der Anwendungsebene

Die Abbildungen 6.2 und 6.3 zeigen den gesamten Ablauf eines logisch synchronen Aufrufes anhand des Beispiels `get_gtrid_list`. In Abbildung 6.2 ist der zeitliche Ablauf dargestellt, wenn der Sender (*ConTract Manager* blockierend wartet, während Abbildung 6.3 den gleichen Auruf in der asynchronen (nicht blockierenden Variante) zeigt.

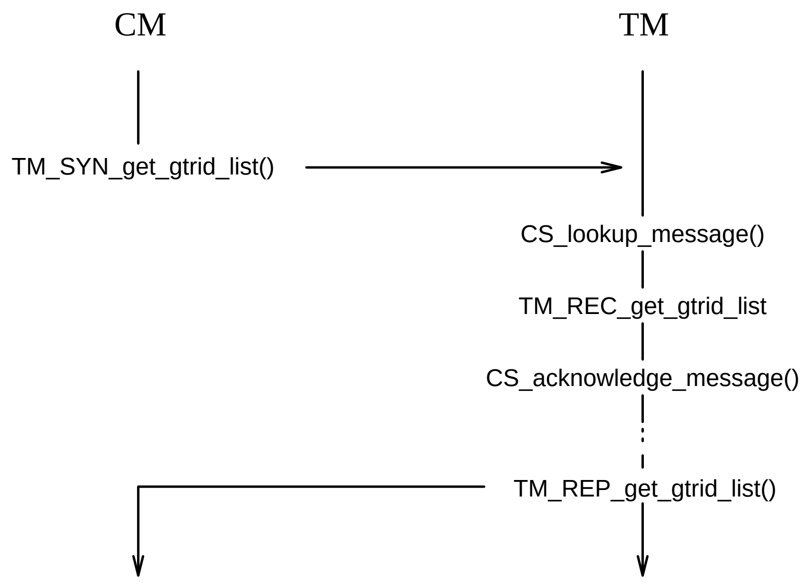

Abbildung 6.2: synchroner Aufruf von `get_gtrid_list()`

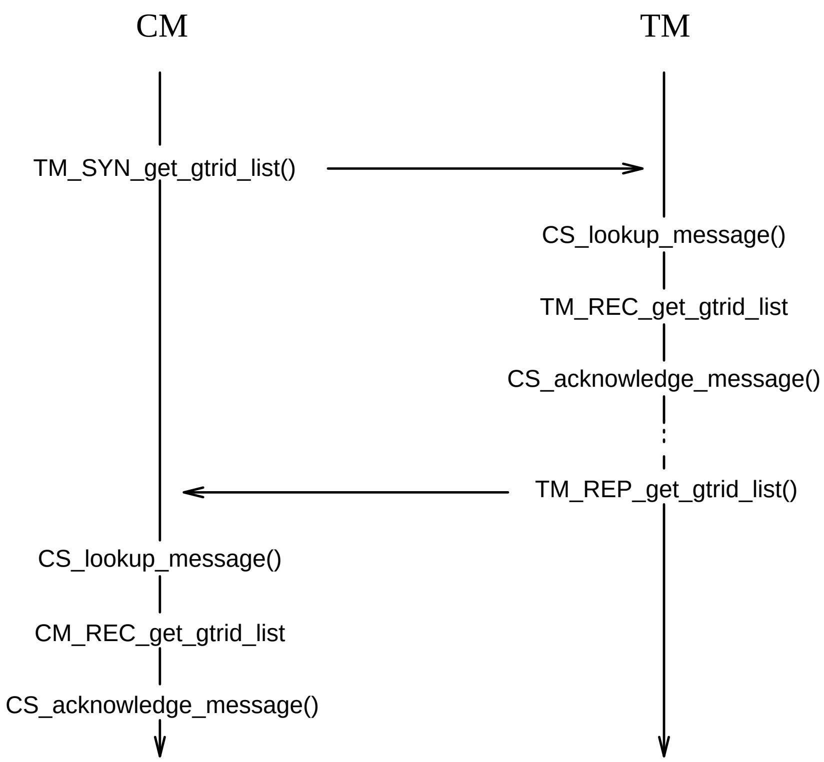

Abbildung 6.3: asynchroner Aufruf von `get_gtrid_list()`

6.3 Nachrichtenebene

Die Nachrichtenebene ist die zweite Ebene von oben und hat im APRICOT System eine besondere Bedeutung, da hier der eigentliche Briefkastenmechanismus realisiert ist. Darüber hinaus gibt es noch zwei weitere Aufgabengebiete. Zum einen wird der nächsthöheren Ebene ein Basis-Nachrichtenfunktion (zum Versenden) zur Verfügung gestellt, zum anderen existieren noch Funktionen, um der tieferen Ebene das „Einwerfen von Post", sprich das Abliefern eingegangener Nachrichten, zu ermöglichen.

