Protokollierungs- und Wiederanlaufverfahren

Masmanidis Ioannis

2. Juni 1992

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einführung in die Problemstellung

Rechnersysteme bestehen aus einer Vielzahl von Hardware- und Software-Komponenten, die zwangsläufig durch das Auftreten gewisser Ereignisse, aufhören zu funktionieren. Solche Ereignisse können zu unerwartetem Fehlverhalten des Systems führen. Aus diesem Grund sind im Laufe der Jahren viele Ansätze zur Bildung fehlertoleranter Systeme diskutiert und entwickelt worden. Das Ziel dabei war, dem Benutzer ein wohldefiniertes Verhalten des Systems im Fehlerfall zu garantieren bzw. ihm gewisse Fehler zu verbergen.

Der Entwurf und Implementierung fehlertoleranter Systeme ist keine triviale Aufgabe, da man die verschiedenen Systemaktivitäten steuern und kontrollieren muß, nicht nur wenn alle Komponeneten fehlerfrei arbeiten, sondern auch, wenn manche von diesen nicht mehr bzw. fehlerhaft funktionieren. Diese Aufgabe ist vor allem bei großen verteilten Anwendungen schwierig, weil es hier an wohldefinierten Verarbeitungskonzepten zur Steuerung, Koordination und Überwachung fehlt.

In der letzten Dekade hat man erkannt, daß Transaktionen ein sehr hilfreiches Mittel sind, um fehlertolerante verteilte Systeme zu realisieren. Das Transaktionskonzept bietet einen mächtigen Mechanismus zur Synchronisation und Koordination paralleler Abläufe und vorallem eine wohldefinierte Semantik zur Fehlerbehandlung an. Daher findet dieses Konzept immer mehr Berücksichtigung, wenn es darum geht, Systeme zur Modellierung verteilter Anwendungen zu entwickeln ([9, 6, 22]).

Dabei hat man aber erkannt, daß das klassische Transaktionskonzept gewisse Grenzen bei der Implementierung langlebiger Anwendungen in verteilten Umgebungen aufweist, sobald nicht nur einfache Daten, sondern beliebig strukturierbare Objekte verarbeitet werden sollen. Aus diesem Grund hat man durch mehrere Ansätze die Erweiterung des Transaktionkonzeptes angestrebt, so daß sein Nutzen sich über die Anwendungsbereiche der gegenwärtig existierenden konventionellen Datenbanksysteme erstreckt.

Durch einen solchen Ansatz im Rahmen des DFG-Schwerpunktprogramms *Objektbanken für Experten* ist das ConTract-Modell [25, 26] entstanden. Dieses Modell soll unter anderem einen Beitrag zur Entwicklung verallgemeinerter Transaktionskonzepte zum Bau und

Betrieb verteilter, fehlertoleranter Systeme leisten. Hauptmerkmal dieses Systemtyps ist es, langlebige und verteilte Anwendungen zuverlässiger auszuführen, als dies mit den gegenwärtig verfügbaren Systemen möglich ist.

Ein wichtiger Aspekt dabei ist die *forward-Recovery*. Für gewisse Anwendungen ist die *alles oder nichts* Semantik der Transaktions-Recovery nicht akzeptabel. Man kann z.B einem Benutzer nicht zumuten, den Vorgang einer Flugbuchung, der unter Umständen langwierig sein kann, zu wiederholen, nur weil beim Ausdrucken des Flug-Tickets das Terminal ausfällt. Vielmehr sollte man dem Benutzer die Möglichkeit geben, auf einem anderen Terminal lediglich den Druck zu wiederholen. Im Aspekt der *forward-Recovery* im Rahmen einer robusten Abwicklung im ConTract-Modell liegt der Schwerpunkt der vorliegenden Arbeit.

1.2 Aufgabenstellung

Im Rahmen des ConTract-Projekts im Institut für Parallele und Verteilte Höchstleistungsrechner der Fakultät Informatik an der Universität Stuttgart wurde ein prototypisches Laufzeitsystem zur interpretativen Abwicklung von Ablaufskripten realisiert. Diese Komponente soll nun um Mechanismen zur robusten Kontrollflußverwaltung erweitert werden.

In einem ersten Schritt sollen die Auswirkungen der möglichen Fehler, die in der prototypischen Umgebung des ConTract-Systems auftreten können, auf die ConTract-Verarbeitung untersucht und Vorschläge zu ihrer Behandlung angegeben werden.

Im zweiten Schritt soll der interne Verarbeitungszustand eines ConTracts im Falle einer Unterbrechung analysiert werden. Daraus soll ein Verfahren entwickelt werden, durch das der ConTract-Manager den Ausführungszustand eines ConTracts während der Kontrollflußabwicklung protokollieren und beim Wiederanlauf (während der *forward-Recovery*) wieder rekonstruieren kann.

Im letzten Schritt sollen die Alternativen skizziert werden, die für die Realisierung eines globalen Logs im APRICOTS existieren. Weiterhin sollen die Konsequenzen in bezug auf das für den ConTract-Manager entwickelte Protokollierungs- und Wiederanlaufverfahren diskutiert werden, die aus der Implementierung eines globalen Logs resultieren.

1.3 Aufbau der Arbeit

Einführende Begriffe, sowie die prototypische Architektur für das ConTract-Modell werden im nächsten Kapitel vorgestellt. Kapitel 3 skizziert die Mechanismen, die der ConTract-Manager zur Kontrollflußabwicklung implementiert. Anschließend wird die Vorgehensweise des ConTract-Managers während der Abwicklung eines ConTracts beschrieben.

Im Kapitel 4 werden die Ereignisse, die zu einem Fehlverhalten der Komponenten des ConTract-Systems führen können, und ihre Auswirkungen auf die ConTract-Verarbeitung analysiert. Kapitel 5 schlägt Maßnahmen zur Behandlung der auftretenden Fehler auf Systemebene vor.

In den Kapiteln 7 bis 9 wird ein Verfahren zum Protokollieren und Rekonstruieren des

Verarbeitungszustands eines ConTracts vorgestellt. Dieses Verfahren beruht auf der im Kapitel 6 vorausgegangenen Analyse der Zustände, die ein aktiver ConTract nach einem Systemfehler annehmen kann. Einige Überlegungen zur Realisierung eines globalen Log-Services ist das Thema, das im Kapitel 10 behandelt wird.

Kapitel 2

Das ConTract-Modell

2.1 Motivation

Das ConTract-Modell ist ein Ansatz, objektorientierte Konzepte mit dem Transaktionskonzept zusammenzuführen. Durch dieses Modell sollen langlebige Aktivitäten in verteilten Objektbanksystemen implementiert werden. Während das Prinzip der Datenabstraktion beim Objektorientierten Ansatz den Vorteil der Verarbeitung beliebig strukturierbarer Objekte als Einheit bietet, stellt das Transaktionskonzept anwendungsunabhängigen Mechanismen zur Synchronosation und Kontrolle von Arbeitsschritten zur Verfügung, die auf einen gemeinsamen Datenbestand operieren. Weiterhin garantieren Transaktionen die Konstistenz und Integration der Daten.

Wie schon in dem einleitenden Kapitel erwähnt worden, hat man dabei das klassische Transaktionskonzept erweitert, um die Nachteile zu beseitigen, die sich für die Verarbeitung in verteilten Objektbanksystemen aus der Atomaritäts- und Isolationseigenschaft ergeben.[1]

Was das ConTract-Modell neu in der verteilten Verarbeitung einführt ist:

- Man trennt die Beschreibung der Anwendungs- von den Kontrollaspekten. Während der Anwendungsprogrammierer nur den Kontrollfluß zu spezifizieren braucht, übernimmt das System sämtliche Synchronisations- und Kontrollaufgaben.
- Die Programmierung eines Vorgangs erfolgt in zwei Stufen: Die Spezifikation des Kontrollflußes (Skript) und die Kodierung der einzelenen Arbeitsschritten (Steps).
- Flexible Synchronisation durch Invarianten statt durch explizite Sperren [26].
- Forward-Recovery: Der Vorgang wird auch nach einem Fehler fortgesetzt– nötigenfalls auf einem anderen Rechner– statt zurückgesetz, wie es in den konventionellen Systemen der Fall ist.
- Falls das Beenden eines Vorgangs nicht mehr möglich ist, wird er in kontrollierter Weise in einem Anfangszustand zurückgebracht (Kompensation eines ConTracts).

[1] Die Nachteile des klassischen Transaktionkonzeptes werden in [11] ausführlicher beschrieben.

2.2 ConTracts

Ein ConTract stellt einen in sich abgeschlossenen Vorgang einer Anwendung dar. Die Programmierung eines ConTracts zerfällt wie schon im vorigen Abschnitt erwähnt wurde, in zwei Teile: Die Erstellung des ConTract-Skriptes und die Programmierung der einzelnen Arbeitsschritte. Im ConTract-Skript wird der Kontrollfluß eines Vorgangs niedergelegt. Man kann es mit dem Hauptprogramm einer konventionellen Programmiersprache vergleichen. Die Kontrollstrukturen, die zur Spezifikation des Kontrollflußes verwendet werden können, sind in [20] ausführlicher beschrieben. Hier verzichten wir auf eine Wiederholung der Beschreibung. Stattdessen erläutern wir die Rolle des Skriptes anhand eines einfachen Beispieles.

Die Abbildung 2.1 zeigt ein Beispiel-Skript, das den (vereinfachten) Vorgang einer Banküberweisung implementiert.

```
 ConTract Bank_Ueberweisung

  Begin_Context_Decl
    Konto_Nr1, Konto_Nr2, status : long;
    Betrag : double;
  End_Context_Decl

  Begin_Script
    S1 : Lese_Konto_und_Betrag (Out : kontonr1 : Konto_Nr1 : long,
                                Out : kontonr2 : Konto_Nr2 : long,
                                Out : betrag : Betrag : double);
    S2 :  ParBegin
          S3 : Abbuchung ( ....kontext_step_3...);
          S4 : Zubuchung ( ....kontext_step_4...);
          S5 : Eintragen_ins_Protokoll(....kontext_step_5...);
         ParEnd
    S6 : Quitieren (....kontext_step_6...);
End_Script
```

Abbildung 2.1: Ein vereinfachtes Beispiel für einen ConTract

Im Kontextdeklarationsteilteil, das durch die Schlüsselwörter *begin_context_decl* und *end_context_decl* geklammert ist, ist der Kontext deklariert, d.h. die globalen Objekte, die innerhalb des Skriptes verarbeitet werden. Im Anwendungsblock (begin_script, end_script) ist die Ablaufstruktur spezifiziert.

Die elementare Schritte (z.B Eingabe, Abbuchung, Zubuchung, etc.) sind die sogenannte Steps.[2] Ein Step ist die kleinste Berechnungseinheit in einem ConTract. Im Skript wird nur sein Aufruf spezifiziert. In der Kodierungs eines Steps liegt der eigentliche Aufwand der Anwendungsprogrammierung. Ein Step wird unabhängig von seiner Verwendung im Skript programmiert. Die Skript und die Step-Programmierung sind zwei von einander

[2] Auf die vollständige Deklaration der Steps wurde verzichtet. Nur für den Step S1 ist beispielsweise eine vollständige Deklaration seines Kontexts aufgeführt. Die genaue Spezifikation ist in [20] beschrieben.

unabhängige Vorgänge (zweistufiges Programmiermodell).

Der Vorgang der Überweisung wird wie folgt durch das Skript abgewickelt. Der Step S1 liest (aus dem Terminal) die Kontonummer des Auftraggebers, die Kontonummer des Empfängers und den Betrag, der überwiesen werden soll. Dann werden durch die Anweisung S2 parallel (spezifiziert durch das Schlüßelwort ParBegin) die Steps: S3 (Abbuchung des Betrages vom Konto des Auftraggebers), S4 (Zubuchung des Betrags auf das Konto des Empfängers) und S5 (Protokollierung des Vorgangs) ausgeführt. Am Ende wird das Überweisungsformular auf einen Drucker quittiert (Step S6).

2.3 Die APRICOTS-Architektur

APRICOTS stellt eine Abkürzung für die prototypische Implementierung des ConTract-Systems (a prototype implemention of the ConTract-System). In der Abbildung 2.2 sind die Komponenten, aus denen das APRICOT-System besteht.

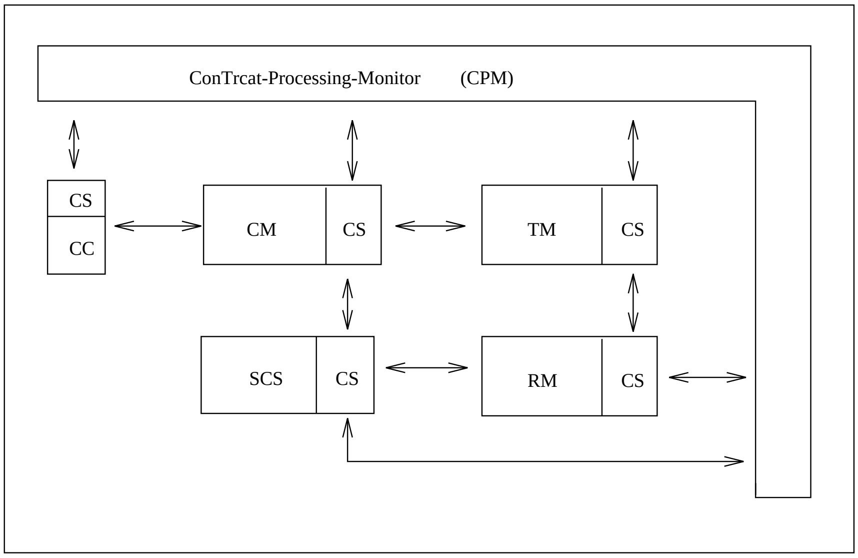

Abbildung 2.2: Die Komponenten des APRICOTS-Systems.

Der ConTract-Client (CC) realisiert die Schnittstelle des APRICOT-Systems zum Benutzer. Er stoßt unter anderem die Ausführung eines ConTracts an, veranlaßt die Suspendierung, Reaktivierung und Migration des ConTracts.

Der ConTract-Processing-Monitor (CPM) ist die Komponente, die die gesamte APRICOT-Konfiguration in einem Knoten überwacht.

Der ConTract-Manager (CM) ist die Komponente, die das ConTract-Skript intepretiert. Seine Hauptaufgabe ist die Steuerung und Überwachung der Abwicklung eines ConTracts. Er aktiviert und synchronisiert die Steps, anhand der Skript-Spezifikationen. Im ConTract Modell ist vorgesehen, daß ein ConTract-Manager mehrere ConTracts parallel abwickeln kann. In der gegenwärtigen Implementierung des APRICOTS wird angenommen, daß jeder ConTract-Manager nur einen ConTract verarbeitet. Dies setzen wir auch in dieser Arbeit voraus. Damit ist im weiteren der Begriff "Zustand des ConTract-Managers" gleichbedeutend mit dem Zustand des von ihm bearbeiteten ConTracts. [3]

Der Transaktions-Manager (TM) verwaltet die Transaktionen, überwacht deren Beziehungen und koordiniert das Commit-Protokol für die verteilte Transaktion.

Ein Step-Computation-Server (SCS) führt einen oder mehrere Steps aus. Auf jedem Knoten können mehrere Step-Computation-Server aktiviert sein.

Das Kommunikationssystem (CS) regelt den Nachrichtentransport zwischen den Komponenten. In der gegenwärtigen Implemmentierung des ConTract-System übernimmt das Kommunikationssystem das Scheduling.

Ein Ressource-Manager (RM) kann ein Objektverwalter sein oder jeder andere Dienst wie z.B ein Datenbank-System, File-System etc., der im System verfügbar ist.

Das Zusammenwirken der APRICOTS-Komponenten bei der Ausführung eines Steps ist im Abschnitt 3.2.2 ausführlich beschrieben.[4]

---

[3] Das im weiteren Ablauf dieser Arbeit beschriebene Protokollierungs- und Wiederanlaufverfahren Verfahren sind aber auch für den Fall anwendbar, daß ein ConTract-Manager mehrere ConTracts parallel abwickelt.

[4] Eine Ausführliche Beschreibung der Beziehung zwischen den APRICOTS-Komponenten und die physische struktur der APRICOTS-Prozesse findet man in [4, 17, 1]

Kapitel 3

Das Betriebsmodell im APRICOTS

Der ConTract-Manager fungiert als zentraler Koordinator der APRICOTS-Komponenten während der Abwicklung eines ConTracts. Aus diesem Grund ist der ConTract-Manager das Hauptobjekt auf den sich die erste Überlegungen und Bemühungen richten, um das bei der Implementierung des ConTract-Modells angestrebte Ziel zu ereichen, nähmlich langlebige und verteilte Anwendungen zuverlässiger auszuführen als es mit den gegenwärtig verfügbaren Systemen möglich ist.

In einem ersten Schritt werden die Vorgehensweise des ConTract-Managers zur Abwicklung des Kontrollflußes und die dabei verwendete Mechanismen näher betrachtet. Im weiteren werden die Fehler analysiert, die während der ConTract-Verarbeitung auftreten können. Diese Analyse soll dann der Ausgangspunkt zur Entwicklung der Verfahren zur robusten und zuverlässigen Kontrollflußabwicklung sein.

An dieser Stelle muß bemerkt werden, daß auf eine ausführliche Beschreibung des ConTract-Managers verzichtet wird, da dies nicht Gegenstand dieser Arbeit ist. Es wird lediglich auf die Kontrollflußabwicklung näher eingegangen. Für eine detaillierte Beschreibung des ConTract-Managers wird der Leser auf [33] verwiesen.

3.1 Der ConTract-Manager

3.1.1 Struktur des ConTract-Managers

Im Abschnitt 2.3 ist schon erwähnt worden, daß die Steuerung und Überwachung der Ausführung eines ConTracts im Aufgabenbereich des ConTract-Managers liegt. Dies beinhaltet zum einem die Interpretation des Skriptes, d.h der er aktiviert und synchronisiert die Steps gemäß des im Skript spezifizierten Kontrollflußes. Dabei nimmt er die Meldungen über den Ausgang der Ausführung der Steps auf, wertet sie aus und führt entsprechend die nächsten Aktionen zur Fortsetzung der Kontrollflußabwicklung aus.
Zum anderen führt er die Interaktion mit dem ConTract-Client, d.h er bietet eine Schnitt-

stelle an, die der Anwendung bzw. dem Bezutzer die Steuerung und Überwachung der ConTract-Abwicklung ermöglicht.

Die Arbeitsweise des ConTract-Managers kann man als ereignisorientiert bezeichnen. Er steuert die Contract-Veararbeitung, indem er auf eingetretene Ereignisse reagiert. Das Feuern eines Events, die *step_finished*-Meldung eines Step-Computation-Servers, die Zustandsänderung einer Transaktion, eine Nachricht vom ConTract-Client etc, stellen solche Ereignisse dar. Der ConTract-Manager bietet den anderen APRICOTS-Komponenten eine Schnittstelle an, deren einzelne Funktionen die Reaktionen auf die eingetretene Ereignisse implementieren. Die Schnittstelle des Contract-Managers ist in [33] ausführlicher beschrieben.

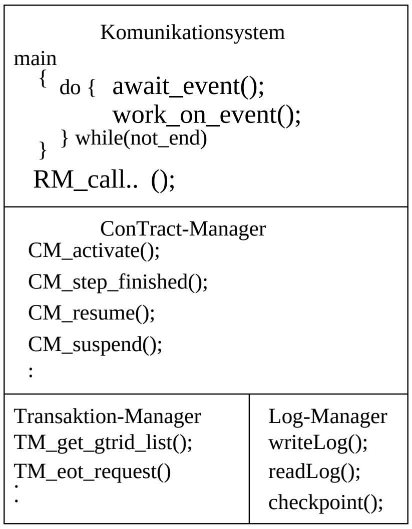

Abbildung 3.1: Die Struktur des ConTract-Managers

Der Aufbau des ConTract-Managers entspricht den in [17] beschriebenen Aufbau einer APRICOTS-Komponenete. Der ConTract-Manager ,wie auch jede andere Komponente, wird vom Kommunikationssystem initialisisert, d.h die *main*-Routine wird vom Kommunikationssystem implementiert. Zusätzlich werden auch Module des Transaktions-Managers und des Log-Managers dazugebunden. Dadurch wird eine Interprozeß-Kommunikation zwischen diesen Komponenten vermieden, was zur Minimierung der Betriebskosten entscheident beiträgt. Abbildung 3.1 stellt den Aufbau des ConTract-Managers graphisch dar. [1]

3.1.2 Interpretation des Contract-Skriptes

Für die Interpretation des Skriptes stellt der ConTract-Manager zwei Basisdienste zur Verfügung: Die Koordination der Step-Ausführung und die Anbindung an das Transaktionsystem.

[1]Vergleiche mit Abbildung 2.2.

Der ConTract-Manager erzeugt für jeden auszuführenden Step eine Step-Instanz, die er durch den zuständigen Step-Computation-Server ausführen läßt. Das Ergebnis der Step-Ausführung analysiert er und führt die geforderten Aktionen aus (siehe Abschnitt 3.2.2). Treten während der ConTract-Abwicklung Konflikte auf, leitet er die im entsprechenden ConTract-Skript angegebenen Lösungschritte ein.

Um atomare Berechnungsschritte zu definieren, bildet der ConTract-Manager Steps bzw. Step-Gruppen auf Transaktionen ab. Transaktionsereignisse wertet der ConTract-Manager aus, um Entscheidungen über den weiteren Ablauf zu treffen. Der Einsatz von Transaktionen wird im nächste Abschnitt näher betrachtet.

3.1.3 Steuerung des Kontrollflußes

Der ConTract-Manager verwendet zur Steuerung des Kontrollflußes zwei Mechanismen : Den Event-Trigger-Mechanismus und Transaktionen.

Event-Trigger Mechanismus

Events sind Ereignisse, die zur Aktivierung und Synchronisation der Steps dienen. Ein Step wird nur dann aktiviert, wenn sein Start-Ereignis triggert (auch feuert). Ein Event triggert, wenn er eine bestimmte Anzahl von "Tokens" erhalten hat. Die Anzahl der "Tokens", die ein Event erhalten muß, um zu triggern, wird bei der Definition dieses Events, d.h bei der Übersetzung [20] des ConTract-Skriptes festgellegt.

Jeder Step, der ausgeführt wird, sendet ein Token an seinen Terminierungs-Event. Dabei darf ein Step mehrere Terminierungs-Events besitzen, die durch Bedingungen mit diesem Step gekoppelt sind. Nach der Ausführung des Steps werden diese Bedingungen ausgewertet. Liefert die Auswertung einer Bedingung den Wert "TRUE", wird an das zugehörige Event ein Token gesendet.

In der Abbildung 3.2 ist das Beispiel aus dem Abschnit 2.2 in Form eines Petri-Netzes dargestellt. Daraus kann man die Ablaufstruktur des ConTracts erkennen: S1 sendet, sobald er ausgeführt ist, ein Token an e1. e1 triggert und aktiviert seinerseits den Step S2. Die Steps S3, S4 und S5 müssen synchronisiert werden, sodaß S5 erst dann aktiviert wird, wenn alle drei Steps S3, S4 und S5 beendet sind. Dies wird dadurch erreicht, daß S3, S4 und S5 jeweils ein Token an das Start-Ereignis (e5) von S6 senden. Dieses Ereignis muß drei Tokens erhalten, um triggern zu können.

Zur Implementierung des Event-Trigger Mechanismus, bildet der ConTract-Manager den Kontrollfluß, der im ConTract-Skript beschrieben wird, auf ein Prädikat Transitionsnetz ab. Zur Abwicklung des Kontrollflußes interpretiert der ConTract-Manager dieses Netz.

Transaktionen

Transaktionen sind der Basismechanismus, auf den sich die fehlergeschützte ConTract-Abwicklung stützt. Der ConTract-Manager definiert Transaktionen, um Steps als atomare

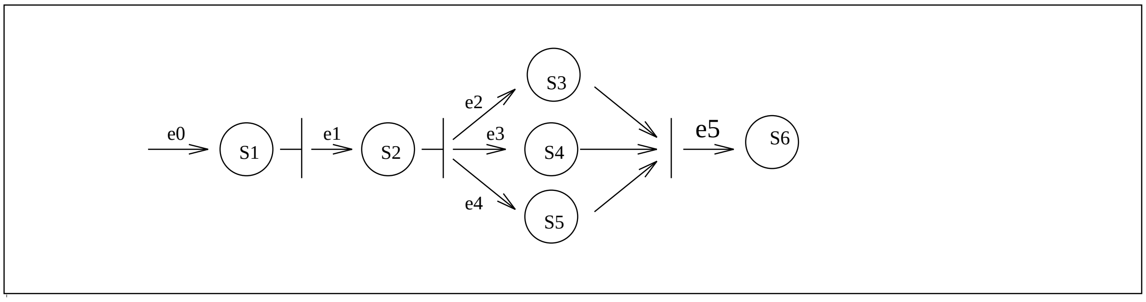

Abbildung 3.2: Der ConTract Überweisung in der Form eines Petri-Netzes.

Berechnungsschritte auszuführen. Jedem Step, der ausgeführt werden soll, wird eine Transaktion zugeordnet, die die Ausführung dieses Steps klammert.

Falls der Step erfolgreich ausgeführt wird, kann die Step-Transaktion im Zustand "committed" überführt werden. Damit können die Ergebnisse der Step-Ausführung Systemfehler überleben. Konnte die Ausfürung eines Steps nicht erfolgreich beendet werden, muß die Step-Transaktion abgebrochen und damit auch ihre Zwischenergebnisse zurückgesetzt werden. Damit erhält jeder Step die für die Transaktionen geltenden ACID-Eigenschaften.

Der Skript-Programmierer kann auch Transaktionen definieren, die mehrere Steps bzw. Transaktionen umfassen. Damit können atomare Einheiten innerhalb eines Skripts definiert werden, die aus mehreren Steps bzw. Teiltransaktionen bestehen.

Im Beispiel Skript könnte eine Transaktion definiert werden, die die Steps S3 und S4 umfaßt. Dies könnte im Transaktionsteil des Skripts [20] wie folgt erfolgen:

```
BEGIN_TRANSACTIONS
T1(S3,S4,S5)
END_TRANSACTIONS
```

Damit hat der Skriptprogrammierer erreicht, daß alle drei Steps als eine atomare Aktion (Transaktion) ausgeführt werden. Kann einer dieser Steps nicht erfolgreich ausgeführt werden, dann müssen die andere zwei ebenfalls zurückgesetzt werden.

Um die Koordination der Commit-Abwicklung einer aus mehreren Teiltransaktionen bestehenden Transaktion zu ermöglichen, müssen die Teiltransaktionen - nach einer erfolgreichen Ausführung der entsprechenden Steps- in einem Zwischenzustand überführt werden, aus dem sowohl der Zustand "committed" als auch der Zustand "aborted" zu erreichen sind.

Bezogen auf das Beispiel-Skript heißt es: Wenn S3 erfolgreich beendet ist, dann darf seine Step-Transaktion kein "commit" durchführen, bevor der Ausgang der Ausführung der Steps S4 und S5 bekannt ist. Sonst kann passieren, daß diese Step-Transaktion zurückgesetzt werden muß, weil die Ausführung eines der Steps S4 und S5 fehlgeschlagen ist. Dies soll aber nach dem Transaktions-Paradigma nicht möglich sein.

Um solche Fälle zu vermeiden, werden Teilransaktionen zunächst in den Zustand "prepared" überführt. Sobald eine Entscheidung anhand der Zuständen der restlichen Teil-Transaktionen getroffen werden kann, erfolgt der Übergang im Zustand "committed" bzw. "aborted".

Näheres über die Behandlung der Transaktionen und ihre Beziehungen im ConTract-Modell kann der Leser aus [19, 28] erfahren.

3.1.4 Dynamischer Zustand eines ConTracts

Um dem Leser das Verständnis der Beschreibung der Kontrollflußabwicklung sowie der nachfolgender Fehleranalyse, des Protokollierungs- und Wiederanlaufverfahren zu erleichtern, werden in diesem Abschnitt einige Begriffe aus der Welt des ConTract-Manager erläutert.

Im Abschnitt 3.1.3 ist erwähnt worden, daß der ConTract-Manager eine Art Prädikat Transitionsnetzes anhand der Skriptspezifikationen aufbaut, um die Bedingungen zur Aktivierung und Terminierung der Steps zu implementieren. Dieses Prädikat Transitionsnetz wird in eine Struktur aufgenommen, die wir im weiteren *Script* [33] nennen. Diese Struktur ist vom ConTract-Skript, der den Kontrollfluß eines ConTracts beschreibt zu unterscheiden. Die Struktur *Script* stellt den im ConTract-Skript spezifizierten Kontollfluß in der Form eines Netzes dar. Diese Struktur enthält im einzelnen folgende Daten:

- Die **Events**. Der Zustand der Events, d.h die Anzahl der empfangenen bzw. erwarteten Tokens, bestimmt den Zeitpunkt seines Feuerns und damit den Zeitpunkt der Aktivierung seiner Nachfolger-Steps. Diese Daten korrekt zu verwalten, ist der Schlüssel zur zuverlässigen Abwicklung des ConTracts.

- Die **Steps**. Für jeden im ConTract-Skript aufgeführten Step verwaltet der ConTract-Manager einen Eintrag, der außer der bei der Deklaration dieses Steps angegebenen Informationen wie z.B Namen, Kontext, Step-Nummer, etc. auch seine "history" gespeichert ist. Dazu gehört die Versionsnummer, die Anzahl der Aktivierungen sowie die Anzahl der abgeschlossenen Ausführungen dieses Steps.
  Jede Ausführung eines Steps ist durch ihre Versionsnummer, die bei jeder Aktivierung dieses Steps erzeugt (inkrementiert) wird, von den vorausgegangenen bzw. nachfolgenden Aktivierungen desselben Steps unterscheidbar. Die Versionierung jeder Step-Ausführung ist zur Realisierung der Kontextverwaltung und der Kompensation [2] notwendig. Die Anzahl der Aktivierungen und die Anzahl der abgeschlossenen Ausführungen eines Steps dienen zur Aufzeichnung seiner "Geschichte". Sie werden bei jeder Aktivierung bzw. nach jeder Ausfürung aktualisert.

- Die **aktiven Step-Instanzen**. Für jede Aktivierung eines Steps erzeugt der ConTract-Manager eine neue Step-Instanz . Er belegt sie mit den für die Ausführung des Steps benötigten Informationen und sendet sie an einem Step-Computation-Server zur Ausführung. Eine Kopie dieser Step-Instanz fügt er in die Struktur *Script* ein. Jede Step-Instanz wird nach dem Empfang der Meldung über ihren Ausführungsende aus dem *Script* wieder gelöscht.

[2] Die Kontextverwaltung wird in [26, 5] ausführlich beschrieben.

- Weitere **Zusatzdaten**, mit deren Hilfe das Prädikat Transitionnetzes aufgebaut und interpretiert werden kann. Diese Daten werden nicht näher diskutiert, da sie für die weitere Beschreibung nicht relevant sind.

An dieser Stelle führen wir folgende Begriffe ein, die im weiteren Verlauf dieser Arbeit öfters erwähnt werden:

- **Dynamische ConTract-Daten-Strukturen**: Dieser Begriff umfaßt alle in der Struktur *Script* enthaltenen Daten und die Variable, in denen der aktuelle interne Zustand des ConTracts gehalten wird.
- **Zustand der dynamischen ConTract-Daten**: Die aktuelle Wertebelegung der dynamischen ConTract-Daten-Strukturen.
- **Verarbeitungs bzw. Ausführungs Zustand eines ConTracts**: Enthält alle aktivierten, beendeten und zur Ausführung bereiten Step-Instanzen.
- **(Dynamischer) ConTract-Zustand**: Oberbegriff, der den Zustand der dynamischen ConTract-Daten und den Verarbeitungs- bzw. Ausführungs- Zustand eines ConTracts beinhaltet.

3.2 Kontrollflußabwicklung

Das Verständnis der Vorgänge, die sich während der Ausfürung eines ConTrcats abspielen, ist die Voraussetzung, um die Analyse der Auswirkungen der im APRICOTS auftretender Fehler auf den Verarbeitungszustand eines ConTracts und die Beschreibung des für den ConTract-Manager entwickelten Protokollierungs- und Wiederanlaufverfahrens mitverfolgen zu können. Bevor wir auf die weiteren Einzelheiten über das Vorgehen des ConTracts-Managers bei der Verarbeitung eines ConTracts näher eingehen, geben wir in der Abbildung 3.3 eine Überblick der Zustände, die ein ConTract bzw. ConTract-Manager während der Kontrollflußabwicklung annehmen kann. Damit soll das Verständnis der in den nachfolgenden Abschnitte beschriebene Vorgänge erleichtert werden.

3.2.1 Aktivierung eines ConTracts

Ein ConTract-Client kann durch das ACTIVATE-Kommando einen ConTract-Manager beauftragen, einen bestimmten ConTract zu aktivieren. Der Name des auszuführenden ConTracts wird als Parameter dieses Kommandos an den ConTract-Manager übergeben.

Als Reaktion auf dieses Kommando setzt der ConTract-Manager den ConTract im Zustand INIT und führt die Initialisiserung der internen Datenstrukturen, die er für die Abwicklung eines ConTracts benötigt. Anschließend liest er aus den Zwischencode Relationen die Zwischendarstellung [20, 33] des Kontrollflusses, der im zugehörigen ConTract-Skript spezifiziert wurde. Damit baut der ConTract-Manager das Prädikat Transitionsnetz (d.h die Struktur Script) auf.

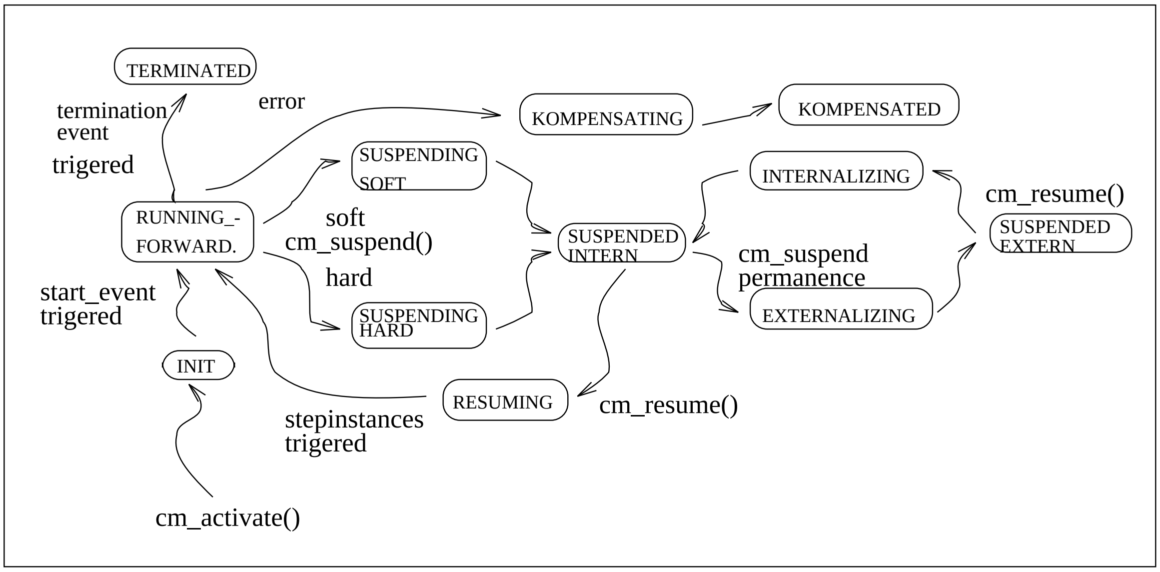

Abbildung 3.3: Das Zustandsübergangsdiagramm eines ConTracts bzw. ConTract-Managers

Als nächstes setzt der ConTract-Manager den ConTract im Zustand RUNNING_FORWARD und triggert das Start-Event des ConTracts. Damit beginnt die Abwicklung des Kontrollflusses. Die weitere Vorgehensweise des ConTract-Managers dabei, wird in den nächsten Abschnitte beschrieben.

3.2.2 Kontrollflußabwicklung im Zustand RUNNING_FORWARD

Die Abwicklung eines ConTracts beginnt mit der Triggerung seines Start-Events. Anschließend werden die Steps berechnet, deren Start-Event das ausgelöste Ereignis ist. Diese Steps müssen aktiviert werden.
Dazu erzeugt der ConTract-Manager für jeden dieser Steps eine Step-Instanz. Für jede erzeugte Step-Instanz wird vom Transaktions-Manager eine eindeutige Transaktions_Id (GTRID) angefordert (*get_gtrid*-Nachricht). Damit installiert der Transktions-Manager für jede angeforderte Transktions-Id eine neue Transaktion. Unter dem Schutz dieser Transaktion soll die entsprechende Step-Instanz ausgeführt werden. Die neu erzeugten Step-Instanzen werden aktiviert, indem sie an die Komponente für das Scheduling von ConTract-Steps durch den Funktionsaufruf *schedule_stepinstance_list* übergeben werden. Das Kommunikationssystem ermittelt seinerseits für jeder Step-Instanz einen Server, der den Kode des entsprechenden Steps implementiert (Step Computation Server) und sendet ihm den Auftrag für die Ausführung der Step-Instanz.

Das Ende der Ausführung einer Step-Instanz wird dem ConTract-Manager durch eine *step_finished*-Nachricht mitgeteilt, die vom Step-Computation-Server über das Kommunikationssystem gesendet wird. Diese Nachricht enthält zusätzlich einen Fehlerkode, der anzeigt, ob die Ausführung der Step-Instanz erfolgreich war oder nicht und gibt gegebenfalls Auskunft über die Fehlerquelle. Anhand dieses Fehlerkodes trifft der ConTract-Manager Entscheidungen über die weitere Abwicklung des Kontrollflusses.

Fortsetzung der Contract-Verarbeitung nach erfolgreicher Step-Ausführung

Ist der Step erfolgreich ausgeführt, fordert der ConTract-Manager den Transaktions-Manager durch einen asynchronen *EOT_request*-Aufruf die entsprechende Step-Transaktion zu beenden, d.h in dem Zustand "committed" zu überführen. Sobald der Transaktions-Manager das Zweiphasen-Commit- Protokoll abgewickelt hat, informiert er den ConTract-Manager durch einen *EOT_request_reply*- Aufruf über den neuen Zustand der Transaktion (siehe Abbildung 3.4).