6.3.1 Schnittstelle nach oben

Die Funktionalität der *Schnittstelle nach oben* läßt sich in zwei Gruppen einteilen:

1. VERSENDEN VON NACHRICHTEN
   Alle synchronen und asynchronen Aufrufe der Anwendungsebene müssen auf eine Basis-Nachrichtenfunktion abgebildet werden. Diese Funktion ist asynchron und nach folgendem Schema aufgebaut:

   `CS_send_message(`*Empfänger, Absender, Beschreibungsinformation, Nachricht*`)`

   Dem Schema kann entnommen werden, daß neben der anderen Darstellung des Nachrichteninhaltes auch eine explizite Adressierung notwendig ist. Dieses Adressierungsschema verwendet aber keine physikalischen Adressen, wie z.B. Internet–Adresse, Rechnernamen, Prozesse-ID's etc., vielmehr erfolgt eine logische Adressierung der einzelnen Systemkomponenten.

2. BEARBEITEN DER MAILBOX
   „Post muß gesichtet und geöffnet werden." Dieser Analogie folgend existieren Funktionen zur Auswahl und zum Lesen einer eingegangenen Nachricht.

6.3.2 Aufgaben der Nachrichtenebene

Die Aufgabe der Nachrichtenebene besteht, wie einleitend erwähnt, aus den Punkten:

1. Funktionalität zum Versenden von asynchronen Nachrichten.
2. Realisierung des Briefkastens und der notwendigen Verwaltungsfunktionen.
3. Funktionen zum Eintragen von „Messages" in den Briefkasten durch die tiefere Schicht.

Bei der Realisierung des Briefkastens sind einige Anforderungen zu erfüllen.

- IDENTIFIKATION EINER EINZELNEN NACHRICHT
  Hierzu muß eine geeignete Umsetzung der in der Anwendungsebene bekannten **„handle"** erfolgen. Eine vergebene „handle" muß eindeutig sein, und eine Neuverwendung darf frühestens dann erfolgen, wenn alle ausstehenden Rückantworten eingegangen sind.

- Zuordnung entfernter Nachrichten
  Andere Komponenten benutzen andere „handle“. Beim Zurücksenden eines Ergebnisses und bei der Ermittlung des Verarbeitungszustandes (siehe *Statusfunktion*, Abschnitt 6.2.4.3 auf Seite 71) einer Nachricht muß eine Zuordnung der verschiedenen „handle“ erfolgen.

- Verlustfreiheit
  Nachrichten dürfen nicht verloren gehen. D.h., nach Annahme einer eingegangenen Nachricht muß die Nachrichtenebene dafür sorgen, daß die Anwendung diese auch erhalten kann. Der Ausfall der Anwendung oder der Zusammenbruch des ganzen Rechnersystems darf nicht zum Verlust einer Nachricht führen.

- Wiederanlauf
  Beim Wiederanlauf der ausgefallenen Komponenten müssen der Anwendung die bereits gelesenen aber noch nicht quittierten Nachrichten erneut zur Verarbeitung geliefet werden. Die hierfür notwendigen Aktionen sind auf dieser Ebene realisiert.

- Überlauf des Briefkasten
  Ein „Briefkasten“ besitzt ein begrenztes Fassungsvermögen. Der Briefträger wird dieses Problem dadurch lösen, daß er entweder die Post nicht in den Briefkasten, sondern davor oder darunter (irgenwohin) legt oder sie mit ins Postamt nimmt, um sie dort zu deponieren.

  Der „Postbote“ im Communication System ist die Nachrichtenebene. Das Überlaufproblem beim APRICOT-Briefkasten muß also hier gelöst werden. I.A. wird eine Lösung der Aufgabe im Zusammehang mit dem Problem der *Verlustfreiheit* realisiert werden.

- Nachricht bei Unerreichbarkeit
  Existiert der Empfänger nicht, oder ist er nicht mehr verfügbar[6], dann muß eine entsprechende Rückantwort im Falle eines synchronen Aufrufes (Call) erzeugt werden, um die Anwendung von dieser Tatsache zu informieren.

6.3.3 Schnittstelle zur nächsten Ebene

Die Nachrichtenebene muß der tieferen Schicht zwei Funktionen zur Verfügung stellen:

1. Eintragen der Nachricht in den Briefkasten
   Eingegangene Nachrichten müssen bei der Nachrichtenebene abgeliefert und in die Mailbox eingetragen werden.