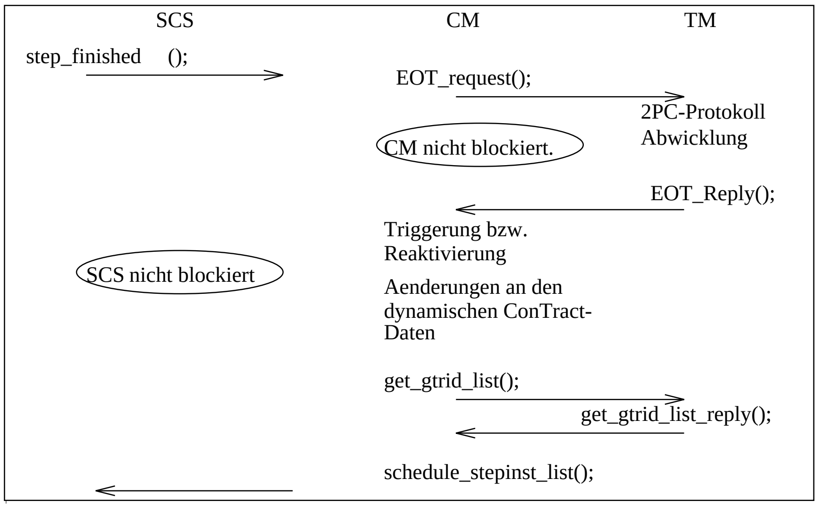

Abbildung 3.4: Bearbeitung einer step_finished-Nachricht im Zustand RUNNING_FORWARD

Falls die Abwicklung des Zweiphasen-Commit Protokolls erfolgreich verlaufen ist (die Transaktion wurde in den Zustand "committed" überführt), stößt der ConTract-Manager die Triggerung der beendeten Step-Instanz an. Dazu sendet er an die Terminierungs-Events des entsprechenden Steps jeweils ein Token. D.h die Anzahl der empfangenen Tokens dieser Events wird jeweils um eins erhöht. Das kann dazu führen, daß ein oder mehrere Events feuern.

Für jeden Event, der gefeuert hat, berechnet der ConTract-Manager die Nachfolger-Steps, die aktiviert werden müssen. Wie bei der Triggerung des Start-Events des ConTracts, wird für jeden dieser Steps jeweils eine neue Step-Instanz erzeugt und eine neue Transaktion installiert. Alle neu erzeugte Step-Instanzen werden in die sogenannte Step-Instanz-Liste eingefügt. Nachdem alle eingetretene Ereignisse bearbeitet worden sind, übergibt der ConTract-Manager die Liste der auszuführenden Step-Instanzen an das Kommunikationssystem, das seinerseits das Scheduling dieser Step-Instanzen durchfürt. Davor wird die beendete Step-Instanz aus den dynamischen ConTract-Daten gelöscht.

War die Abwicklung des Zweiphasen-Commit Protokolls für die Step-Transaktion nicht erfolgreich, setzt der Transaktions-Manager die Transaktion im Zustand "aborted", d.h die Resultate der Ausführung der beendeten Step-Instanz werden zurückgesetzt. Dann teilt er dem ConTract-Manager das Zurücksetzen der Step-Transaktion durch den *EOT_request_reply*-Aufruf mit. An dieser Stelle entscheidet der ConTract-Manager, ob der Step erneut ausgeführt werden darf. Dazu prüft er ab, ob die maximale Anzahl der erlaubten Wiederholungen für die Ausführung eines Steps überschritten ist.

Falls das nicht der Fall ist, fordert er für dieselbe Step-Instanz eine neue Transaktions_Id vom Transaktions-Manager an und erhöht die Anzahl der Wiederholungen der neuen Step-Instanz um eins. Dann aktuallisiert der ConTract-Manager die Datenstruktur Script (Versionierung des entsprechenden Steps, neue Transaktions-Id) und anschließend stößt er erneut durch einen *schedule_stepinst_list* - Aufruf die Ausführung der Step-Instanz. Der einzige Unterschied zur Bearbeitung einer Step-Instanz, deren Step-Transaktion im Zustand "committed" überführt werden konnte, ist, daß hier keine Triggerung der beendeten Step-Instanz angestoßen wird. Es wird lediglich dieselbe Step-Instanz unter dem Schutz einer neuen Transaktion reaktiviert.

Falls die maximale Anzahl der erlaubten Wiederholungen überschritten ist, benachrichtigt der ConTract-Manager den ConTract-Client, der den ConTract gestartet hat und leitet die Kompensation des ConTracts ein.

Ist die Step-Transaktion der beendeten Step-Instanz im Zustand "prepared" überführt worden, wird diese Step-Instanz nicht weiter verarbeitet. Es folgt weder eine Triggerung noch eine Reaktivierung. In diesem Fall definiert der ConTract-Manager durch eine Nachricht (define_ta_event) an den Transaktions-Manager ein Ereignis das triggern soll, sobald während der weiteren Verarbeitung des ConTracts diese Step-Transaktion einen der Endzustände "committed" oder "aborted" erreicht hat. Durch die Triggerung dieses Ereignisses soll der ConTract-Manager über den neuen Zustand dieser Transaktion informiert werden. Dann kann er entsprechend dem neuen Zustand dieser Transaktion die Triggerung bzw. Reaktivierung der Step-Instanz anstoßen.

Fortsetzung der ConTract-Verarbeitung nach fehlerhafter Step-Ausführung

Konnte eine aktivierte Step-Instanz nicht erfolgreich ausgeführt werden, liefert die *step_finished*-Nachricht einen Fehlerkode, der den Fehler genauer spezifiziert. Anhand des zurückgelieferten Fehlerkodes entscheidet der ConTract-Manager, ob er weiter versuchen soll, den Step auszuführen oder nicht. In beiden Fällen fordert er den Transaktions-Manager wiederum durch einen *EOT_request*-Aufruf, die Step-Transaktion zurückzusetzen.

Erfordert der von der *step_finished*-Nachricht zurückgelieferte Fehlerkode die Ausführung des Steps zu wiederholen, prüft auch an dieser Stelle der ConTract-Manager, ob die maximale Anzahl der erlaubten Wiederholungen überschritten ist und verhält sich entsprechend. Andernfalls wird die Kompensation des ConTracts eingeleitet.

3.2.3 Der Vorgang der Suspendierung eines ConTracts

Hat der ConTract-Manager eine *Suspend*-Nachricht empfangen, setzt er seinen internen Zustand entsprechend dem gewünschten Modus der Suspendierung auf SUSPENDING_SOFT bzw. SUSPENDING_HARD.

Im Zustand SUSPENDING_SOFT wartet er auf das Beenden der Ausführung der aktiven Step-Instanzen. Jedesmal wenn er eine *step_finished*-Nachrich erhält, fordert er entsprechend dem gelieferten Fehlerkode den Abbruch bzw. das Beenden der entsprechenden Step-Transaktion (*EOT_request*-Aufruf). Trifft die Antwort des Transaktions-Managers auf diese Anforderung ein, wird die entsprechende Step-Instanz in die sogenannte *Suspend-Liste* eingefügt, d.h zur späteren Verarbeitung vorgemerkt, ohne daß ihre Triggerung angestoßen wird.

Haben alle aktiven Step-Transaktionen den Zustand "committed", "aborted", oder "prepared" erreicht, geht der ConTract-Manager in den Zwischenzustand SUSPENDED_INTERN über.

Im Zustand SUSPENDING_HARD fordert der ConTract-Manager vom Transaktions-Manager, alle aktive Step-Transaktionen abzubrechen. Damit wird die Ausführung der aktiven Step-Instanzen beendet und von den Step-Computation-Server an den ConTract-Manager gemeldet. Trifft die Meldung über den Abbruch einer Step-Instanz ein, wird diese Step-Instanz in die *Suspend-Liste* angehängt. Sobald die *Suspend-Liste* vollständig ist, setzt der ConTract-Manager den ConTract in den Zustand SUSPENDED_INTERN.

Falls durch die Suspend Nachricht eine *permanente* Suspendierung gefordert wurde, sichert der ConTract-Manager den Zustand der dynamischen Daten (interner Zustand INTERNALIZING), setzt den ConTract im Zustand SUSPENDED_EXTERN, gibt den für die dynamische Daten reservierten Speicher frei und bricht die Ausführung ab.

3.2.4 Fortsetzung eines Suspendierten ConTracts

Die Bearbeitung eines suspendierten ConTracts kann wieder durch eine *resume*-Nachricht aufgenommen werden. Der Empfang einer solchen Nachricht führt zum Übergang des ConTracts in den Zustand RESUMING_FORWARD.

Falls der ConTract-Manager die *resume*-Nachricht im Zustand SUSPENDED_INTERN empfangen hat, setzt er die Kontrollflußabwicklung fort, indem er den ConTract im Zustand RESUMING überführt, und die Triggerung bzw. die Neuaktivierung der Step-Instanzen aus der Suspend-Liste anstößt. Dazu prüft er für jede Step-Instanz den Zustand ihrer Transaktion ab.

Ist die Step-Transaktion im Zustand "committed" erfolgt die Triggerung dieser Step-Instanz. Step-Instanzen, deren Transaktion im Zustand "aborted" ist, bereitet der ConTract-Manager zur Triggerung vor, während Step-Transaktionen im Zustand "prepared" nicht weiter behandelt werden. An dieser Stelle muß man folgendes herausheben: Die Step-Instanzen, die während der Triggerung einer Step-Instanz aus der Suspend-Liste neu erzeugt werden, werden nicht sofort aktiviert. Der ConTract-Manager sammelt sie zunächst in einer Liste und erst wenn alle Step-Instanzen in der Suspend-Liste abgearbeitet sind, stößt er ihre

Aktivierung an. Damit *wirkt die Triggerung bzw. Reaktivierung aller Step-Instanzen aus der Suspend-Liste auf den aktuellen ConTract-Zustand wie die Triggerung einer einzelnen Step-Instanz* aus.
Unmittelbar vor der Aktivierung der auszuführenden Step-Instanzen setzt der ConTract-Manager den ConTract im Zustand RUNNING_FORWARD.

Wird die Reaktivierung aus dem Zustand SUSPENDED_EXTERN gefordert, baut der ConTract-Manager seine interne Datenstrukturen - ähnlich wie im Zustand INIT - auf und rekonstruiert (interner Zustand INTERNALIZING) den Zustand, den er bei der Externalisierung gesichert hat. Dabei wird die Suspend-Liste wieder aufgebaut und der Zustand des ConTracts auf SUSPENDED_INTERN gesetzt. Aus diesem Zustand wird die Ausführung des ConTracts wie oben beschrieben fortgesetzt.

Kapitel 4

Fehlermodell

Im vorigen Kapitel wurde der ConTract-Manager und seine Vorgehensweise bei der Abwicklung eines ConTracts beschrieben. Um Verfahren zu entwickeln, durch die dieser Vorgang gegen Fehler geschützt bzw. nach einem Fehler fortgesetzt werden kann, werden in diesem Kapitel die möglichen Ereignisse analysiert werden, die zur Beinträchtigung der Ausführung eines ConTracts führen könnten.

4.1 Fehlerarten

In diesem Abschnit werden die Fehler aufgezählt, die in einem verteilten System auftreten können.

Eingabefehler. Falsche Eingaben des Benutzers sind Fehler, die am häufigsten auftreten.

Anwendungsbedingte Fehler treten während der Ausführung von Programmschritten auf, aus denen die Anwendungsfunktionen implementiert sind. Solche Fehler führen dazu, daß gewisse Funktionen nicht vollständig, fehlerhaft oder überhaupt nicht ausgeführt werden.

Transaktionsfehler treten auf, wenn die Anwendung den Abbruch einer Transaktion fordert. Das kann z.B vorkommen wenn ein Ressource-Manager, der an einer Transaktion teilnimmt, seine Arbeit aus verschiedenen Gründen z.B Zugriff auf ein gesperrtes Objekt nicht fortsetzen kann und daher den Abbruch der Transaktion fordert.

*Prozeßausfälle.*Durch Ausnahmesituationen kann es vorkommen, daß ein oder mehrere Prozesse abstürzen. Das hat zur Folge, daß eine oder mehrere Komponenten ihre Dienste nicht weiter anbieten können.

Ein *Kommunikationsfehler* liegt vor, wenn Nachrichten zwischen zwei Komponenten nicht ordnungsgemäß transportiert werden. Man kann dabei mehrere Fälle unterscheiden: a) Eine Nachricht kommt beim Empfänger nicht an. b) Der Empfänger empfängt eine Nachricht, die er nicht bearbeiten kann, weil diese Nachricht während der Übertragung manipuliert worden ist. c) Eine Nachricht wird dupliziert. d) Nachrichten überholen sich. e) Eine noch schwierigere Situation liegt vor, wenn das Netzwerk partitioniert wird. In diesem Fall bilden sich Gruppen von Prozessen, wobei die Prozesse innerhalb einer Gruppe miteinander kommunizieren können, aber keine Kommunikation zwischen Prozessen verschiedener Gruppen statfinden kann.

Ein Knotenausfall führt dazu, daß die Berechnung auf einem Knoten abrupt beendet ist. Dabei kann man zwischen einem temporären und einem permanenten Knotenausfall unterscheiden. Im Falle eines temporären Ausfalls ist der Knoten nur für eine bestimmte Zeit nicht verfügbar, während nach einem permanenten Ausfall der Knoten für immer aus dem System verschwindet.

Fallen mehrere Knoten aus dann stoppt die Verarbeitung auf einem großen Teil evtl. auf dem gesamten System. Kommen dabei viele permanente Knotenausfälle vor, dann spricht man von einer "Katastrophe". Dies ist ein sehr schwieriger Fall weil viele Daten, die auf diese Knoten gespeichert waren, in der Zukunft nicht mehr erreichbar sind.

Gerätefehler sind ebenfalls eine Ursache dafür, daß Teile des nichtflüchtigen Speichers verloren gehen.

4.2 Klassifizierung der Fehler

Um die Analyse der im vorigen Abschnitt beschriebenen Fehler zu erleichtern, nehmen wir eine weitere Klassifizierung vor, indem wir diese Fehler aus einer anderen Sicht betrachten: Wir charakterisieren sie durch die Menge an Informationen bzw. Daten, die nach ihrem Auftreten verloren geht. Dementsprechend können wir folgende Fehler-Arten unterscheiden:

- Fehler, die *nicht den Verlust des Hauptspeichers* herbeiführen. Hierzu gehören die Eingabefehler, die Anwendungsfehler, die Kommunikationsfehler und die Prozeßabstürze.
- Fehler, die den *Verlust des Hauptspeichers* zur Folge haben. Temporäre Knotenausfälle und Geräteausfälle gehören in dieser Klasse.
- Fehler, die zur *Zerstörung des nichtflüchtigen Speichers* führen, wie z.B Gerätefehler und permanente Knotenausfälle.
- Fehler, deren Folge der *Verlust des stabilen Speichers* ist. Der stabile Speicher wird durch die redundante Speicherung der Daten auf mehreren nichtflüchtigen Speichermedien realisiert. Mehrere temporäre bzw. permanente Knotenausfälle sind die Hauptursachen für den Verlust des stabilen Speichers.

4.3 Eine Beispiel-Konfiguration

Die Auswirkungen der Fehler werden an einer einfachen Beispiel-Konfiguration verdeutlicht. Sie wird in der Abbildung 4.1 dargestellt.

Auf den Knoten Odysseus und Penelope bearbeiten die ConTract-Manager CM1 und CM2 jeweils eine Überweisung (Überweisung_1 bzw. Überweisung_2). Auf dem Knoten Odysseus ist die komplette APRICOTS-Umgebung installiert, in der eine Überweisung durchgeführt

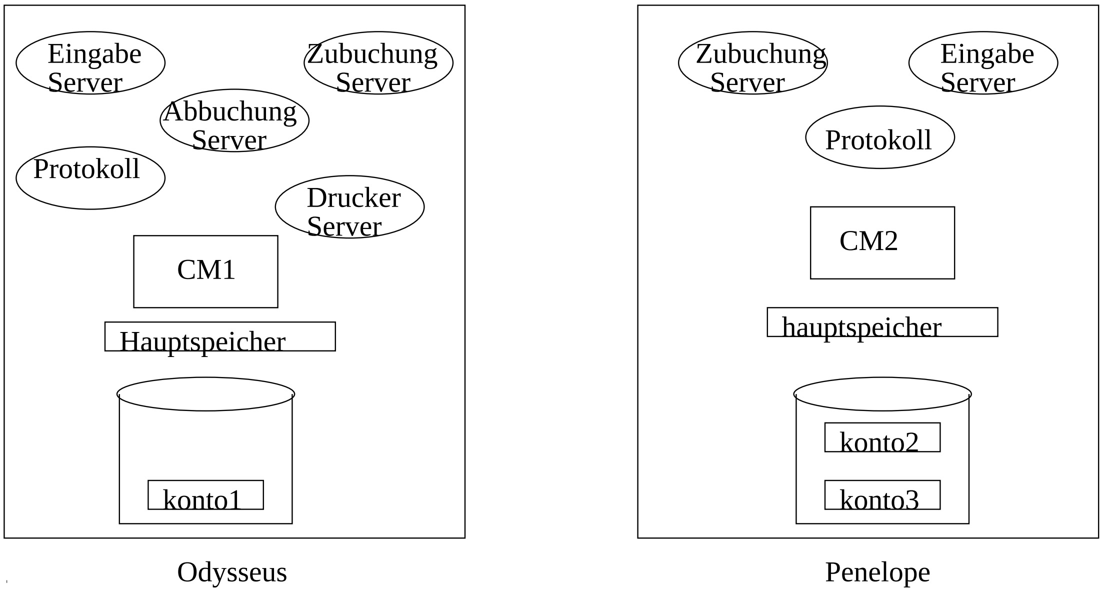

Abbildung 4.1: Beispiel-Konfiguration

werden kann, während auf den Knoten Penelope nur die Step-Computation-Server Zubuchung, Eingabe und Protokollierung existieren. Beide ConTract-Manager bearbeiten eine Überweisung deselben Autraggebers (konto1) an zwei verschiedenen Empfänger (konto2, konto3).

4.4 Auswirkungen der Fehler auf die ConTract-Verarbeitung

4.4.1 Fehler ohne Verlust des Hauptspeichers

Um im weiteren die Betrachtungen zu vereinfachen, kann man ohne Beschränkung der Allgemeinheit annehmen, daß Eingabefehler ein spezieller Fall der Anwendungsfehler sind.

Auswirkungen anwendungsbedingter Fehler

Anwendungsbedingte Fehler können dazu führen, daß ein oder mehrere Steps nicht erfolgreich ausgeführt werden können. In unserem Beispiel kann ein solcher Fall die Ablehnung einer Abbuchung sein, falls der Auftreggeber sein Kreditlimit überschritten hat. Der ConTract-Manager kann in solchen Fällen den ConTract nicht weiter verarbeiten, weil er kein anwendungsspezifisches Wissen besitzt, um Maßnahmen zur Behebung dieser Fehler ergreifen zu können. Stellt die Anwendung keine Lösungen zur Verfügung, muß der ConTract-Manager den ConTract kompensieren, falls aufeinander folgende Versuche, diese Steps auszuführen, fehlschlagen (siehe Abschnitt 3.2.2).

Auswirkungen der Transaktionsfehler

Transaktionsfehler veranlassen den Transaktions-Manager die betroffene Transaktion (hier Step-Transaktion) abzubrechen. Step S3 wird z.B abgebrochen, weil das Konto des Auftraggebers gesperrt ist. Der Step-Computation-Server meldet in diesem Fall dem ConTract-Manager die Step-Ausführung als gescheitert. Hier muß der ConTract-Manager ebenfalls den ConTract kompensieren, falls aufeinander folgende Versuche diesen Step auszuführen fehlschlagen.

Auswirkungen der Kommunikationsfehler

Im Abschnitt 3.2.2 ist das Kommunikations-Protokoll beschrieben, das die APRICOTS-Komponenten zur Ausführung eines Steps verfolgen. Wie sich Kommunikationsstörungen während dieses Vorgangs auf die Contract-Abwicklung auswirken, zeigen wir am Beispiel der Kommunikation zwischen dem ConTract-Manager und einen Step-Computation-Server.

Erhält der Step-Computation-Server die Nachricht des ConTract-Manager zur Ausführung des Steps nicht, führt er keinen Step aus. Der ConTract-Manager wartet aber auf die entsprechende *step_finished*-Meldung des Step-Computation-Servers, da aus seiner Sicht der Step ausgeführt werden sollte. Diese Meldung wird aber nie eintreffen, da der Step-Computation-Server eine solche Meldung nur am Anschluß an eine Step-Ausführung sendet. Das hat zur Folge, daß der ConTract-Manager ohne weitere Maßnahmen blockiert wird. Dieselbe Situation tritt ein, wenn der Step-Computation-Server einen Step ausgeführt hat, aber seine *step_finished*-Meldung beim ConTract-Manager nie ankommt.

Ein weiterer Fall, der die Auswirkungen der Kommunikationsfehler auf die ConTract-Abwicklung deutlich zeigt, sind duplizierte Nachrichten. Wird ein *schedule_step*-Aufruf dupliziert, dann kann der Step mehrmals ausgeführt werden. Wird z.B. der Auftrag des ConTract-Managers eine Abbuchung auszuführen dupliziert, dann wird das Konto des Auftraggebers mit dem doppelten Betrag belastet. Umgekehrt könnte eine duplizierte *step_finished*-Meldung dazu führen, daß der ConTract-Manager die Triggerung einer Step-Instanz doppelt durchführt. Das kann zur Folge haben, daß einige Steps mehrfach aktiviert werden.

Auswirkungen der Prozeßabstürze

Es gibt zwei Variationen, wie sich Prozeßausfälle auf die Verarbeitung eines ConTracts auswirken können. a) Der ConTract-Manager-Prozeß ist abgestürzt. b) Andere Step-Computation-Server- bzw. Ressource-Manager-Prozesse sind ausgefallen.

Im ersten Fall wird die ConTract-Abwicklung unterbrochen. Die Hauptspeicherstrukturen des ConTract-Managers sind nicht mehr zugreifbar. Das hat zur Folge, daß man den dynamischen Zustand des ConTracts nicht mehr rekonstuieren kann. Aus diesem Grund ist weder die Fortsetzung noch die Kompensation des ConTracts möglich. Im zweiten Fall kann der ConTract-Manager seine Arbeit fortsetzen, andere Komponenten aber, die für die Ausführung der Steps benötigt werden, sind nicht mehr funktionsfähig. Der ConTract-

Manager kann den ConTract nur dann fortsetzen, wenn im System andere Koponenten vorhanden sind, die dieselben Dienste wie die ausgefallenen anbieten und mit der Ausführung dieser Steps beauftragt werden können. Sind solche Komponenten nicht vorhanden, dann kann der ConTract-Manager den ConTract kompensieren aber auch nur unter der Voraussetzung, daß im System funktionsfähige Step-Computation-Server bzw. Ressource-Manager gibt, die die Kompensation-Steps ausführen können. Ist weder eine Kompensation noch eine Fortsetzung des ConTracts möglich, dann ist der ConTract-Manager blockiert.

Blockiert ist der ConTract-Manager auch im Falle, daß gewisse Komponenten ausgefallen sind, während sie Steps eines ConTracts bearbeiten. Der ConTract-Manager wird keine Step-finished-Nachrichten Meldung über den Ausgang der Ausführung dieser Steps erhalten können, um die ConTract-Abwicklung fortzusetzten.

Wir verdeutlichen diese Situationen an unserer Beispiel-Konfiguration. Stürtz der CM1-Prozeß ab, dann ist die Bearbeitung der Überweisung_1 unterbrochen. Die Daten-Strukturen des CM1, die den Zustand des ConTracts Überweisung_1 beschreiben sind nicht mehr zugreifbar. Das bedeutet, man kann diesen ConTract weder fortsetzen noch kompensieren, falls seine Zustandsdaten nur im flüchtigen Hauptspeicher gehalten werden.

Fällt der Server Zubuchung auf dem Knoten Odysseus aus, während der Step1 bearbeitet wird, dann führt der Zubuchungs-Server auf den Knoten Penelope den Step S4. Fällt dagegen der Server Abbuchung aus, dann kann der CM1 den Step S3 nicht mehr ausführen. Er kann aber den ConTract kompensieren, da die dafür benötigte Eingabe-, Zubuchungs- und Protokoll-Server verfügbar ist.

Wenn der Drucker- und der Abbuchungs-Server abstürzen, nachdem S3 ausgeführt worden ist aber noch bevor der CM1 den Step S5 aktiviert hat, dann ist der ConTract blockiert. Der CM1 kann weder den Step S5 ausführen lassen noch die bisher geleistete Arbeit kompensieren. Dafür fehlen die benötigten Server.

Falls der Abbuchungs-Server ausfällt, während er den Step S3 ausführt, dann wird der CM1 nie eine *step_finished*-Meldung von diesem Server erhalten können. Daher wird er den Step S5 nicht aktivieren können.

4.4.2 Auswirkungen des Verlustes des Hauptspeichers

Durch einen Knotenausfall werden alle auf dem Knoten aktive APRICOTS_Prozesse unkontrolliert beendet. Man kann hierbei differenzieren: a) Es fällt ein Knoten aus, auf dem der ConTract-Manager läuft. b) Es fällt ein Knoten aus, auf den Step-Computation-Server bzw. Ressource-Manager Teile eines ConTracts bearbeiten, aber nicht der ConTract-Manager Prozeß läuft.

Im Fall a) wird der ConTract unterbrochen und kann nicht mehr fortgesetzt bzw. kompensiert werden. Im Fall b) kann der ConTract-Manager den ConTract unter der Voraussetzung weiter verarbeiten, daß Ersatzkomponenten für die ausgefallene gibt und die Objekte, die auf den ausgefallenen Knoten gespeichert sind, über andere Knoten erreichbar sind. Sonst bleibt der ConTract-Manager blockiert, bis die ausgefallenen Komponenten wieder funktionsfähig sind bzw. Ersatzkomponenten erzeugt werden.

Fällt der Knoten Odysseus aus der Beispiel-Konfiguration aus, ist die Bearbeitung der Überweisung_1 unterbrochen. Der CM2 kann seine Arbeit fortsetzen, dazu braucht er aber den Drucker-Server. Den ConTract Überweisung_2 zu kompensieren ist er aber auch nicht in der Lage, weil der Server für die Abbuchung fehlt.

4.4.3 Auswirkungen des Verlustes des nichtflüchtigen Speichers

An dieser Stelle wird eine weitere Voraussetzung getroffen, um die weitere Analyse der Fehler zu erleichtern. Wir nehmen an, daß das Auftreten eines Gerätefehlers den totalen Ausfall des zugehörigen Knotens verursacht ("failfast"-Module [1]). Damit schließen wir die Gerätefehler auch in den Begriff permanente Knotenausfälle ein.

Auch im Falle eines permanenten Knotenausfalls werden die Prozesse auf den ausgefallenen Knoten abrupt beendet. Das bedeutet, daß es auch hier zu einer Unterbrechung oder Blockierung eines oder mehrerer ConTracts kommen kann. Dabei kann ein ConTract auch dann blockiert werden, wenn alle zur Ausführung seiner Steps benötigten Komponenten verfügbar sind. Der Grund dafür kann der Verlust der Objekte sein, die auf den ausgefallenen Knoten gespeichert sind.

In der Beispiel-Konfiguration könnte der Verlust des Knoten Penelope eine solche Situation herbeiführen. Dann könnte nämlich der CM1 die Überweisung_1 weiter verarbeiten, da alle dafür benötigten Server funktionsfähig sind. Die Ausführung des Steps S4 würde aber scheitern, da auf das Konto des Empfängers (konto2) nicht mehr zugegriffen werden kann. Das bedeutet, der ConTract muß kompensiert werden. Voraussetzung dafür ist auch hier die Verfügbarkeit aller benötigten APRICOTS-Komponenten und Objekte. Wäre z.B auch konto1 auf dem Knoten Penelope gespeichert, dann wäre auch die Kompensation der Überweisung_1 nicht durchführbar.

ConTracts, die durch permanente Knotenausfälle unterbrochen sind, können Teilergebnisse auf anderen Knoten hinterlassen. Damit könnte die Konsistenz der Daten verletzt werden, insbesondere wenn weitere aktive ConTracts die Teilergebnisse weiter verarbeiten. Fällt z.B der Knoten Penelope aus, nachdem die Abbuchung erfolgt ist, dann ist das Konto1 schon um den überwiesenen Betrag vermindert, ohne daß der Vorgang korrekt abgeschlossen ist. Wird ein weiterer Vorgang auf diesem Konto durchgeführt (z.B Zinsberechnung), dann liefert er weitere falsche Ergebnisse.

[1] Siehe [12] Kapitel " Fault Tolerance"

Kapitel 5

Fehlerbehandlung

In diesem Kapitel werden ansatzweise Maßnahmen diskutiert, die man zur Behandlung der im vorigen Kapitel angegebenen Fehler ergreifen kann. Weil die Betrachtung aller Fehlerarten den Rahmen dieser Arbeit sprengen würde, muß die Menge der hier betrachteten Fehler reduziert werden, in dem wir folgende Tatsachen in Betracht gezogen werden:

- Gewisse Fehler, können von anderen Systemkomponenten oder Basisdiensten auf einer tieferen Ebene behandelt werden.
- Manche Fehler kommen in der Realität äußerst selten vor. Ihre Auswirkungen sind unvorhersehbar. Diese Fehler können nur durch besondere Maßnahmen - im allgemeinen nur durch manuellen Eingriff- behandelt werden, die wir hier nicht betrachten wollen.

Damit können wir für die weitere Diskussion folgende Voraussetzungen treffen:

- Kommunikationsfehler bleiben dem ConTract-Manager sowie den restlichen APRICOTS-Komponenten verborgen. Sie können von den Basisdiensten des Kommunikationssystems erkannt und behoben werden. Im einzelnen bedeutet das: Die APROCOTS-Komponenten bearbeiten nur korrekte Nachrichten. Duplizierte bzw. fehlerhafte Nachrichten werden vom Kommunikationssystem erkannt und verworfen bzw. korrigiert. Alle Nachrichten werden geliefert, es sei denn, der Empfänger existiert nicht mehr. Im folgenden setzen wir voraus, daß Nachrichten in der richtigen Reihenfolge geliefert werden. Partitionierungen sind sehr seltene Ereignisse und werden hier nicht betrachtet.
- "Katastrophen" treten in der Realität sehr selten auf. Zu ihrer Behandlung, soweit sie möglich ist, können bereits entwickelte Verfahren, wie z.B. Wiederanlauf aus einer Archivkopie eingesetzt werden.
- Transaktionsfehler werden nicht separat betrachtet. War die Ursache für den Abbruch einer Transaktion temporärer Natur z.B kurzzeitiges Sperren eines Objektes, dann kann sie durch eine Wiederholung der Ausführung, so wie es im Abschnitt 3.2.2 beschrieben wurde, behandelt werden. Andernfalls kann man annehmen, daß der

Grund für ihren Abbruch ein Fehler ist, der in eine der Klassen gehört, die hier behandelt werden.

5.1 Anforderungen an die Fehlerbehandlung

Die Maßnahmen für die Behandlung im ConTract-Modell werden von der Konsisteneigenschaft dieses Modells diktiert. Reuter und Wächter haben sie in [26] wie folgt formuliert:

Ein ConTract terminiert unter (System-) Garantie in endlicher Zeit und in einem korrekten Endzustand.

Diese Eigenschaft stellt im einzelnen folgende Anforderungen an die Fehlerbehandlung:

- Ein Contract soll im Fehlerfall nicht stoppen, sondern fortgesetzt werden, sobald es der Systemzustand erlaubt.
- Ein bereits aktivierter ConTract sollte auf keinen Fall in einem undefinierten Zustand enden oder aus dem System verschwinden.
- Falls es unmöglich ist, einen aktivierten ConTract in endlicher Zeit zu beenden, dann muß man diesen ConTract kompensieren.

Diese Anforderungen bestimmen auch die Ziele, die die im nächsten Abschnit angegebenen Behandlungsmaßnahmen verfolgen.

5.2 Behandlung der anwendungsbedingten Fehler

Ein zentralen Ansatz bei der Implementierung des ConTract-Modells ist, die Anwendungs- von den Kontrollaspekten zu entkoppeln [26]. Diese Entkopplung ist durch den ConTract-Manager realisiert. Seine Aufgabe ist, den in einem Skript beschriebenen Kontrollfluß zuverlässig abzuwickeln. Dazu bearbeitet er sein Prädikat Transitionsnetz, ohne die Semantik der einzelnen Schritte (Steps) zu kennen.

Er kann eine begrenzte Anzahl von Fehlerkodes bearbeiten, die lediglich über den Ausgang einer Step-Ausführung Auskunft geben. Er ist nicht in der Lage anwendungsbedingte Fehler bzw. Fehlerkodes zu bearbeiten.

Eine Möglichkeit anwendungsbedingte Fehler zu behandeln, ist, ihre Behandlung in der Kontrollflußabwicklung im Skript durch entsprechende Verzweigungen zu formulieren. Es ist aber unpraktisch, in jedem Skript alle Fehler zu behandeln, die in einer umfangreichen Anwendung auftreten können. Aus diesem Grund wäre die Realisierung einer Komponente, die anwendungsspezifisches Wissen zur Behandlung anwendungsbedingter Fehler besitzt, von Vorteil. Man könnte sich eine Komponente vorstellen, die anwendungsbedingte Fehlercodes analysieren kann und entsprechend die erforderlichen Maßnahmen einleitet. Nennen wir diese Komponente anwendungsspezifischen Recovery-Manager.

Wenn man einen anwendungsspezifischen Recovery-Manager vorsieht, dann muß man ein Protokoll entwickeln, das seine Interaktion mit dem ConTract-Manager regelt. Man könnte sich folgenden Ablauf vorstellen: Der ConTract-Manager erhält eine step_finished-Meldung, die zusätzlich einen Fehlerkode liefert. Ist der Step erfolgreich ausgeführt, setzt der ConTract-Manager die ConTract-Abwicklung fort. Andernfalls prüft der CM, ob ein anwendungsspezifischer Fehlerkode vorliegt. Falls nicht, führt er seine eigene Fehlerbehandlung (siehe Abschnit 3.2.2) durch. Falls ein anwendungspezifischer Fehlercode geliefert wurde, beauftragt er den anwendungspezifischen Recovery-Manager mit der Fehlerbehandlung. Dieser führt anhand des vom ConTract-Manager übergebenen Fehlerkodes die erforderlichen Maßnahmen durch und meldet ihren Ausgang dem CM. Ist die Fehlerbehandlung erfolgreich verlaufen, kann der ConTract-Manager den Kontrollfluß weiter abwickeln, falls nicht, kann er anhand der Meldung des anwendungspezifischen-Recovery-Managers den Step reaktivieren, oder den ConTract kompensieren. Eine weitere Möglichkeit wäre, den ConTract zu suspendieren, bis die anwendungsbedingte Störung behoben ist.

5.3 Behandlung von Prozeßausfällen

Das Thema Zuverlässigkeit und Verfügbarkeit von Prozessen ist ausführlich in der Literatur [12] behandelt. Dabei sind mehrere Verfahren zur Bildung von hoch zuverlässigen und verfügbaren Prozessen beschrieben worden. Die Realisierung von Prozeßpaaren ist dabei der zentrale Ansatz, auf den sich die meisten Verfahren stützen.

Auf Prozeßpaare kann man auch für die Behandlung der Prozeßausfälle in der APRICOTS-Umgebung ansetzen. Dabei wäre es aber unpraktisch, für jeden APRICOTS-Prozeß ein Prozeßpaar vorzusehen. Einerseits weil man mit Performance-Einbußen rechnen muß, da umso niedriger ist die Leistungfähigkeit dieses Systems je mehr Prozesse auf dieses System laufen. Anderseits wäre der Entwurf und die Implementierung jeder einzelnen Komponenten viel aufwendiger, da Prozeßpaare zu es keine triviale Aufgabe ist Prozeßpaare zu implementieren.. Aus diesem Grund ist es sinnvoller, eine zentrale Komponente vorzusehen, die die restlichen APRICOTS-Prozesse überwachen soll. Diese Komponente sollte man dann als Prozeßpaare implementieren.Diese Komponente realisiert der ConTract-Processing-Monitor.

Auf jeden Knoten existiert ein ConTract-Processing-Monitor, der die gesamte APRICOTS-Konfiguration auf diesem Knoten überwacht und verwaltet. Die Aktivierung jedes weiteren APRICOTS-Prozesses, soll dann durch diesen Prozeß erfolgen. Der ConTract-Processing-Monitor sendet dann auf jeden Prozeß, den er aktiviert hat, in regelmässigen Abständen Nachrichten, durch die er den Zustand dieses Prozesses abfragt. Hat eine Komponente nach Ablauf eines bestimmten Zeitraums nicht auf seine Nachricht reagiert, kann er annehmen, daß diese Komponente nicht mehr funktionsfähig ist. Er muß dann die Recovery dieser Komponente anstoßen. Die ConTract-Processing-Monitore auf den verschiedenen Knoten können sich gegeseitig überwachen, um Ausfälle zu erkennen.

Weil im Falle eines Ausfalles eines CPM auch seine Hauptspeicher-Strukturen und damit die Informationen über die aktuelle Konfiguration verloren gehen, muß der CPM diese Informationen stabil speichern. Das bedeutet daß, auf jeden Knoten eine Tabelle vorhanden sein soll, in der die Informationen über die APRICOTS-Konfiguration stabil verwaltet werden.

Im Abschnitt 4.4.1 ist beschrieben worden, daß Prozeßfehler entweder zur Unterbrechung oder zur Blockierung eines ConTracts führen. Der Fall einer ConTract-Unterbrechung wird im Kapitel 6 ausführlich behandelt. Hier beschränken wir uns auf die Bemerkung, das die Erkennung des Ausfalls eines ConTract-Managers und seine Reaktivierung nicht ausreichen, um einen ConTract fortzusetzen. Der ConTract-Manager muß auch in der Lage sein, den Zustand des abgebrochenen ConTracts zu rekonstruieren und die "forward"-Recovery durchzuführen.

Im folgenden betrachten wir nur den Fall, in dem ein oder mehrere Step-Computation-Server bzw. RM ausgefallen sind.