2. Statusabfrage bzgl. einer fremden Nachricht
   Komponenten können mithilfe der Statusfunktion (siehe Abschnitt 6.2.4.3 ab Seite 71) den Zustand der Verarbeitung nachfragen. Solche Anfragen gehen über die tieferen Ebene ein und müssen an die Nachrichtenebene weitergeleitet werden.

[6] Beispiel: Ein SS hat sich abgemeldet.

6.3.4 Spezifikation der oberen Schnittstelle

6.3.4.1 Nachrichtenversand

Den Funktionen der Anwendungsebene steht als Basismechanismus die asynchrone CS_send_message Funktion zur Verfügung.

Für die Adressierung der einzelnen Komponenten wird ein dreistufiges logisches Adressierungschema verwendet.

1. Kennung des Rechnerknotens
   Jedem Rechner, der im APRICOT System konfiguriert[7] ist, wird eine eigene Kennung vergeben. Dies kann u.U. auch die Ethernet–Adresse sein. Wichtig ist nur die Eindeutigkeit der Kennung im APRICOT System.

2. Klassenangabe der Empfängerkomponente
   Die Systemarchitektur des APRICOT System definiert eine Gliederung der einzelnen Komponente in Klassen.

AP,OM	...Klasse der *Applications*
CC	...Klasse der *ConTract Client*s
CM	...Klasse der *ConTract Manager*
COM	...Klasse der *Compensation Manager*
PM	...Klasse der *ConTract Processing Monitor*
LM	...Klasse der *Global Log Manager*
SS	...Klasse der *Step Computation Server*
RM	...Klasse der *Resource Manager*
RC	...Klasse der *Recovery Manager*
TM	...Klasse der *Transaction Manager*

   Die zweite Stufe des Adressierungschema besteht deshalb aus einer Klassenkennung.

3. Kennung der Instanzierung
   Aus manchen Klassen können zur gleichen Zeit auf demselben Rechner mehr als eine Instanz (Prozeß) aktiv sein. Z.B. können mehrere *Step Computation Server* konfiguriert sein. Zur eindeutigen Adressierung der Instanzen wird die dritte Stufe des Adressierungschematas verwendet.

Die genauer Spezifikation der Funktion lautet:

```
CS_send_message(cs_receiverid, pto_cs_msg_info, pto_cs_message)
```

[7] Die Verwaltung und Verteilung der Konfigurationsinformation gehört in den Zuständigkeitsbereich des PM.

Parameter	C-Typ	Erklärung
cs_receiverid	CS_AP_ADRESS	Kennung (Adresse) des Empfängers der Nachricht. Ist der Empfänger auf demselben Rechner (lokale Kommunikation), so muß bei der Adresse nicht die Kennung des eigenen Rechners angegeben werden. Es genügt die Angabe CS_NULL_NODE.
pto_cs_msg_info	CS_MSG_INFO *	Zeiger auf eine Struktur mit Informationen zur Nachricht. Hierzu gehört u.a. der Typ der Nachricht und die lokale „handle“.
pto_cs_message	void *	Zeiger auf die eigentliche Nachricht. Die Interpretation der Daten, auf die verwiesen wird, ergibt sich aus der Typangabe in der CS_MSG_INFO Struktur.

Bemerkungen:

- Die Angabe des Absenders ist beim Aufruf der Funktion nicht notwendig. Die Information, wer Absender der Nachricht ist, wird durch das Communication System hinzugefügt.

6.3.4.2 Briefkastenbearbeitung

Auf der Anwendungsebene existieren drei Gruppen von Funktionen, mit denen eine Berarbeitung des Briefkastens möglich ist.e

1. Warten auf Nachrichten
   Diese Gruppe besteht nur aus der Funktion CS_lookup_message. Sie besitzt die gleiche Funktionalität wie die Funktionen CS_lookup_message der Anwendungsebene. Genauer gesagt: Es ist die gleiche Funktion. Eine Beschreibung der Funktion befindet sich auf Seite 68.

2. Lesen des Nachrichteninhalts
   Zum Lesen einer Nachricht existieren in der Anwendungsebene viele Funktionen. Diese benutzen eine Funktion, die den Inhalt einer Nachricht zur Verfügung stellt.

```
CS_get_data(cs_handle, cs_msg_type, pto_data)
```

Parameter	C-Typ	Erklärung
cs_handle	CS_HANDLE	Handle zur Identifikation des Eintrags in der Mailbox.
cs_msg_typ	CS_MSGTYP	Typ der Nachricht. Die Angabe bestimmt die Interpretation der Daten (der Bytefolge).
pto_data	char *	Zeiger auf den Speicherbereich zur Aufnahme des Nachrichteninhalts.