Wir unterscheiden hierbei zwei weitere Fälle: a) Die Komponenten sind ausgefallen, während sie Steps eines ConTract bearbeiten. b)Die ausgefallene Komponente haben zum Zeitpunkt des Ausfalls keine Steps bearbeitet, aber werden benötigt, um weitere Steps abzuwickeln.

Im ersten Fall wird der ConTract-Manager blockiert, da er auf die Nachricht step_finished einer Komponente wartet, die aber nicht funktionsfähig ist. Da in diesem Fall die Ausführung des Steps als gescheitert betrachtet werden muß, kann der ConTract-Manager versuchen, andere Komponenten mit der Ausführung dieser Steps zu beauftragen. Dies kann der ConTract-Manager aber nur dann tun, wenn er das Scheitern der Step-Ausführung erfahren kann. Eine Möglichkeit dazu wäre, daß der ConTract-Prozessing-Monitor den (bzw. jeden) ConTract-Manager über den Ausfall der Step-Computation-Servern informiert. Weil aber der ConTract-Manager keine Kenntnis darüber hat, welche Steps von welchen Step-Computation-Server bearbeitet werden, ist dies nicht sinnvoll. Sinnvoller wäre, daß die Komponente, die das Scheduling durchführt, den Ausfall erkennt und dem ConTract-Manager das Scheitern der Step-Ausführung durch eine step_finished-Nachricht meldet, die sie selbst generiert. Dann könnte der ConTract-Manager die Ausführung des Steps durch eine andere Komponente veranlassen. Der Scheduler-Prozeß kann den Ausfall einer Komponenten entweder durch den ConTract-Processing-Monitor erfahren oder durch einen "timeout"-Mechanismus erkennen, den er selbst implementiert.

Die zweite Variante, die zur Blockierung des ConTracts führt, kann man wie folgt behandeln: Falls die Fortsetzung der ConTract-Abwicklung durch den Ausfall der Komponenten nicht bzw. nicht innerhalb einer bestimmten Zeit möglich ist, kann der ConTract-Manager den ConTract kompensieren. Kann die Kompensation aufgrund dieses Ausfalls auch nicht abgeschloßen werden, dann könnte der ConTract-Manager die Situation einer obersten Kontrollinstanz z.B.dem System-Administrator melden und sich suspendieren, bis die ausgefallenen Komponenten wieder betriebsfähig sind oder neue erzeugt werden. Dann kann der Systemadministrator den ConTract-Manager reaktivieren, um den ConTract weiter zu verarbeiten.

Wichtig bei Prozeßausfällen ist, daß der ausgefallene Prozeß (Komponente), nach seiner Reaktivierung die Objekte, die er verwaltet, in einen transaktionskonsistenten Zustand überführen kann, bevor er seine Dienste zur Verfügung stellt.

5.4 Behandlung eines temporären Knotenausfalls

Temporäre Knotenausfälle können in der Regel wie die Prozeßausfälle behandelt werden, da sie sich im allgemeinen ähnlich auf die ConTract-Abwicklung auswirken. Weil dabei auch der Contract-Prozessing-Monitor ausgefallen ist, muß man dafür sorgen, daß beim Wiederanlauf des Knoten, der ConTract-Prozessing-Monitor reaktiviert wird, der seinerseits die gesamte APRICOTS-Konfiguration rekonstruiert und die Aktivierung der abgebrochenen ConTracts anstößt. Voraussetzung hierfür ist, daß jede Komponente beim Wiederanlauf ihre eigene Recovery durchführen kann.

Fällt ein Knoten für längere Zeit aus,dann sollte das keine Auswirkungen auf die Verarbeitung anderer Knoten haben. Das bedeutet, falls ausgefallene Ressource-Manager für die Abwicklung auf anderen Knoten aktiver ConTracts benötigt werden, soll möglich sein, diese automatisch auf betriebsfähige Knoten zu erzeugen bzw. zu replizieren. Weiterhin sollte man dafür sorgen, daß auf den ausgefallenen Knoten befindliche Objekte weiter verfügbar sind, damit die Abwicklung anderer auf diese Objekte zugreifender ConTracts nicht beinträchtigt wird. Das bedeutet, daß die Replizierung der Daten auf mehreren Knoten erforderlich ist.

Weitere Maßnahmen zur Behandlung länger dauernder temporärer Knotenausfälle führen wir im nächsten Abschnitt im Zusammenhang mit permanenten Knotenausfällen auf.

5.5 Behandlung eines permanenten Knotenausfalls

Ein permanenter Knotenausfall soll - wie auch ein temporärer Knotenausfall - keine Auswirkungen auf die Verarbeitung auf anderen Knoten haben. Daher soll auch in diesem Fall, das automatische Erzeugen von Ersatzkomponenten möglich sein. Die Replizierung der Daten ist auf jeden Fall erforderlich.

Die unterbrochenen ConTracts sollen aber in einem definierten Endzustand terminieren (Konsistenzeigenschaft). Andernfalls können, wie im Abschnit 4.4.3 angedeutet wurde, inkonsistente Datenbestände produziert werden. Daher wäre es von Vorteil, wenn man die auf den ausgefallenen Knoten aktiven ConTracts auf einem anderen Knoten fortsetzen bzw. kompensieren könnte (Migration eines ConTracts [1]). Dazu muß man in der Lage sein, den Zustand dieser ConTracts auf anderen Knoten rekonstruieren zu können. Dies läßt sich durch Replizierung [8] der ConTract-Daten bzw. der Log-Daten und der Konfigurationstabelle des ConTract-Processing-Monitors erreichen.

Eine weitere Maßnahme, die man zur Behandlung permanenter aber in einigen Fällen auch temporärer Knotenausfälle ergreifen könnte, ist das automatische Rekonstruieren der gesamten APRICOTS-Umgebung eines Knotens auf einem Ersatzknoten. Das ist vorallem wichtig für Knoten, die eine spezielle Aufgabe im System bearbeiten. Dabei müssen zwei Voraussetzungen erfüllt sein. Erstens müssen die Informationen über die genaue Zusammensetzung der Konfiguration auf dem ausgefallenen Knoten vorhanden sein. Zweitens der Zustand der auf den ausgefallenen Knoten aktiven ConTracts bzw. gespeicherten Objekten

[1] Vergleiche mit [26]

muß rekonstruierbar sein. Diese zwei Forderungen geben Anlaß, hier ein weiteres mal auf die Notwendigkeit hinzuweisen, Maßnahmen zu treffen, die den Verlust von Daten -soweit es möglich ist- auszuschließen.

Das Kompensieren bzw. Fortsetzen unterbrochener ConTracts auf Ersatzknoten, sollte man auch im Falle eines längerdauernden temporären Knotenausfalls durchführen können.

5.6 Zusammenfassung

In den vorausgegangenen Abschnitten wurden verschiedene Ansätze diskutiert, um die im Kapitel 4 aufgeführten Fehler zu behandeln. Aus dieser Diskussion kann man folgende Punkte herausheben:

- Für die Behandlung anwendungsspezifischer Fehler ist eine weitere Komponente mit anwendungsspezifischem Wissen notwendig. Für die Integration dieser Komponente in das ConTract-System ist ein Protokoll erforderlich, durch das sie mit dem ConTract-Manager und evtl. den restlichen Komponenten kommunizieren kann.
- Ein ausfallsicherer Monitor-Prozeß soll die gesamte APRICOTS-Konfiguration auf einem Knoten überwachen. Er muß in der Lage sein, diese Konfiguration nach einem Ausfall zu rekonstruieren.
- Um die zuverlässige ConTract-Verarbeitung auch nach dem Ausfall einer oder mehrerer Komponenten zu garantieren, muß jede dieser Komponenten in der Lage sein, beim Wiederanlauf die Objekte, die sie verwaltet, wieder in einen konsistenten Zustand zu bringen.
- Die Daten (incl. Log-Daten) sollen auf mehreren Knoten redundant gespeichert sein, damit sie auch nach einem permanenten Knotenausfall verfügbar sein können.
- Die automatische Replizierung der Komponenten ist auch ein Mittel zur Realisierung eines robusten Systems.

Abschließend sollte man bemerken, daß die Realisierung eines robusten ConTract-Verarbeitungssystems nur dann möglich ist, wenn die zuverlässige Kommunikation zwischen den beteiligten Prozessen garantiert ist.

Kapitel 6

Unterbrechungsanalyse

Im vorausgegangenen Kapitel haben wir Maßnahmen diskutiert, die man zur Behandlung der im APRICOTS-System häufig anzutreffenden Fehler ergreifen kann. In einigen Fällen, wie z.B dem Ausfall eines Knoten, haben wir für die Fehlerbehandlung gefordert, daß der ConTract-Manager den Zustand eines unterbrochenen ConTracts rekonstruieren kann. Das ist aber nicht ohne weiteres möglich. Weitere Maßnahmen, die erforderlich sind, um den Zustand eines unterbrochenen ConTracts zu rekonstruieren, wollen wir in den nächsten Kapiteln vorstellen. Lösungsansätze dafür können wir aber erst dann entwickeln, wenn wir den internen Verarbeitungszustand eines ConTracts im Fehlerfall näher untersucht und analysiert haben. Diese Analyse werden wir in diesem Kapitel betreiben. Unser Ziel dabei ist, Erkenntnisse zu gewinnen, aus denen man ein Verfahren zum Protokollieren und Wiederherstellen des dynamischen ConTract-Zustands ableiten kann. Für unsere Betrachtung werden nur die Fehler berücksichtigt, die zu einer Unterbrechung der ConTract-Verarbeitung führen, wobei nur der Hauptspeicherinhalt verloren geht. Zur Bezeichnung dieser Fehler werden wir im folgenden auch den Begriff Systemfehler verwenden.

6.1 Unterbrechung der Aktivierung eines ConTracts

Ein Systemfehler kann die Aktivierung eines ConTracts unterbrechen, entwender während sich der ConTract im Zustand INIT oder schon im Zustand RUNNING_FORWARD befindet.

Im ersten Fall trifft die Unterbrechung lediglich den Vorgang der Initialisierung und den Aufbau des Prädikattransitionsnetzes. Dieser Vorgang kann ohne weiteres wiederholt werden, da dazu der Contract-Manager nur Daten benötigt, die schon in den Zwischencode Relationen stabil verwaltet werden. Das bedeutet, daß für die Reaktivierung des ConTracts in diesem Fall keine weitere Vorkehrungen getroffen werden müssen. Lediglich der Versuch, den ConTract zu aktivieren, soll stabil registriert werden.

Der Fall einer Unterbrechung des ConTracts im Zustand RUNNING_FORWARD wird im nächsten Abschnitt ausführlich behandelt.

6.2 Unterbrechungsanalyse im Zustand RUNNING_FORWARD

6.2.1 Unterbrechung einer Step-Ausführung

Durch einen Systemfehler während der Abwicklung eines ConTracts kann die Steps in den folgenden Gruppen einteilen:

- Steps, deren Step-Transaktionen durch eine *get_gtrid*-Nachricht schon installiert wurde, aber noch nicht aktiviert worden sind, d.h der ConTract-Manager hat bis zum Zeitpunkt des Abbruchs noch keinen Step-Computation-Server mit der Ausführung dieser Steps beauftragt.
- Steps, die aktiviert wurden und gerade von einem Step-Computation-Server bearbeitet werden.
- Steps, deren Ausführung beendet ist, aber ihre Step-Transaktion noch keinen der drei möglichen Endzustände "committed", "aborted" oder "prepared" erreicht hat.
- Steps, deren Ausführung beendet ist, ihre Step-Transaktion einen der drei möglichen Endzustände erreicht hat, aber ihre Triggerung bzw. Reaktivierung durch den Systemfehler verhindert wurde.
- Steps, die vollständig bearbeitet wurden, d.h ihre Triggerung bzw. Reaktivierung war zum Zeitpunkt der Unterbrechung schon abgeschlossen.

Weil jeder Step unter dem Schutz einer Transaktion ausgeführt wird, hängt die Behandlung jeder Step-Instanz beim Wiederanlauf vom Zustand der zugehörigen Step-Transaktion ab, die ihrerseits die ACID Eigenschaften erfüllt. Demnach müssen nach einem Systemfehler alle Transaktionen zurückgesetzt werden, die bis zu diesem Zeitpunkt weder den Zustand "committed" noch den Zustand "prepared" erreicht haben. Die Ergebnisse der Transaktionen, die den Zustand "committed" erreicht haben, sollen stabil gespeichert werden, während Transaktionen die sich zu diesem Zeitpunkt schon im Zustand "prepared" befanden, wieder in diesen Zustand überführt werden sollen.

Für die zum Zeitpunkt des Abbruchs in Bearbeitung befindlichen Steps bzw. Step-Instanzen, bedeutet das:

- Die Resultate aktiver Step-Instanzen, deren Step-Transaktion im Zustand "running" ist, müssen beim Restart zurückgesetzt und ihre Step-Transaktion in den Zustand "aborted" überführt werden. Diese Arbeit wird vom Transaktions-Manager und den jeweils an der Transaktion beteiligten Resource-Managers übernommen. Was der ConTract-Manager in diesem Fall tun soll, um die Verarbeitung des ConTracts fortzusetzen, ist, die Reaktivierung dieser Step-Instanzen anzustoßen.
- Die Auswirkungen der Step-Instanzen, deren Step-Transaktion sich schon vor dem Auftreten des Systemfehlers im Zustand "committed" befand, müssen permanent gespeichert werden. Dies unterliegt ebenfalls dem Aufgabenbereich des Transaktions-Manager und der beteiligten Resource-Manager . Der ConTract-Manager muß dafür sorgen, daß die Triggerung dieser Step-Instanzen neu angestoßen bzw. fortgesetzt wird, falls sie durch den Systemfehler verhindert bzw. unterbrochen wurde.

- Die Ergebnisse der Ausführung der Step-Instanzen, deren Step-Transaktion schon vor dem Abbruch den Zustand "prepared" erreicht hat, müssen beibehalten werden, bis die weitere Verarbeitung dieser Step-Instanz entsprechend der Beschreibung im Abschnitt 3.2.2 erfolgen kann.
- Vollständig bearbeitete Step-Instanzen sollen beim Restart (falls noch nicht geschehen) aus den dynamischen ConTract-Daten gelöscht werden.

6.2.2 Unterbrechung der Bearbeitung einer step_finished- Nachricht

Während der Bearbeitung einer *step_finished*-Nachricht wie sie im Abschnitt 3.2.2 beschrieben wurde, kann das Auftreten eines Systemfehlers zu den folgenden Szenarien führen :

- Szenario 1 : Die Berechnung wird nach dem Empfang der *step_finished*-Nachricht und vor dem Empfang der EOT_request_reply Nachricht unterbrochen.
- Szenario 2 : Die Berechnung wird nach dem Empfang der EOT_request_reply Nachricht und während der Triggerung der beendeten Step-Instanz, aber noch vor dem *schedule_stepinst_list()*-Aufruf unterbrochen.
- Szenario 3 : Die Berechnung wird unterbrochen während des *schedule_stepinst_list()* Aufrufes.
- Szenario 4 : Die Berechnung wird nach dem *schedule_stepinst_list()* Aufruf unterbrochen.

Die Szenarien 1 und 2 kann man gemeinsam betrachten, da der ConTract-Manager vom Empfang der *step_finished* Nachricht bis zum Anstoß der Triggerung keine Änderungen an den dynamischen Daten und dem Verarbeitungszustand des ConTracts vornimmt. Solche Änderungen finden nur während der Triggerung der beendeten Step-Instanz statt.

Nach den Szenarien 1 und 2 hat es bis zum Zeitpunkt des Abbruchs folgende Änderungen an den Verarbeitungszustand des ConTracts gegeben :

- Ein Step ist erfolgreich ausgeführt.
- Durch die Triggerung der beendeten Step-Instanz können bis zum Systemabbruch einige dynamische Daten geändert worden sein, wie z.B die Anzahl der empfangenen Tokens einiger Events, die dynamischen Daten eines Steps etc. Wichtig ist aber, daß keine weiteren Steps aktiviert wurden.

Um die Forderung nach einer zuverlässigen Kontrollfluß-Abwicklung zu gewährleisten, müssen, wie schon im vorigen Abschnitt beschrieben wurde, nach dem Neustart des Systems :

- Der Zustand der dynamischen ConTract-Daten, wie z.B Events, Step-Daten etc., rekonstruiert werden.
- Die abgebrochenen Step-Instanzen neu gestartet werden, deren Step-Transaktionen zum Zeitpunkt des Abbruchs sich im Zustand "active",oder "aborted" befanden.

- Die Triggerung der Step-Instanzen angestoßen werden,deren Step-Transaktionen zum Zeitpunkt des Abbruchs sich im Zustand "committed" befanden.

Das bedeutet aber, daß der Ausführungszustand installiert werden muß, in dem sich der ConTract zum Zeitpunkt des Eintreffens der Antwort auf die *EOT_request*-Nachricht befand, da bis zum Systemabsturz keine weitere Steps aktiviert wurden. Dieser Zustand wird von den dynamischen Daten vollständig beschrieben, die bis zum Zeitpunkt des Empfangs der *EOT_request_reply*-Nachricht berechnet wurden.

Die dynamischen Daten, die nach dem Eintreffen der *EOT_request_reply*-Nachricht berechnet wurden, würden dem Verarbeitungszustand beschreiben, der nach dem darauffolgendem *schedule_stepinst_list ()*-Aufruf gelten würde. D.h für das Rekonstruieren des aktuellen Zustands des ConTracts sind diese Daten irrelevant. Sie können erneut berechnet werden, falls der Zustand der dynamischen Daten vor dem Empfang der *EOT_request_reply*-Nachricht vollständig rekonstruiert werden kann und die Triggerung der beendeten Step-Instanz erneut angestoßen wird.

Diese Situation kann man durch das Beispiel der Überweisung verdeutlichen, wenn man annimmt, der ConTract wird während der Triggerung von S2 unterbrochen. Vor der Triggerung war die Anzahl der erhaltenen "Tokens" der Start-Events der Steps S3, S4 und S5 jeweils gleich 0. Nehmen wir weiter an, vor der Unterbrechung erhielten die Start-Ereignisse von S3 und S4 jeweils ein Token. Weil die Steps S3, S4 und S5 noch nicht aktiviert worden sind, kann man beim Wiederanlauf die Anzahl der "Tokens" für die Startereignisse von S3 und S4 wieder auf 0 setzen und die Triggerung von S2 erneut anstoßen.

Im weiteren kann beim Wiederanlauf die Triggerung der ausgeführten Step-Instanzen nur dann fortgesetzt bzw. neu angestoßen werden, wenn der Zustand der zugehörigen Step-Transaktionen bekannt ist. Die Triggerung z.B von S2 kann nur dann wiederholt werden, wenn das "commiten" der zugehörigen Step-Transaktion bekannt ist. Das bedeutet, der Zustand einer Step-Transaktion bzw. der entsprechendn Step-Instanz muß beim Empfang der *EOT_request_reply*-Nachricht protokolliert werden. Andernfalls muß der Zustand aller Step-Transaktionen, die während der gesamten Bearbeitung des ConTracts bis zum diesem Zeitpunkt aktiviert wurden, vom Transaktions-Manager abgefragt werden, vorausgesetzt der Transaktions-Manager vergißt keine Transaktion, nachdem sie in den Zustand "committed" bzw. "aborted" überführt worden ist. Das erfordert aber zusätzlichen Kommunikationsaufwand zwischen ConTract-Manager und Transaktions-Manager.