3. Verwaltung der Nachrichten
   Hierunter fallen insbesondere die Funktionen zum Sperren, Freigeben und Entfernen von Nachrichten der Mailbox.

   - *Sperren*

     `CS_lock_message(cs_handle)`

   - *Freigeben*

     `CS_unlock_message(cs_handle)`

   - *Löschen*

     `CS_delete_message(cs_handle)`

6.3.5 Spezifikation der unteren Schnittstelle

6.3.5.1 Eintragen einer Nachricht

Das Hinzufügen einer neuen Nachricht erfolgt mit:

`CS_insert_message (&cs_handle, cs_senderid, pto_cs_msg_info, pto_cs_message)`

Parameter	C-Typ	Erklärung
cs_senderid	CS_AP_ADRESS	Kennung (Adresse) des Ansenders der Nachricht.
pto_cs_msg_info	CS_MSG_INFO *	Zeiger auf eine Struktur mit Informationen zur Nachricht. Hierzu gehört u.a. der Typ der Nachricht und die entfernte „handle".
pto_cs_message	void *	Zeiger auf die eigentliche Nachricht. Die Interpretation der Daten auf die verwiesen wird, ergibt sich aus der Typangabe in der CS_MSG_INFO Struktur.
cs_handle	CS_HANDLE	Handle zur Identifikation des neuen Eintrags in der Mailbox.

6.3.5.2 Statusabfrage einer Nachricht

Die Statusabfrage kann von der Komponente aus erfolgen, die die Nachricht gesendet hat. Sie benutzt dazu eine ähnliche Funktion wie die, die der Anwendungsebene zur Verfügung gestellt wird (Funktion `CS_get_msg_state` von Seite 72). Der Unterschied besteht in der „handle"-Angabe. Hier muß die „handle" des Absenders und nicht die lokale angegeben werden.

`CS_get_msg_state_remote (cs_remote_handle, &cs_msg_state)`

6.4 Bewertung

Welche Erfahrungen wurden mit dem Kommunikationskonzept und der Realisierung des ConTract *System gemacht?*

Das Arbeit dieser Diplomarbeit kann bezüglich einer Bewertung des Communication System in zwei Teilbereiche gegeliedert werden:

1. Allgemeines Kommunikationskonzept
2. Realisierung des Kommunikationssystems

6.4.1 Kommunikationskonzept

Ein Ziel, daß mit der Entwicklung eines allgemeinen Kommunikationskonzeptes verbunden war, ist die Entlastung der Programmierer durch einfach hanzuhabende Mechanismen. Die Realisierung des APRICOT Systems umfaßte mehrere Diplom– und Studienarbeiten. Die Zusammenarbeit hat gezeigt, daß der Kommunikationsmechanimus von den jeweiligen Bearbeitern gut verstanden wurde und sie sich auf die Implementierung ihrere Aufgaben konzentieren konnten.

Das Kriterium „Unabhängigkeit“, insbesondere Unabhängigkeit vom Betriebssystem konnte überprüft werden. Die Implementierung des Communication System erfolgte nur unter dem UNIX–Betriebssystem SUN OS (4.1). Eine Portierung auf andere Betriebsysteme, auch auf andere UNIX–Derivate, ist nicht erfolgt. Die erstellten Programm wurden in der Sprache „C“ erstellt und als Konstrukte des Betriebssystems wurden nur *Shared Memory* und *Semaphore* verwendet. Die Konstrukte sind Bestandteil des *System V* Standards. Deshalb ist zu erwarten, daß eine Portierung auf andere UNIX–Betriebsysteme ohne großere Schwierigkeiten möglich ist.

Die Kommunikation über das Netzwerk wurde mihilfe von Remote Procedure Calls (RPC) realisiert. Wie Erfahrungen aus anderen Arbeiten am Institut zeigen, ist eine Umstellung auf Socket–basierte Kommunikation mit dem Protokoll TCP/IP in einer heterogenen Umgebung möglich.

6.4.2 Realisierung des Kommunikationskonzeptes

Die Realsierung der „Mailboxes“ in dem Kommunikationskonzept erfolgt auf der Basis von *Shared Memory*. Die Briefkästen sind Bestandteil des Adressraums der einzelnen Komponenten. Im Falle der lokalen Kommunikation schreiben die Komponenten direkt in die Mailbox des Empfängers. Synchronisiert wird der Zugriff auf den gemeinsamen Speicher durch Semaphore. Der Vorteil dieses Verfahrens ist, daß die Mailbox nach der Initialisierung solange erhalten bleibt bis sie explizit gelöscht werden oder das (Betriebs)System beendet wird. D.h., die Mailbox ist unabhängig von der Existenz des zughehörigen Komponentenprozeß.

Eine Konsequenz aus der Unabhängigkeit vom Komponentenprozeß ist allerdings, daß Vorkehrungen zur Erhaltung der Konsistenz des Briefkasten getroffen werden müssen. Die

Aufgabe hat große Ähnlichkeit mit der Implementierung von Datenbanken. So wurde für die Mailbox eine Art *Twin–Slot–Verfahren* (siehe [16]) verwendet. Änderungen einer Transaktion (eines Kommunikationsaufrufes) werden in einer zweite Speicherseite ausgeführt und nach Ende des Aufrufes freigegeben. Diese Vorgehensweise läßt erkennen, daß die Kommunikation mit einigem Overhead verbunden ist. Das größte Problem sind deshalb auch die Ausführungszeiten. Die Ergebnisse einiger Laufzeitemessungen sind im nächsten Abschnitt zusammengefaßt.

6.4.3 Zeitmessungen

Die im folgenden gemachten Angaben zu Ausführungszeiten wurde alle auf demselben Rechner (SPARCstation SLC, ohne Netzanbindung) ausgeführt. Die Messungen bezüglich des realisierten Communication System erfolgten anhand des `activate_contract`–Aufrufs des CC an den CM im lokalen Fall. Aufrufparameter ist der Name des ConTracts, Ergebnis ist der ConTract–Identifier (cid).

Folgende Messungen wurden ausgeführt:

1. Synchrone Aufruf
   Der CC ruft den CM mit der snychronen Variante des `activate_contract` Calls auf. Der CM wartet blockierend mit der `CS_lookup_message`–Funktion, ließt die Aufrufparameter mit der `CM_REC_activate_contract`–Funktion und sendet das Ergebnis mit der `CM_REP_activate_contract`–Funktion.

Gemessene Ausführungszeit pro Durchlauf:	20,22 ms

2. Asynchrone Aufruf
   Die Abfolge der Aufruf ist identisch bis auf die Entgegennahme des Ergebnisses durch den CC. Er wartet auf das Ergebnis mit der `CS_lookup_message`–Funktion und ließt es mit dem `CC_REC_activate_contract`–Aufruf. Im Unterschied zur synchronen Version wurden allerdings 100 Aufträge abgesetzt bevor die Ergebnisse gelesen wurden.

Gemessene Ausführungszeit pro Durchlauf:	18,31 ms

3. Aufruf mithilfe eine RPCs
   Zur Gegenüberstellung mit einem RPC wurden der Aufruf noch als solcher implementiert:

Gemessene Ausführungszeit pro Aufruf:	3,92 ms

Anmerkungen:

- Bei der synchronen Abarbeitung finden pro Ablauf auf jeden Fall zwei Prozeßwechsel statt (CC $\mapsto$ CM $\mapsto$ CC).
- Im asynchronen Fall werden zuerst die Sendeaufträge geschrieben und erst dann muß ein Prozeßwechsel zum CM erfolgen. Dies hat aber zur Konsequenz, daß die Mailbox alle Aufträge speichern muß. Um einen deutlich meßbaren Laufzeitunterscheid zu erhalten sollten deshalb viele Aufträge abgesetzt werden. Dadurch vergrösert sich der Aufwand beim Eintrag der Nachrichten (das Kopieren der Speicherseiten dauert länger), wodurch der Vorteil der geringern Anzahl von Prozeßwechsel nahezu kompensiert wird. Bei der Messungen wurde nur 100 Aufträge gesendet, deshalb unterscheiden sich die asynchrone und die synchrone Ausführungzeiten nur unwesentlich.

Zusammenfassung:

Die Implementierung des Communication System, mit denen obige Zeiten gemessen wurde, war die erste Version. D.h., es fand keine Überarbeitung unter Performance–Gesichtspunkten statt. Nach einer solchen Überarbeitung ist sicherlich mit einen Reduktion der Ausführungszeit zu rechnen. Die erste Version wurde auf Korrektheit ausgegelegt. Deshalb finden bei einem Ablauf (`CM|_activate_contract` bis `CC_REC_activatecontract` insgesamt 19 P–Operationen und 19 V-Operationen auf Semaphore statt. Jeder dieser Operationen dauert ca. 0,30 ms[8]. D.h., in Semaphore–Operationen werden insgesamt 11,21 ms verbraucht.