Aus der obigen Diskussion für die ersten zwei Szenarien kann man folgendes Fazit ziehen:

- Um den Verarbeitungszustand zu rekostruieren, in dem sich der ConTract zum Zeitpunkt der Unterbrechung befand, werden nur die Daten benötigt, die bis zum jüngsten *schedule_stepinst_list()*-Aufruf (einschließlich) – und damit auch bis zum Empfang der nächsten *EOT_request_reply* Nachricht– berechnet wurden.

- Der Zustand aller Step-Transaktionen muß beim Wiederanlauf bekannt sein, um eine abgebrochene Triggerung bzw. Reaktivierung erneut starten zu können.

- Zwischen dem *schedule_stepinst_list*-Aufruf und dem Empfang der Antwort auf eine EOT-Anforderung werden keine relevanten Log-Daten erzeugt.

Nach dem dritten Szenario wurde schon der Ausführungszustand des ConTracts verändert, da weitere Steps aktiviert worden sind. Um in diesem Fall den neuen Ausführungszustand rekonstruieren zu können, müssen dynamische Daten, die diesen Zustand beschreiben, beim Wiederanlauf im Log vollständig vorhanden sein. D.h Informationen über die aktivierten Step-Instanzen müssen aus dem Log entnommen werden können, um ihre Aktivierung erneut anzustoßen. Damit muß auch der Zustand der Events rekonstruierbar sein, deren Feuern zur Ausführung dieser Steps geführt hat.

Wird z.B der ConTract Überweisung unterbrochen nachdem S3, S4 und S5 aktiviert wurden, müssen beim Wiederanlauf die Änderungen, die der ConTract-Manager während der Triggerung von S2 durchgeführt hat (z.B neue Step-Instanzen, Änderung der dynamischen Step-Daten von S3,S4,S5, etc.), bis zum Zeitpunkt des Abbruchs stabil protokolliert sein. Dann braucht man für die Recovery nur die Steps zu reaktivieren.

Um die korrekte Kontrollflußabwicklung während der weiteren ConTract-Verarbeitung zu garantieren, müssen noch die Events protokolliert sein, die während der letzten Triggerung ein Token erhalten aber noch nicht gefeuert haben, sowie die Änderungen an den dynamischen Step-Daten protokolliert werden. Die Protokollierung dieser Daten muß vor dem Ende der Bearbeitung der *step_finished*-Meldung erfolgen. [1]

Die Step-Instanzen, deren Aktivierung durch den Abbruch verhindert wurde, gehören auch zum abgebrochenen Ausführungszustand, da ihre Aktivierung die Folge der Änderungen der dynamischen Daten (Events, dynamische Stepdaten) ist, die zu diesem Zustand geführt haben. Sie müssen beim Restart ebenfalls, wie die abgebrochenen Step-Instanzen, neu aktiviert werden. Damit müssen auch diese Daten vor dem *schedule_stepinst_list*-Aufruf protokolliert werden.

Das Szenario 4 wurde schon durch die Diskussion der Szenarien 1 bis 3 behandelt, da nach der letzten Aktivierung der Step-Instanzen und bis zum Empfang der nächsten *EOT_request_reply*-Nachricht keine weitere Änderung an den dynamischen Daten vorgenommen wird. In diesem Fall müssen genau dieselben Daten protokolliert worden sein wie im Szenario 3.

Dieselben Szenarien können während der Bearbeitung einer *step_finished*-Nachrich abgespielt werden, die dem ConTract-Manager die fehlerhafte Ausführung einer Step-Instanz mitteilt. Da die Bearbeitung in diesem Fall nach demselben Verfahren abläuft, wird auf die weitere Betrachtung dieses Falles verzichtet.

Zusammenfassend kann man aus der Fehlerbetrachtung im Zustand RUNNING_FORWARD folgende Punkte herausheben:

- Im Zustand RUNNING_FORWARD werden Änderungen an den dynamischen ConTract-Daten während der Triggerung bzw. Neuaktivierung einer Step-Instanz vorgenommen.

[1] Im allgemeinen werden am Ende der Bearbeitung einer *step_finished*-Meldung ein oder mehrere Steps aktiviert. In manchen Fällen werden aber während einer Triggerung "Tokens" an Ereignisse gesendet, ohne daß diese Ereignisse feuern. Im Beispiel der Überweisung, wird die Triggerung eines der Steps S3,S4 und S5 lediglich dazu führen, daß die Anzahl der 'Tokens" des Start-Ereignisses von S6 um eins erhöht wird. Der ConTract-Manager wird S6 erst aktivieren, wenn alle drei Steps beendet sind.

- Diese Änderungen an den dynamischen ConTract-Daten können als eine Einheit betrachtet werden, da sie eine gewisse Atomaritätseigenschaft aufweisen. Bei der Recovery müssen entweder alle diese Änderungen im Log vorhanden sein oder keine.
- Die Änderungen an den dynamischen ConTract-Daten, die eine Veränderung des Verarbeitungszustands zufolge haben, müssen vor dem Übergang in den neuen Zustand stabil protokolliert werden.

Aus den obengenannten Punkten kann wie im Abschnitt 7.4.1 beschrieben wird, der Zeitpunkt bestimmt werden, zu dem die Ergebnisse der Triggerung eine ausgeführten Step-Instanz protokolliert werden.

6.3 Unterbrechung der Suspendierung eines ConTracts

Während des Vorgangs der Suspendierung kann ein Systemfehler den ConTract in einen der folgenden Zustände überführen :

- Ein Teil der aktivierten Step-Instanzen ist beendet, die restlichen sind durch den Systemfehler abgebrochen.
- Alle Step-Instanzen sind beendet, d.h der ConTract befindet sich im Zustand SUSPENDED_INTERN und der Vorgang der Externalisierung ist evtl. bereits angestoßen.
- Der ConTract hat den Zustand SUSPENDED_EXTERN erreicht.

Wie schon im Abschnitt 3.2.3 beschrieben wurde, findet während der Suspendierung eines ConTracts weder eine Triggerung noch eine Aktivierung bzw. Reaktivierung von Step-Instanzen statt.

Wird der Vorgang der Suspendierung durch den Systemfehler abgebrochen, während der ConTract sich im Zustand SUSPENDING_SOFT bzw. SUSPENDING_HARD befindet, erreicht der ConTract einen Zustand, in dem einige aktivierte Step-Instanzen schon beendet sind, während die Ausführung der restlichen durch den Systemfehler abgebrochen ist.

Um beim Restart des ConTracts die weitere Abwicklung korrekt durchzuführen, muß die Suspendliste wieder aufgebaut und vervollständigt werden, um den ConTract in den Zustand zu überführen, der von der Suspend-Nachricht gefordert wurde. Zusätzlich soll der Zustand der beendeten Step-Instanzen gerettet werden, um bei der Fortsetzung des ConTracts, die von einer *resume*-Nachricht gefordert wird, die Triggerung bzw. Reaktivierung entsprechend dem Zustand der jeweiligen Step-Instanz anstoßen zu können. Das bedeutet *alle beendeten bzw. aktiven* Step-Instanzen müssen zu jedem Zeitpunkt protokolliert sein.

Damit die von der suspend-Nachricht geforderten Zustandsübergänge beim Restart nachvollzogen werden können, muß beim Restart der Empfang dieser Nachricht bekannt sein.

6.4 Unterbrechung beim Wiederanlauf eines suspendierten ConTracts

Nach einem Systemfehler während des Wiederanlaufs eines suspendierten ConTrats können folgende Situationen auftreten:

- Der Vorgang wird während der Internalisierung unterbrochen (Zustand INTERNALIZING).
- Der ConTract-Manager hat die Internalisierung abgeschlossen aber die Triggerung bzw. die Reaktivierung der Step-Instanzen aus der Suspend-Liste wird unterbrochen.
- Der ConTract wird nach der Aktivierung der Step-Instanzen unterbrochen, die während der Bearbeitung im Zustand RESUMING entstanden sind.

Im ersten Fall führt die Unterbrechung lediglich zum Verlust der aufgebauten Datenstrukturen des ConTracts. Diese lassen sich ohne weiteres wieder aufbauen, wenn man den Vorgang erneut anstößt, da alle dazu benötigten Informationen schon bei der Suspendierung des ConTracts gesichert wurden. Daselbe gilt auch für den zweiten Fall. Hier führt zwar der ConTract-Manager Änderungen an den dynamischen ConTract-Daten durch, aber der neue Ausführungszustand tritt erst nach der Aktivierung der erzeugten Step-Instanzen ein.

Im Abschnitt 3.2.4 ist bemerkt worden, daß sich die Triggerung bzw. Reaktivierung aller Step-Instanzen aus der Suspend-Liste wie die Triggerung einer einzelnen Step-Instanz auswirkt. Weil sich der ConTract-Manager den ConTract schon vor der Aktivierung der auszuführenden Step-Instanzen in den Zustand RUNNING_FORWARD überführt, kann man die Betrachtung des dritten Falles auf die Betrachtung des dritten Szenario aus dem Abschnitt 6.2.2 zurückführen. Auf eine Wiederholung dieser Diskussion wird hier verzichtet.

6.5 Zusammenfassung

Aus der vorausgegangenen Analyse kann man leicht die kritischen Punkte während der ConTract-Abwicklung erkennen, die ein Protokollierungsverfahren für das Einsammeln der Log-Daten berücksichtigen soll. Wir fassen diese Stellen kurz zusammen.

Die Initialisierung der Datenstrukturen des ConTract-Managers nach der Aktivierung eines ConTracts ist ein Vorgang, der idempotent gegen Wiederholungen ist. Wichtig hierbei sind die Ereignisse, die zur ersten Step-Aktivierung führen.

Im Zustand RUNNING_FORWARD ist der Vorgang der Triggerung, der Schritt, der zur Änderung des Ausführungszustands eines ConTracts führt. Die Meldungen über die Zustandsänderung einer Step-Transaktion ist ein weiteres Ereignis, das bei der Protokollierung berücksichtigt werden soll.

Während der Suspendierung werden keine weiteren Steps aktiviert. Die Absicht der Anwendung, den ConTract zu suspendieren, sowie die Zustandsänderungen der bis zu diesem Zeitpunkt aktivierten Steps bzw. ihrer Step-Transaktionen, sind Ereignisse, die sich der ConTract-Manager auf jeden Fall merken muß.

Zwei Schritte führt der ConTract-Manager beim Wiederanlauf eines suspendierten ConTracts durch: das Rekonstruieren des gesicherten Zustands und die Triggerung bzw. Reaktivierung der Step-Instanzen aus der Suspend-Liste.

Kapitel 7

Protokollierung

In diesem Kapitel stellen wir die Verfahren für die Protokollierung des ConTract-Managers vor. Im ersten Abschnitt wird auf die Anforderungen eingegangen, die diese Verfahren erfüllen sollen. Im weiteren werden, nach einem kurzen Überblick über die Mechanismen, die für die transaktionsorientierten Recovery entwickelt worden sind, die obengennanten Verfahren ausführlich beschrieben.

7.1 ConTract-Abwicklung vs. Datenbank-Transaktionen

Im Abschnitt 3.1.3 ist erwähnt worden, daß die ConTract-Verarbeitung sich auf das Konzept der Transaktion stützt. Der ConTract-Manager bildet die auszuführende Steps auf Transaktionen ab, um Steps als atomare Berechnungseinheiten verarbeiten zu lassen. Damit könnte man meinen, es wäre ausreichend, für die Protokollierung und den Wiederanlauf eines ConTract-Managers Verfahren einzusetzten, die für die Recovery in den herkömlichen Datenbanksystemen angewandt werden. Dieser Abschnitt wird zeigen, daß dies nicht zutreffend ist.

In der Terminologie der gegenwärtig existierenden Datenbanksystemen sind Transaktionen atomare Aktionen, die eine Datenbank von einem konsistenden Zustand in einen anderen ebenfalls konsistenten Zustand überführen. Im Vergleich dazu, nützt das ConTract-Modell Transaktionen aus, um in einer verteilten Umgebung Anwendungen zu modellieren, die sich über die Grenzen herkömlicher Datenbanksysteme hinaus erstrecken. In diesem Sinne wird die Transaktion zu einem Konstrukt erweitert, das die Implementierung fehlertoleranten, verteilten Abläufe erleichtert. Dieses Konstrukt bietet im einzelnen Strategien zur Fehlerbehandlung, Synchronisation und Interaktion konkurrender Prozessen, Garantie der Dauerhaftigkeit von Operationen auf abstrakte Objekte, etc.

Im weiteren führen herkömmliche Datenbanksysteme Transaktionen als isollierte Aktionen aus. Zwischen den einzelnen Transaktionen bestehen keine Beziehungen. Das erweiterte Konzept der *geschachtelten* Transaktion erlaubt zwar eine commit-Abhängigkeit zwischen Top-Level und ihre Sub-Transaktionen, aber Top-Level Transaktionen werden als strikt isoliert gegenüber anderen Top-Level Transaktionen betrachtet. Infolge dessen sind trans-

aktionsorientierte Recovery-Verfahren so konzipiert, daß sie lediglich eine Datenbank nach einem Fehler in einen transaktionskonsistenten Zustand überführen.
Demgegenüber erfordert die Konsistenzeigenschaft des ConTract-Modells beim Wiederanlauf nicht nur einen transaktionskonsistenten Zustand zu rekonstruieren sondern den ConTract fortzusetzen und in endlicher Zeit in einem definierten Endzustand terminieren zu lassen. Allein durch die Protokollierung des Zustands und der notwendigen Informationen zum Wiederholen bzw. Zurücksetzen der Transaktionen, wie es für den Wiederanlauf in Datenbanksystemen angewandt wird, ist diese Forderung nicht zu erfüllen. Vielmehr wird benötigt, stabile Erreignisse wie z.B die Aktivierung und Beendigung eines Steps zu realisieren, um in der Lage zu sein, den ConTract-Ablauf nach einer Unterbrechung nachvollziehen zu können.

7.2 Anforderungen

Eins der Hauptziele des ConTract-Modells ist, aktivierte ConTracts immer in einem definierten Zustand zu verwalten und sie in endlicher Zeit korrekt terminieren zu lassen. Die Konsistenzeigenschaft des ConTract-Modells fordert, daß eine ConTract-Verarbeitung nach einem Systemausfall nicht stoppt, sondern fortgesetzt wird, sobald es der Systemzustand erlaubt [26]. Für den ConTract-Manager bedeutet es, daß er in der Lage sein muß, beim Wiederanlauf nach einem Systemausfall, den dynamischen ConTract-Zustand zu rekonstruieren und die Verarbeitung fortzusetzen. Dabei soll der Verlust an geleistete Arbeit minimal gehalten werden. Um diese Forderung zu erfüllen müssen beim Wiederanlauf folgende Voraussetzungen erfüllt sein:

- Alle auf den ausgefallenen Knoten aktive ConTracts müssen bekannt sein.
- Die Daten, die den Verarbeitungszustand eines unterbrochenen ConTracts beschreiben müssen vorhanden und rekonstruierbar sein.

Dazu wird ein Protokoll benötigt, in dem jeder ConTract-Manager den dynamischen Zustand des von ihm bearbeiteten ConTracts auf stabilen Speicher sichert . An das Protokollierungsverfahren, das dabei angewandt werden soll, stellen sich folgende Anforderungen:

- Der Verlust von Log-Daten nach einem Systemfehler muß ausgeschlossen sein.
- Um den Verarbeitungszustand gegen Systemausfälle zu sichern, ist ein Protokollierungsverfahren nach dem Write-Ahead-Log-Prinzip erforderlich.
- Die Zusatzkosten des Protokollierungs- und Wiederanlaufverfahrens sollen nicht zu erheblichen Performanseinbußen des Systems führen.

7.3 Protokollierung bei der Transaktionsorientierten-Recovery

7.3.1 Aktionen während der Transaktions-Recovery

Die Aktionen, die während der Transktions-Recovery durchgeführt werden sollen, müssen das Datenbank-System - und im allgemeinen das Transaktionsverarbeitungsystem - in einem Transaktionskonsistenten Zustand. Ein Transaktionskonsistenter Zustand liegt genau dann vor, wenn er nur die Ergebnisse von erfolgreich beendeter Transaktionen enthält. Um einen solchen Zustand nach einem Fehler zu rekonstruieren, müssen bei der Recovery folgenede Aktionen durchgeführt werden :

- Die Resultate aller Transaktionen, die vor der Unterbrechung schon "committet" haben, müssen nach der Recovery auf den stabilen Speicher vorhanden sein. Wurden diese Ergebnisse vor dem Abbruch nicht stabil gespeichert, müssen sie während der Recovery rekonstruiert werden (REDO).
- Die Transaktionen, die zum Zeitpunkt der Unterbrechung aktiv waren, müssen im Zustand aborted überführt und ihre Ergebnisse zurückgesetzt (UNDO) werden. Zurückgesetzt sollen auch die Resultate der Transaktionen, die vor der Unterbrechung abgebrochen wurden - falls das noch nicht geschehen ist.

7.3.2 Verfahren zur Protokollierung bei der Transaktionsorientierten-Recovery

Mehrere Verfahren sind in den letzte zwei Jahrzenten zur Transaktions-Recovery, die nach verschiedenen Kriterien in mehreren Kategorien aufgeteilt werden können[?]. Eine grobe Unterteilung kann diese Verfahren in zwei Kategorien anordnen: **Carefull Replacement** und **Write-Ahead-Logging (WAL)**.

Das in vielen Implementierungen eingesetztes Carefull Replacement Verfahren vermeidet die Änderungen "in place" von Objekten (z.B Seiten), die während einer Transaktion geändert werden. Für die Speicherung solcher Objekte wird ein neuer Speicherbereich ("Schatten"-Kopie) belegt, der sie aufnimmt. Erst wenn die Transaktion erfolgreich beendet wird, wird der alte Speicherbereich durch den neuen ersetzt. Falls die ändernde Transaktion zurückgesetzt werden soll, wird die "Schatten"-Kopie für das Rekonstruieren des alten Zustands der geänderten Objekten verwendet.
Dieses Verfahren wurde oft durch den zusätzlichen Aufwand kritisiert, der während des normalen Transaktionsabwicklung benötigt wird - Ersetzung beim "commit". Kritik wird ebenfalls auf die geringen Parallelität ausgeübt, die diese Verfahren wegen des Sperrens von Objekten erlaubt, sowie der geringen Unterstützung für die Protokolle , die für die Commit Abwicklung verteilter Transaktionen benötigt werden.

Diese Nachteile lassen sich durch Write Ahead Protokollierungsverfahren vermeiden. Hierbei werden Informationen, die bei der Recovery benötigt werden, ins Log geschrieben, bevor die geänderte Objekte auf dem stabilen Speicher mit ihren neuen Werten überschrieben werden.

In den folgenden Abschnitten werden Alternativen zur Implementierung WAL-Verfahren kurz skizziert.

7.3.3 Logbasierte Transaktions-Recovery

Log-Daten sind redundante Informationen, die zum Zwecke der Recovery nach einem System- oder Gerätefehler verwendet werden.