Nach Aussagen Herrn Prof. Bharat Bharagava, Department of Computer Science, Purdue University, USA — er sprach auf ihm Rahmen eines Informatik–Kolloquiums am IPVR — sind die Fuktionen im Zusammenhang mit *Shared Memory* und *Semaphore* schlecht implementiert. Er hat im Rahmen seines *RAID Distributed Databasesystem* ([4]) eine eigene Implementierung vorgenommen und dabei wesentlich kleiner Ausführungszeiten erhalten.

Bei Überarbeitung der Implementierung kann deshalb zwar mit der Reduzierung der Ausführungszeit gerechent werden, ob allerdings die Zeiten eines Remote Procedure Calls erreicht werden können, ist fraglich.

[8] V–Operation 0,31ms, P–Operation 0,28 ms.

Anhang A

Beispiel für ein ConTract–Skript

Das Beispiel beschreibt die kombinierte Buchung eines Fluges, Hotels und Mietwagens. Eine graphische Darstellung dieses ConTracts befindet sich auf Seite 7, Abbildung 1.2.

```
CONTRACT Business_Trip_Reservation

BEGIN_CONTEXT
 start, ziel, tag_hin, flugnummer, abflugzeit,
 ankunftszeit, kname, hname, averleih                :  STRING;
 taganz, sanzahl, knummer, bettanz, zinummer, index  :  INTEGER;
 betrag_flug, betrag_hotel, betrag_auto, limit       :  REAL;
END_CONTEXT

BEGIN_SCRIPT
 S1  : Flugdateneingabe ( OUT: abflugort     : start        : STRING,
                          OUT: zielort       : ziel         : STRING,
                          OUT: sitzanzahl    : sanzahl      : INTEGER,
                          OUT: abreisetag    : tag_hin      : STRING,
                          OUT: kundenname    : kname        : STRING,
                          OUT: kundennummer  : knummer      : INTEGER,
                          OUT: bettenanzahl  : bettanzahl   : INTEGER,
                          OUT: reisedauer    : taganz       : INTEGER,
                          OUT: hoechstbetrag : limit        : REAL,
                          OUT: zaehler       : index        : INTEGER
                        )
        EXIT_INVARINAT  ( EXECSQL select budget
                                  from   kunden
                                  where  name    = :kname
                                    and  budget >= :limit
                          ENDSQL )

        POLICY : check_revalidate;

 S2  : Flugverbindung_ermitteln
                        ( IN : abflugort     : start        : STRING,
                          IN : zielort       : ziel         : STRING,
                          IN : sitzanzahl    : sanzahl      : INTEGER,
                          IN : abreisetag    : tag_hin      : STRING,
                          OUT: flugkennung   : flugnummer   : STRING,
                          OUT: zeit_abflug   : abflugzeit   : STRING,
```

```
                      OUT: zeit_ankunft  : ankunftszeit : STRING,
                      OUT: flugkosten    : betrag_flug  : REAL
                    )

S3  : PARBEGIN
      S4 : Flugbuchung
                    ( IN : flugkennung   : flugnummer   : STRING,
                      IN : abreisetag    : tag_hin      : STRING,
                      IN : sitzanzahl    : sanzahl      : INTEGER,
                      IN : kundennummer  : knummer      : INTEGER
                    )

      S5 : Flugkosten_abbuchen
                    ( IN : flugkosten    : betrag_flug  : REAL,
                      IN : kundennummer  : knummer      : INTEGER
                    )

           ENTRY_INVARIANT ( EXECSQL select count(*)
                                     from   kunde
                                     where  kundennr = :knummer
                                       and  budget  >= :limit
                                       and  :limit  >= :betrag_flug
                              ENDSQL )

           EXIT_INVARIANT  ( EXECSQL select budget
                                     from   kunde
                                     where  kundennr = :knummer
                                       and  budget  >= :limit - :betrag_flug
                              ENDSQL )
      PAREND

S6  : IF (index : INTEGER   <= 3)
      S7  : SEQ
            S8  : Hotel_und_Autoverleih_festlegen
                    ( IN_OUT : zaehler      : index        : INTEGER,
                      IN     : bettenanzahl : bettanz      : STRING,
                      IN     : reisedauer   : taganz       : INTEGER,
                      OUT    : hotelname    : hname        : STRING,
                      OUT    : hotelkosten  : betrag_hotel : REAL,
                      OUT    : autoverleih  : averleih     : STRING,
                      OUT    : autokosten   : betrag_auto  : REAL
                    )

            S9  : Hotelbuchung_vornehmen
                    ( IN  : kundennummer  : knummer      : INTEGER,
                      IN  : hotelname     : hname        : STRING,
                      IN  : hotelkosten   : betrag_hotel : REAL,
                      OUT : zimmernummer  : zinummer     : INTEGER
                    )

                  ENTRY_INVARIANT ( EXECSQL select count(*)
                                            from   kunde
                                            where  kundennr = :knummer
                                              and  budget  >= :betrag_hotel
                                     ENDSQL )
```

```
            EXIT_INVARIANT  ( EXECSQL select budget
                                      from   kunde
                                      where  kundennr = :knummer
                                        and  budget  >=
                             :limit - (:betrag_flug + :betrag_hotel)
                              ENDSQL )

            POLICY : check_revalidate;

      ENDSEQ

      S10 : Autobuchung_vornehmen
                 ( IN  : kundennummer  : knummer      : INTEGER,
                   IN  : autoverleih   : averleih     : STRING,
                   IN  : autokosten    : betrag_auto  : REAL
                 )

            ENTRY_INVARIANT ( EXECSQL select count(*)
                                      from   kunde
                                      where  kundennr = :knummer
                                        and  budget  >= :betrag_auto
                              ENDSQL )

            EXIT_INVARIANT  ( EXECSQL select budget
                                      from   kunde
                                      where  kundennr = :knummer
                                        and  budget  >=
               :limit - (:betrag_flug + :betrag_hotel + :betrag_auto)
                              ENDSQL )

            POLICY : check_revalidate;
ELSE
      S11 : Flugstornierung
                 ( IN : flugkennung   : flugnummer BY S2  : STRING,
                   IN : abreisetag    : tag_hin           : STRING,
                   IN : sitzanzahl    : sanzahl           : INTEGER,
                   IN : flugkosten    : betrag_flug       : REAL,
                   IN : kundennummer  : knummer           : INTEGER
                 )

ENDIF

S12 : Buchungsdaten_ausgeben
                 ( IN : flugkennung   : flugnummer BY S2  : STRING,
                   IN : abreisetag    : tag_hin           : STRING,
                   IN : kundennummer  : knummer           : INTEGER,
                   IN : sitzanzahl    : sanzahl           : INTEGER,
                   IN : flugkosten    : betrag_flug       : REAL,
                   IN : hotelname     : hname             : STRING,
                   IN : zimmernummer  : zinummer          : INTEGER,
                   IN : bettanzahl    : bettanz           : INTEGER,
                   IN : hotelkosten   : betrag_hotel      : REAL,
                   IN : autoverleih   : averleih          : STRING,
                   IN : autokosten    : betrag_autol      : REAL
                 )
END_SCRIPT
```

```
BEDGIN_CONFLICT_RESOLUTION
 S5  : BEGIN
          S100  :  Kundendaten_korrigieren
                       (
                         IN : kundennummer  : knummer      : INTEGER
                       )
      END

 S9, S10  : BEGIN
           S101  : Kundenlimit_erhoehen
                       (
                         IN : kundennummer  : knummer      : INTEGER
                       )
           END
END_CONFLICT_RESOLUTION


BEGIN_TRANSACTIONS
   T1 (S4, S5)
   T2 (S8, S9, S10)

   DEPENDENCY ( T1 a|->b S2)
   DEPENDENCY ( T2 a|->b T2)
END_TRANSACTIONS
```