Ein log-basiertes System für Transaktions-Recovery besteht aus einer Hierarchie folgender Speichertypen:

- Flüchtiger Speicher als Puffer für die Zwischenspeicherung von Log-Sätzen.
- Nichtflüchtiger Speicher i.a Plattenspeicher, der für die Recovery nach einem Systemfehler verwendet wird (auch temporärer Log genannt).
- Stabiler Speicher, der durch die redundante Speicherung von Log-Informationen auf mehreren nichtflüchtigen Speichermodule mit unabhängigen Fehlerhäufigkeiten implementiert wird. Damit sind die Log-Daten auch nach einem Gerätefehler verfügbar (es sei denn, es sind Fehler aufgetreten, deren Behandlung bei Systementwurf nicht berücksichtigt wurde).

Verfahren zur logbasierte Transaktions-Recovery kann man durch eine Vielzahl von Kriterien charakterisieren, wie die Art der Log-Einträge, die Seitenersetzungsstrategien, die Art der Seitenersetzung etc.

Value vs. Transition-Logging

Protokolliert man den Zustand der von der Transaktionen berührten Objekte, spricht man von *value*-Logging. Werden für Protokollierung nur die Operationen auf die Objekte und die Argumente dieser Operationen verwendet, nennt man das Verfahren *transition*-Logging.

Beim *value*-Logging enthällt jeder Log-Satz den alten (before-Image) und den neuen (after-Image) Zustand eines Objektes. Die Information über den alten Zustand eines geänderten Objektes kann bei der Recovery zum Zurücksetzen (UNDO-) der Ergebnisse einer abgebrochenen Transaktion, während der neue Zustand zum rekonstruieren der Resultaten abgeschlossener Transaktionen (REDO-) verwendet werden.

Während des REDO-Teils der Recovery wird beim transition-Logging die protokollierete Operation mit ihrer Argumente erneut angewandt. Zum Zurücksetzen der Resultate einer Operation, die während einer nicht erfolgreich abgeschlossenen Transaktion aktiviert wurde, wird eine Operation angewand, die die Effekte der ursprünglichen exakt kompensiert.

Value-Logging hat den Vorteil, daß die Länge der protokollierten Sätze sehr viel kleiner ist als beim transition-Logging. Ihr Einsatz aber in der Praxis wird vor allem durch folgende Nachteile erschwert:

- Für das Zurücksetzen der Ergebnisse nicht erfolgreich beendetet Transaktionen, muß nicht die ändernde Operation protokolliert werden, sondern eine, die die Effekte der ursprünglichen kompensiert. Solche Operationen zu definieren ist aber besonders schwierig, wenn mehrere Objekte von einer Änderungsoperation betroffen werden, wie es z.B bei Netzwerkorientierten Datenbanksysteme häufig der Fall ist.

- Ein Systemfehler kann eine Datenbank in einem speicherinkosistenden Zustand hinterlassen, besonders dann, wenn mehrere Seiten zusammen geändert werden. In einer solchen Datenbank können aber keine Operationen ausgeführt werden und damit kein UNDO- bzw. REDO-Recovery durchgeführt werden, wenn ein logisches Protokollierungsverfahren angewendet wird.

Verwaltung des Log-Puffers

Ein weiteres Kriterium, das ein Protokollierungsverfahren charakterisiert, ist die Verwaltung des verwendeten flüchtigen Speichers. Der Log-Puffer dient zur Minimierung der Kosten der Protokollierung. Die gewählte Ersetzungsstrategie der Objekte (i.a Seiten im Log-Puffer) beinflußt entscheidend das Vorgehen bei der Recovery und die Leistungsfähigkeit des Systems während des normalen Betriebs.

Einbringstrategien, die eine Menge von Seiten im Log-Puffers durch Operationen ausschreiben und deren Effekt es ist, daß entweder alle Seiten oder keine eingebracht werden, bezeichnet man als *atomaren* Einbringstrategien (ATOMIC) [13]. Kann eine solche Operation durch einen Fehler abgebrochen werden, so daß ein Teil der Seiten schon eingebracht worden ist der Rest aber nicht, spricht man von einer *nicht atomaren* Einbringstrategie (NO-ATOMIC).

Im Falle einer nicht atomaren Einbringstrategie, kann ein Systemfehler die Datenbank in einem undefinierten Zustand hinterlaßen, während im Falle einer atomaren Einbringstrategie, die Datenbank sich nach einem solchen Fehler in demselben (definierten) Zustand befindet, der nach dem letzten erfolgreichen Ausschreiben eingetreten ist.

Ist es während der Protokollierung jederzeit erlaubt, geänderte Seiten aus dem Puffer im stabilen Speicher zu überschreiben, spricht man von einem STEAL-Algorithmus. Im Gegensatz dazu, bezeichnet man als NO-STEAL, die Algorithmen, die fordern, die von einer Transaktion geänderten Seiten im Puffer mindestens solange zu behalten, bis die Transaktion erfolgreich abgeschlossen ist.

NO-STEAL Algorithmen vereinfachen das Recovery-Verfahren enorm, da ein Zurücksetzen nicht erfolgreich beendeter Transaktionen nicht notwendig wird – ist die Transaktion abgebrochen worden, dann sind mit Sicherheit ihre Änderungen nicht sichtbar–. Zu ihrer Realisierung benötigt man aber sehr lange Puffer, besonder wenn sie zur Implementierung von Anwendungen mit hohen Transaktionsraten verwendet werden sollen.

Das Rücksetzen der Ergebnissen von erfolgreich beendeten Transaktionen kann während der Recovery vermieden werden, wenn bei der EOT-Behandlung alle von der beendeten Transaktion geänderte Seiten vom Puffer auf den stabilen Speicher übertragen werden (FORCE-Strategie). Der Vorteil dieser Strategie gegenüber einer NO-FORCE-Strategie liegt in der kleiner Menge von Informationen, die protokolliert werden müssen -(da keine REDO-Informationen benötigt werden) und in der geringere Komplexität des Recovery-

Algorithmus (kein REDO-Vorgang benötigt). Diese Strategie weist aber einen großen Nachteil auf, nämlich, die hohe Kosten, die die EOT-Behandlung während des normalen Betriebs verursacht.

Checkpoints

Die Datenmenge, die bei der Recovery analysiert werden muß, kann durch das Generieren periodischer Checkpoints während der normalen Transaktionsverarbeitung reduziert werden.

Bei der Generierung eines Checkpoints werden Informationen über den Zustand des Puffers gesammelt und stabil gespeichert. Im einfachsten Fall werden alle Seiten im Puffer ausgeschrieben, wobei sichergestellt wird, daß alle Änderungen vor diesem Punkt stabil gespeichert sind. Damit braucht man bei der Recovery kein REDO von Transaktionen durchzuführen, die vor dem jüngsten Checkpoint ihr "commit" erreicht haben.

Eine Beschreibung der verschiedenen Verfahren zur Generierung von Checkpoints findet sich in [13].

Kosten bei der Transaktion-Recovery

Die Verschiedene Protokollierungsverfahren für transaktionsorientierte Recovery verursachen verschiedene Kosten während des normalen Betriebs sowie während der Recovery. Eine Aussage darüber zu machen, welches Recovery Verfahren besser ist , ist schwierig, da es keine allgemeingültige Kriterien zu Bewertung von Algorithmen gibt.

Um den Aufwand eines Recovery Verfahren abzuschätzen, braucht man Kriterien, die eine quantitative Bewertung erlauben und in ihrer Gesamtheit ein Maß für die Gesamtkosten dieses Verfahren ergeben. Da eine Aufstellung solcher Bewertungskriterien nicht Teil dieser Arbeit ist, wird nur auf zwei Aspekte zu Beurteilung der Leistungsfähigkeit von Recovery-Verfahren kurz eingegangen.

Der erste Aspekt betrifft die Kosten während der normalen Transaktionsverarbeitung. Diese werden bestimmt, durch die Häufigkeit der Schreiboperationen auf den nichtflüchtigen Speichers sowie die Menge der Daten, die bei diesen Operationen übertragen werden. Die Strategie zur Pufferverwaltungs ist dabei ein wichtiger Faktor. Recovery Verfahren, die das Ausschreiben des Puffers während der EOT-Behandlung fordern (FORCE), weisen vergleichsweise geringere Performance im normalen Betrieb auf, als Recovery-Verfahren die einen NO-FORCE Algorithmus verwenden. Auch die Wahl des Checkpoint-Verfahrens beinflußt entscheindend die Leistungsfähigkeit eines Datenbanksystems. Je öfters z.B Checkpoints erzeugt werden, um so geringer wird die Transaktionsverarbeitungsrate.

Der zweite Aspekt zur Aufwandsabschätzung eines Recovery-Verfahren, ist das Leistungsverhalten des Verfahrens während eines Restarts. Diese Kosten werden vor allem durch folgende Faktoren beeinflußt:

- Die Anzahl der Sätze, die während der Recovery analysiert werden müssen.

- Die Anzahl der Sätze, die aus dem Log in den Puffer geladen, dort geändert und wieder ausgeschrieben werden sollen.

Die Seitenersetzungsstrategie trägt auch hier entscheidend zur Minimierung bzw. Erhöhung der Kosten bei. Wird eine NO-FORCE Strategie zur Protokollierung verwendet, sind bei der Recovery Daten, sowohl für UNDO nicht beendeter, als auch für REDO beendeter Transaktionen erforderlich, während durch den Einsatz eines FORCE-Algorithmus nur UNDO-Informationen benötigt werden. NO-STEAL-Algorithmen führen wiederum zur Minimierung der Kosten bei der Recovery, da dabei der UNDO-Vorgang entfällt, und damit weniger Sätze betrachtet werden müssen.

7.4 Protokollierungsverfahren für den ConTract-Manager

In den nachfolgenden Abschnitten wird das Verfahren vorgestellt, das der ConTract-Manager zur Sicherung des Verarbeitungszustands eines ConTracts anwendet. Dabei soll der Versuch unternommen werden, das Verfahren unabhängig von der gewählten Implementierung zu beschreiben. Es läßt sich aber nicht vermeiden, bei der Beschreibung eines Protokollierungsverfahrens bestimmte Ausdrücke wie z.B "zwischenspeichern" oder "stabil protokollieren" zu verwenden. Aus diesem Grund erwähnen wir an dieser Stelle die Existenz eines Log-Puffers zur Zwischenspeicherung sowie die Existenz nichtflüchtiger Speichermodule zur Realisierung des stabilen Speichers. Dem ConTract-Manager wird eine Schnittstelle bereitgestellt, über die er auf das Log zugreifen kann. Die weiteren Realisierungsdetails werden im Kapitel 9 ausführlich beschrieben.

7.4.1 Protokollierung im Zustand RUNNING_FORWARD

Vorüberlegungen

Die Anhaltspunkte für die Protokollierung während der Bearbeitung einer *step_finished*-Nachricht sind im Abschnitt 6.2.2 skiziert worden. Die erste Erkenntnis, die man aus der Diskussion im obengenannten Abschnitt gewonnen hat, betrifft die für die Protokollierung relevanten Daten. Es wurde beschrieben, daß man beim Restart unbedingt über alle Informationen über die aktivierten Step-Instanzen, den Zustand der zugehörigen Step-Transaktion sowie darüber, ob die Step-Instanz vollständig bearbeitet wurde oder nicht, verfügen muß, um den Ausführungszustand des ConTracts zum Zeitpunkt des Abbruchs rekonstruieren zu können. Daraus folgt, daß man für jede aktivierte Step-Instanz folgende Ereignisse protokolieren muß:

- Ihre Aktivierung
- Jede Änderung an dem Zustand der zugehörigen Step-Transaktion
- Ob die Triggerung dieser Step-Instanz abgeschlossen ist oder nicht.

- Da bei der Aktivierung einer Step-Instanz einige Daten verändert werden, die den dynamischen Zustand des entsprechenden Steps beschreiben, müssen diese Veränderungen auch protokolliert werden.

Um im weiteren Verlauf der ConTract-Verarbeitung die Forderung nach einer zuverlässigen Kontrollflußabwicklung zu erfüllen, muß der Zustand aller Events, deren Zustand bis zum Zeitpunkt des Abbruchs verändert wurde, bekannt sein. Damit wird garantiert, daß jeder Event in der weiteren Contract-Abwicklung genau zu dem Zeitpunkt feuert, der in der Spezifikation des Contract-Skriptes festgelegt wurde. Ist die Anzahl der erhaltenen Tokens eines Events vor dem Crash nicht dem ConTract-Ablauf entsprechend protokolliert worden, kann es vorkommen, daß dieses Event frühzeitig oder überhaupt nicht feuert. Dies kann zur einer fehlerhaften Kontrollflußabwicklung oder zu einem "Deadlock" führen. Da während der Bearbeitung einer *step_finished*-Nachricht (genauer während der Triggerung einer beendeten Step-Instanz) die Anzahl der Tokens einiger Events erhöht bzw. nach dem Feuern erniedrigt wird, muß an dieser Stelle, der Zustand (die received-Tokens) dieser Events protokolliert werden.

Eine zweite Erkenntnis über die Protokollierungsstrategie, die aus dem Abschnitt 6.2.2 gewonnen werden kann, ist der Zeitpunkt der Protokollierung bzw. der Speicherung der erzeugten Protokolldaten.

Es wurde im obengenannten Abschnitt herausgehoben:

- Die Änderungen an den dynamischen Daten müssen *vor* dem Übergang in den neuen Zustand protokolliert werden. Da dieser Übergang durch die Aktivierung der Step-Instanzen erfolgt, müssen die Daten vor dem *schedule_stepinst_list* -Aufruf protokolliert werden. Das entspricht der WAL-Protokollierungstrategie.

- Alle in diesem Bearbeitungsschritt vorgenommenen Änderungen können den neuen Zustand vollständig beschreiben, wenn sie in ihrer Gesamtheit im Log vorliegen. Das bedeutet, daß diese Logsätze nur dann für die Recovery verwendet werden können, wenn auch der letzte Log-Satz in diesem Arbeitsschritt erzeugt wurde. Der wiederum ist mit Sicherheit spätestens beim *schedule_stepinst_list*-Aufruf erzeugt worden.

Damit ist es naheliegend die Logsätze im Hauptspeicher bzw. Puffer bis zur Aktivierung der neuen Step-Instanzen zwischenzuspeichern. Unmittelbar vor dem *schedule_stepinst_list*-Aufruf sollen sie auf den stabilen Speicher übertragen werden. Dabei soll ein Log-Satz am Ende des Puffers angehängt werden, der das Ende der Triggerung für die beendete Step-Instanz anzeigt. Dieser Eintrag entfällt im Falle einer fehlerhaften Step-Ausführung.

Es bleibt noch zu klären, wann die Antwort auf die EOT-Anforderung für die Transaktion der behandelten Step-Instanz im nichtflüchtigen Teil des Logs geschrieben wird. Dazu gibt es zwei Möglichkeiten

- Unmittelbar nach Empfang der Antwort auf diese Anforderung

- Zusammen mit den übrigen Log-Daten die während der nachfolgender Bearbeitung bis zur *schedule_stepinst_list*-Aufruf erzeugt wurden.

Im ersten Fall werden für die stabile Speicherung der Log-Daten, die in diesem Arbeitsschritt anfallen, zwei Schreiboperationen auf das externe Speichermedium erforderlich. Das bedeutet aber zusätzliche Kosten während des normalen Betriebs gegenüber der zweiten Möglichkeit, die nur eine Schreiboperation für die Speicherung derselben Datenmenge erfordert. Der Nachteil im zweiten Fall liegt darin, daß, falls ein Systemfehler auftritt bevor die Daten stabil protokolliert wurden, muß der ConTract-Manager bei der Recovery den Status dieser Step-Transaktion durch eine Nachricht an den Transaktions-Manager abfrage, was zusätliche Kommunikationskosten zur Folge hat. Weil aber Systemfehler nicht die Regel, sondern die Ausnahme bilden, wird die zweite Möglichkeit vorgezogen.

Verfahren

Das aus der Diskussion in diesem Abschnitt abgeleitete Protokollierungsverfahren wird in der Abbildung 7.1 wiedergegeben. Nach den Empfang der Antwort auf die EOT-Anforderung wird der Zustand der beendeten Step-Transaktion in den Log-Puffer geschrieben. Bei der folgenden Triggerung bzw. Neuaktivierung werden die Änderungen an den dynamischen ConTract-Daten, sowie die auszuführende Step-Instanzen im Log-Puffer vermerkt. Ist die Triggerung beendet, werden alle im Log-Puffer vermerkte Änderungen durch eine *flushlog*-Operation auf das externe Speichermedium übertragen.

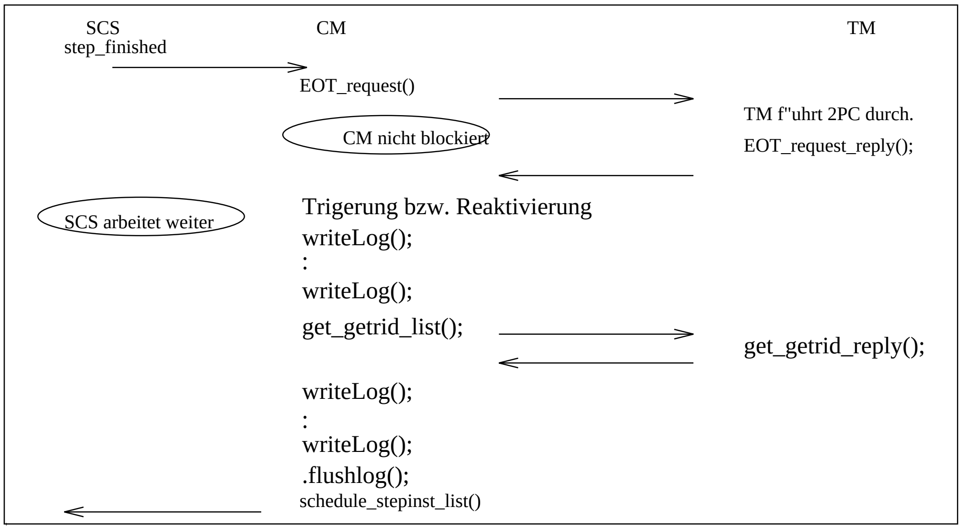

Abbildung 7.1: Protokollierung während der Bearbeitung einer step_finished-Nachricht im Zustand RUNNING_FORWARD

7.4.2 Protokollierung während der Suspendierung

Um den Vorgang der Suspendierung nach einer Unterbrechung fortzusetzen, werden folgende Informationen benötigt:

- Das vom ConTract-Manager empfangene Suspend-Komando und die übergebenen Parameter (duration, modus).
- Die aktive Step-Instanzen.
- Die Step-Instanzen, die der ConTract-Manager schon vor dem Abbruch an die Suspend-Liste angehängt hat.

Um diese Daten zu sichern, geht der Contract-Manager wie folgt vor:

Bevor er die Suspendierung einleitet, protokolliert er die Suspend-Nachricht und ihre Parameter durch einen Satz auf dem stabilen Log.
Im weiteren Ablauf während der Suspendierung werden die Endzustände der Steptransaktionen, sowie die Zustandsübergänge des ConTracts protokolliert. Im Falle einer längerdauernden Suspendierung wird, bevor der ConTract auf den Zustand SUSPENDED_EXTERN gesetzt wird, ein Protokollsatz geschrieben, der auf die beim letzten Checkpoint gesicherten Daten des ConTracts verweist. Damit wird garantiert, daß der ConTract den Zustand SUSPENDED_EXTERN erreicht hat, bevor alle seine dynamische Daten gesichert wurden.