Anhang B

Abkürzungen der Komponenten

Die Komponenten des APRICOT System und ihre Abkürzungen faßt nachfolgende Tabelle zusammen.

Application	AP
ConTract Client	CC
ConTract Manager	CM
Compensation Manager	COM
Communication System	CS
Global Log Manager	LM
ConTract Processing Monitor	PM
Object Manager	OM
Step Computation Server	SS
Resource Manager	RM
Recovery Manager	RC
Transaction Manager	TM

Anhang C

Literaturverzeichnis

[1] Özapl Babaoğlu. Fault-Tolerant Computing Based on Mach. *acm press. Operating Systems Review*, 24(1):27–39, Jan. 1990.

[2] P. Bernstein, W. Emberton, V. Trehan. Transaction Processing, Databases and Fault Tolerant Systems: DECdta — Digital's Distributed Transaction Processing Architecture. Digital Technical Journal Vol. 3(1), Digital Equipment Corporation, Winter 1991.

[3] P. Bernstein, M. Hsu, B. Mann. Implementing Recoverable Requests Using Queues. *Proc. ACM SIGMOD*, May 1990. Digital Equipment Corporation.

[4] B. Bhargava, E. Mafla, J. Riedl. Communication in Raid distributed database. *Computer Networks and ISDN Systems*, March 1891.

[5] A. Birrell, B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):33–59, Feb. 1984.

[6] S. Ceri, G. Pelagatti. *Distributed Databases. Principles and Systems.* MacGraw-Hill Book Company, 1984. ISBN 0-07-010829-3.

[7] Digital Equipment Corporation. *Guide to VMS Programming Resources. VMS Version 5.0.* Part 3: Communication, April 1988. Order Number AA-LA57A-TE.

[8] Digital Equipment Corporation. *VMS I/O User's Reference Manual Part 1. VMS Version 5.0.* Part 7: Mailbox Driver, April 1988. Order Number AA-LA84A-TE.

[9] J.L. Eppinger, L.B. Mummert, A.Z. Spector. Camelot and Avalon - A Distributed Transaction Facility, 1991.

[10] J. Gray et al. A Straw Man Analysis of Probability of Waiting and Deadlock. *IBM Research Report RJ3066(38112), San Jose, California*, Feb. 1981.

[11] H. Garcia-Molina, K. Salem. Sagas. *Proc. ACM SIGMOD*, 1987.

[12] F. Haberhauer. *Entwurf und Implementierung von Mechanismen zur Überwachung der Verfügbarkeit lokaler und entfernter Prozesse.* Universität Stuttgart, Institut für Informatik, Juli 1985. Studienarbeit Nr. 435.

[13] IBM. *Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*, Dec. 1985.

[14] M.F. Kaashoeck, A. Tanenbaum, S.F. Hummel, H.E. Bal. An efficient reliable broadcast protocol. In *10th Int. Conference on Distributed Computing Systems*, S. 140–154, 1990.

[15] B. Liskov, L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. SIGPLAN '88. Conference on Programming Language Design and Implementation*, S. 260–267, Atlanta, Georgia, June 1988.

[16] Herausgeber P.C. Lockemann, J.W. Schmidt. *Datenbank–Handbuch.* Springer Verlag, Berlin Heidelberg, 1987. ISBN 0-387-10741-X.

[17] R. Lutz. *Verteilte Transaktionsverarbeitung mit Transaktionsbeziehungsnetzen.* Universität Stuttgart, Institut für Informatik, Sept. 1991. Diplomarbeit (in Vorbereitung).

[18] Sun Microsystems. *Network Programming Guide*, March 1990. Part Number: 800-3850-10.

[19] P.E. O'Neil. The Escrow Transaction Method. *ACM Transactions on Database Systems 11 (4)*, S. 405–430, Dec. 1988.

[20] P. Peinl, A. Reuter, H. Sammer. High Contention in a Stock Trading Database – A Case Study. *Proc. ACM SIGMOD*, 1988.

[21] A. Reuter. *ConTracts: A Means for Extending Control Beyond Transaction Boundaries.* Proc. 3rd Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1989.

[22] A. Reuter. *Implementation of Transaction Processing Operating Systems.* Morgan Kaufmann Publisher, 1991. First Draft.

[23] A. Reuter, H. Wächter. Neue Transaktionskonzepte. Technischer Bericht, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner IPVR, Breitwiesenstraße 20-22, W-7000 Stuttgart 80, 1991.

[24] K. Rothermel. A Communication Mechanism Supporting Actions. *Computer Networks and ISDN Systems*, 15:97–188, 1888. Elsevier Science Publisher B.V. (North-Holland.

[25] A. Tanenbaum. *Computer Networks.* Prentice-Hall International, second edition, 1989.

[26] D. Terry. Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments. Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, Feb. 1885.

[27] H. Wächter und A. Reuter. The ConTract Model. Institute for Parallel and Distributed High Performace Computers (IPVR), University of Stuttgart, Breitwiesenstrasse 20-22, W-7000 Stuttgart 80, Germany, 1991. Manuscript published in: A. K. Elmagarmid (ed.), Transaction Models for Advanced Database Applications, Morgan Kaufmann Publishers, 1991.

[28] P. Wahl. *Interpretative Abwicklung von Prädikat–Transitionsnetzen.* Universität Stuttgart, Institut für Informatik, Aug. 1991. Studienarbeit Nr. 1021.

[29] L.R. Walmer, M.R. Thompson. A Programmer's Guide to the Mach Environment. Technischer Bericht, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, 1989.

[30] L.R. Walmer, M.R. Thompson. A Programmer's Guide to the Mach System Calls. Technischer Bericht, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, 1989.

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

[32] X/OPEN. *Distributed Transaction Processing: The XA Spezification.* XOPEN, April 1990. Priliminary Specification XO/PRELIM/90/020.

[33] M. Young, D. Thompson, E. Jaffe. A Modular Architecture for Distributed Transaction Processing. In *USENIX Technical Conference, Dallas, Texas*, Winter 1991.