7.4.3 Protokollierung im Zustand RESUMING

Damit der Contract-Manager den Wiederanlauf eines suspendierten ConTracts erneut anstoßen kann, falls dieser Vorgang durch einen Fehler unterbrochen worden ist, protokolliert er die Suspend-Nachricht unmittelbar, nachdem er sie empfangen hat. Im weiteren Ablauf protokolliert der ConTract-Manager die Änderungen an den ConTract-Daten, die er während der Triggerung bzw. der Vorbereitung zur Reaktivierung der Step-Instanzen aus der Suspend-Liste durchführt. Diese Änderungen vermerkt er, wie auch während der Triggerung im Zustand RUNNING_FORWARD, zunächst in den Log-Puffer und überträgt sie erst unmittelbar vor der Aktivierung der auszuführenden Step-Instanzen auf den nichtflüchtigen Speicher.

Es muß bemerkt werden, daß es nicht notwendig ist, die Zwischenzustände zu protokollieren, die der ConTract während dieses Vorgangs annimmt. Diese sind idempontent gegen Unterbrechungen, da sie jederzeit aus den während der Suspendierung protokollierten Daten rekonstruiert werden können. Lediglich der letzte Übergang in den Zustand RUNNING_FORWARD muß im Log vermerkt werden.

7.4.4 Checkpoints

Um den Zustand der dynamischen ConTract-Daten zu rekonstruieren, kann es unter Umständen notwendig sein, daß der ConTract-Manager Log-Sätze liest, die er schon in den ersten Schritten der Kontollflußabwicklung protokolliert hat. Bei langlebigen Aktivitäten kann es dazu führen, daß der ConTract-Manager Log-sätze lesen muß, die sehr weit hinten im Log liegen. Dies kann aber sehr hohe Kosten beim Wiederanlauf verursachen.

Um dies zu vermeiden, werden periodische Checkpoints erzeugt, wobei alle Daten protokolliert werden, die den ConTract-Zustand beschreiben. Damit braucht man beim Wiederanlauf, den Log nur ab dem letzten Sicherungspunkt zu betrachten. Als Kriterium für die Erzeugung eines Sicherungspunktes kann der Ablauf eines bestimmten Ausführungsintervalls, die Aktivierung einer bestimmten Anzahl von Steps, etc. dienen. Dabei kan der ConTract-Manager einen Checkpoint erzeugen, indem er eine Nachricht generiert, die er an sich selbst sendet. Diese Nachricht soll eine relativ hohe Priorität haben. Während eines Checkpoints kann der Contract-Manager keine weitere Nachrichten bearbeiten. Die übrigen Komponenten können aber wohl ihre Arbeit fortsetzen. Das bedeutet, die Ausführung von Steps muß nicht unterbrochen werden, um einen Checkpoint zu generieren. Es ist aber selbstverständlich, daß durch ein Checkpoint die Aktivierung weiterer Steps verzögert wird. Ein Checkpoint kann auch auf Anforderung des ConTract-Clients generiert werden.

Das im nächsten Kapitel beschriebene Wiederanlaufverfahren berücksichtigt solche Sicherungspunkte.

7.5 Bewertung

In diesem Abschnitt wollen wir eine Abschätzung der zusätzlichen Kosten angeben, die die Protokollierung während des normalen Betriebs verursacht. Dabei wird nicht beabsichtigt, ein absolutes Kostenmaß zu geben, wie z.B "das Verfahren verlängert die Bearbeitung eines ConTracts um x Sekunden". Eine solche Aussage wäre nicht sinvoll, einerseits, weil sie sich nur auf eine bestimmte Umgebung beziehen kann, andererseits, weil das ConTract-System gegenwärtig nicht soweit ausgereift ist, um solche Leistungsmeßungen durchführen zu können. Hier soll lediglich eine Abschätzung angegeben werden, die relative Faktoren wie z.B Anzahl der Blocktransfers, enthält.

Im weiteren betrachten wir nur die Vorwärtsabwicklung (im Zustand RUNNING_FORWARD) eines ConTracts, da dabei am häufigsten protokolliert wird. Der Aufwand der Protokollierung während der Bearbeitung in den restlichen Zuständen ist gering.

Im Abschnitt 7.3.3 ist erwähnt worden, daß die Anzahl der Ein-/Ausgabe Operationen der entscheidenter Faktor für die Abschätzung der Kosten ist, die ein Protokollierungs bzw. Recovery-Verfahren während des normalen Betriebs bzw. des Wiederanlaufs verursacht. Während des normalen Betriebs schreibt der ConTract-Manager Sätze auf den stabilen Log vor der Aktivierung von Steps oder während eines Sicherungspunktes. Wir betrachten die beide Fälle getrennt.

Wir gehen bei der Abwicklung des ConTracts davon aus, daß jede Step-Instanz erfolgreich ausgeführt wird. Im Falle einer sequentiellen Ausführung, triggert jede Step-Instanz ein einziges Ereignis, das seinerseits die Aktiviereung einer neuen Step-Instanz anstößt. dabei werden die dynamische Daten des zugehörigen Steps verändert. Der ConTract-Manager protokolliert hierbei vier Sätze, jeweils einen für die obengenannten Objekten. Wenn wir eine Länge von b Bytes für jeden Log-Satz und eine Blocklänge von g Bytes annehmen, werden am Ende einer Triggerung

```
4b/g (1)
```

Blocks auf den stabilen Speicher übertragen. In der weiteren Beschreibung verwenden wir die Abkürzung (auch als Einheit) [BLTR] für einen Blocktransfer.

Für den Fall, daß k Steps parallel aktiviert werden, errechnet man

```
(3k+1)b/g [BLTR]   (2)
```

Das bedeutet, die Kosten der Protokollierung steigen proportional zum Parallelitätsgrad eines ConTracts.

Wenden wir uns den Kosten zu, die während eines Checkpoints anfallen. Hier werden alle dynamische ConTract-daten gesichert. Falls der ConTract s Steps enthält, dann sind unter der Voraussetzung, daß jeder Step nur einen einzigen Start-Event[1] besitzt, 2s Sätze für die Events und Steps zu protokollieren. Wenn man weiter annimmt, daß zum Zeitpunkt des Checkpoints m Step-Instanzen aktiv sind, dann müssen insgesamt

```
2s+m+1  (3)
```

Sätze – ein Satz wird zur Protokollierung des Contact-Zustands benötigt– auf den stabilen Log übertragen werden bzw.

```
(2s+m+1)b/g [BLTR].  (4)
```

Aus den Formeln (2) und (4) erkennt man, daß diese Kosten des Protokollierungsverfahrens mit steigender Step-Anzahl proportional steigen, was auf hohe Kosten schließen läßt. Der Grund dafür liegt darin, daß jede Step-Aktivierung mit den dazu verbundenen Erreignissen stabil protokolliert wird (vgl. FORCE). Dies ist aber nicht zu vermeiden, da man sonst die Forderung nach zuverlässigen und robusten Kontrollflußabwicklung nicht mehr erfüllen könnte.

7.6 Verbesserungsvorschläge

Die im vorigen Abschnitt berechnete Kosten für die Protokollierung lassen sich in gewisser Weise eingrenzen.

Eine Möglichkeit, hier Verbesserungen zu erzielen, ergibt sich aus der folgenden Überlegung: da bei den oben angegebenen Abschätzungen die Anzahl der Steps als fest betrachtet werden muß, könnte man einen besseres Verhalten erreichen, indem wir versuchen den Faktor b/g soweit wie möglich zu minimieren. Es könnte z.B dadurch erreicht werden, daß man die Länge des Log-Satzes auf ein Minimum einschränkt, während man eine größere Blocklänge wählt.

Eine weitere Möglichkeit die Kosten der Protokollierung zu reduzieren ,wäre, beim Checkpoint nicht alle Contract-Daten zu protokollieren, sondern nur diejenigen, deren Wert ungleich ihr Initialisierungswert ist. Man könnte z.B nur die Events zu protokollieren, bei denen die Anzahl der empfangenen Tokens größer als 0 ist, da man die restliche Events beim Wiederanlauf aus dem Zwischencode rekonstruieren kann. Genauso könnte man nur die Steps protokollieren, die schon aktiviert wurden.

[1] Das Terminierungs-Ereignis eines Steps ist gleichzeitig das Stert-Ereignis eines anderen.

7.7 Zusammenfassung

In diesem Kapitel ist ein Verfahren beschrieben worden, das man als den ersten Schritt ansehen kann, die Forderungen zu erfüllen, die die Konsistenzeigenschaft des ConTract-Modells an die Kontrollflußabwicklung stellt. Dieses Verfahren bietet dem ConTract-Manager einen Mechanismus an, um den Zustand eines ConTracts sichern zu können. Damit ist er in der Lage, nach einer durch einen Fehler herbeigeführten Unterbrechung den ConTract fortzusetzen und in einem definierten Zustand zu beenden. Dies wird vor allem durch das WAL-Prinzip garantiert, auf das sich der Protokoolierungsalgorithmus stützt. Die WAL-Eigenschaft stellt sicher, daß während der ConTract-Bearbeitung kein Zustandsübergang erfolgt, ohne daß vorher die den neuen Zustand beschreibenden Informationen protokolliert sind. Im einzelnen bedeutet dies:

- Der Contract-Manager veranlaßt vor dem Übergang in einen neuen internen Zustand [2] , daß der neue Zustand stabil protokolliert wird.
- Auszuführende Step-Instanzen werden schon vor ihre Aktivierung ins Log vermerkt.
- Erreignisse werden festgehalten, bevor sie die Änderungen des Verabeitungszustands bewirken. Das Feuern eines Events z.B wird protokolliert, bevor die von ihm getriggerten Step-Instanzen aktiviert worden sind.

Durch die Pufferung bis zum *schedule_stepinst_list*-Aufruf der Änderungen, die der ConTract-Manager während der Triggerung einer erfolgreich beendeten Step-Instanz durchführt, wird vermieden, daß man beim Wiederanlauf einen Zustand rekonstruiert, der vor dem Abbruch überhaupt nicht eingetreten war.

Im weiteren wird durch die Protokollierung der Zustandsübergängen einer Step-Transaktion vermieden, daß beim Wiederanlauf:

- eine Step-Instanz mehrfach aktiviert wird.
- eine abgebrochene Triggerung einer Step-Instanz nicht fortgesetzt wird.[3]

Durch die Protokollierung der dynamischen Step-Daten wird sichergestellt, daß die Versionierung der aktivierten Step-Instanz nach dem Wiederanlauf weiterhin eindeutig ist. Das ist ein grundlegendes Korrektheitskriterium vor allem für die Kontextverwaltung und die Kompensation.

Abschließend ist noch zu erwähnen, daß durch die Protokollierung bestimmter Nachrichten der ConTract-Manager in der Lage ist, die ihm übermittelten Komandos (z.B suspend, resume) auch nach einem Systemausfall auszuführen.

---

[2] Siehe Abbildung 3.3

[3] Nur dann wird die Triggerung einer Step-Instanz beim Wiederanlauf nicht fortgesetzt bzw. wiederholt, wenn der Zustand dieser Transaktion im Log mit "committet" vermerkt ist.

Kapitel 8

Wiederanlauf

8.1 Einige Vorbemerkungen

Ziel des Wiederanlaufverfahrens ist, einen unterbrochenen ConTract nach einer Unterbrechung fortzusetzen und zwar genau in den Zustand, in dem sich der ConTract zum Zeitpunkt des Abbruchs befand. Damit aber die weitere ConTract-Abwicklung auch korrekte Ergebnisse liefert, müssen auch die Objekte, auf denen die Steps operieren, sich in demselben Zustand wie zum Zeitpunkt des Abbruchs befinden. Das bedeutet, daß vor dem Wiederanlauf des ConTracts, müssen die Ergebnisse aller erfolgreich ausgeführten Steps bzw. der Step-Transaktionen rekonstruiert werden. Entsprechend müssen die Ergebnisse aller abgebrochenen Steps bzw. Step-Transaktionen zurückgesetzt werden. In anderen Worten: die Objekte, auf die die Steps zugreifen, müssen in einen "transaktionskonsistenten" Zustand überführt werden. Weil der Transktionsmanager der Koordinator aller im ConTract-System laufenden Transaktionen ist, muß er die Recovery aller, bis zum Abbruch abgewickelte Transaktionen und damit auch der beteiligten Ressource-Manager durchführen, bevor der ConTract-Manager die ConTract-Verarbeitung fortsetzt.
Ein konsistenter Zustand der übrigen Komponeneten des APRICOT-Systems wird bei der Vorstellung des in den folgemden Abschnitten beschriebenen Verfahrens vorausgesetzt.

Weil der Contract-Manager während des Wiederanlaufs Step-Instanzen aktiviert bzw. reaktiviert, ist es erforderlich, die damit verbundene Änderungen an des dynamischen ConTract-Zustand zu protokollieren. Andernfalls ist er nicht in der Lage, diese Änderungen im Falle einer erneuten Unterbrechung nachvollziehen zu können. In der nachfolgender Beschreibung sind die Aktionen zur Protokollierung während des Wiederanlaufs miteinbezogen.

8.1.1 Wiederanlauf im Zustand RUNNING_FORWARD

Die ersten Schritte, die der ConTract-Manager ausführt, nachdem er die Nachricht RESTART_CM empfangen hat, sind dieselben, die er nach dem Eintreffen einer ACTIVATE Nachricht ausführt. Nämlich:

- Initialisieren seiner internen Datenstrukturen

- Laden der Statischen Daten aus der Zwischencode Datenbank.

Im nächsten Schritt rekonstruiert der ConTract-Manager den Zustand der dynamischen Daten, die er beim jungsten Checkpoint -falls vorhanden- gesichert hat.
Im weiteren Ablauf werden die Log-Sätze betrachtet, die ab dem letzten Checkpoint bis zum Abbruch des ConTracts geschrieben wurden und daraus der zuletzt gültige Zustand wie folgt rekonstruiert:

- Für alle Events und Steps, deren Zustand nach dem letzten Checkpoint verändert wurde, wird aus dem Log jeweils die letzte Änderung entnommen und entsprechend aktualisiert.

- Die Step-Instanzen, die zum Zeitpunkt des Checkpoints aktiv waren aber inzwischen vollständig abgearbeitet sind, werden aus der Liste der aktiven Step-Instanzen des ConTracts gelöscht.

- Die Liste der aktiven Step-Instanzen wird mit den Step-Instanzen ergänzt, die nach dem letzen Checkpoint aktiviert wurden und bis zum Abbruch noch nicht vollständig abgearbeitet sind.

Nach diesem Schritt ist der Zustand der dynamischen ConTract-Daten vor dem Abbruch vollständig rekonstruiert. An dieser Stelle wird ein Checkpoint erzeugt und der aktuelle Zustand der dynamischer ConTract-Daten gesichert. Damit wird es vermieden, die bis zu dieser Stelle geleistete Arbeit zu wiederholen, falls ein Systemfehler den Restart unterbricht.

Um die Verarbeitung fortzusetzen, muß noch die unterbrochene Ausführung bzw. Triggerung der Step-Instanzen wiederaufgenommen werden, deren Steptransaktion sich zum Zeitpunkt des Abbruchs schon im Zustand "active" (oder "aborted") bzw. "comitted" befanden.

Dazu bearbeitet der ConTract-Manager die Liste, die alle zum Zeitpunkt des Abbruchs aktive Step-Instanzen enthält. Für jede dieser Step-Instanzen prüft er den aktuellen Zustand ihrer Steptransaktion. Es liest zunächst den Zustand, der in der dynamischen Daten-Struktur dieser Step-Instanz abgespeichert ist. Der gelesene Wert entspricht dem Zustand der Steptransaktion beim Abbruch. Ist noch der Zustand "active" oder "prepared" eingetragen, wird durch die *get_TA_state*-Nachricht der neue Zustand der Steptransaktion vom TM abgefragt - andernfalls entspricht der eingetragene Zustand dem aktuellen Zustand der Step-Transaktion. Der durch die Antwort des TM mitgeteilte Zustand der Step-Transaktion wird protokolliert.

Ist die Steptransaktion im Zustand "aborted", wird sie in der Liste der Step-Instanzen angehängt, die aktiviert werden sollen. Ist die Steptransaktion im Zustand "committed", wird die Step-Instanz in die Liste der Step-Instanzen angehängt, deren Triggerung angestoßen werden muß.

Nachdem die gesamte Liste der aktiven Step-Instanzen bearbeitet worden ist, fordert der ConTract-Manager für jede Step-Instanz, die aktiviert werden soll, eine Transaktions_Id, geht in den Zustand RUNNING_FORWARD und veranlaßt der ConTract-Manager das Ausschreiben des Log-Puffers. Anschließend beauftragt er den CPM mit dem Scheduling der auszuführenden Step-Instanzen.

Im letzten Schritt der Restart-Prozedur wird die Triggerung der Step-Instanzen angestossen, deren Step-Transaktionen " committet" haben. Für jede dieser Step-Instanzen wird der Vorgang der Triggerung (incl. Protokollierung), der im Abschnitt 4.1.1 beschrieben wurde, wiederholt.

Der Vorgang der Reaktivierung der abgebrochenen Steps entspricht der Aktivierung bzw. Reaktivierung (incl. Protokollierung) von Steps im Zustand RUNNING_FORWARD. Damit kann das obengenannte Wiederanlaufverfahren auch dann angewendet werden, wenn ein Systemfehler während der Restart-Prozedur auftritt.

Wird die Restart-Prozedur abgebrochen bevor der Checkpoint installiert und der aktuelle Zustand der dynamischen Daten gesichert wurde, dann tritt dieselbe Situation wie vor dem Begin des Wiederanlaufverfahren ein, da es keine (protokollierte) Änderungen der dynamischen Daten gegeben hat.

Tritt der Systemfehler auf, nachdem der Checkpoint vollendet worden ist, wird der neue Checkpoint für den neuen Wiederanlauf verwendet. Damit geht die Arbeit, die beim ersten Wiederanlaufversuch geleistet wurde, nicht verloren.

Da während der Restart-Prozedur vor jeder Änderung des Ausführungszustands des ConTracts dieser gesichert wird - Auschreiben des Log-Puffers vor jedem *schedule_stepinst_list*-Aufruf -, ist auch der neu eingetretene Verarbeitunszustand nach einer erneute Unterbrechung rekonstruierbar.

Das beschriebene Verfahren wird in der Abbildung 8.1 in der Form eines Pseudokode-Notation angegeben.

8.1.2 Wiederanlauf vom Zustand SUSPENDING

Beim Wiederanlauf eines ConTracts, der während einer laufenden Suspendierung durch einen Systemfehler unterbrochen wurde, werden zunächst dieselben Schritte vorgenommen, wie beim Wiederanlauf aus dem Zustand RUNNING_FORWARD. D.h die Initialisierung und das Überführen der dynamischen Daten auf dem Zustand, in dem sie sich zum Zeitpunkt des Abbruchs befanden, erfolgt nach dem Verfahren aus dem Abschnitt 8.1.1. Der Unterschied zu dem angegebenen Verfahren ist, daß in diesem Fall nicht die Liste mit den Step-Instanzen, die getriggert werden sollen, sondern die Suspendliste rekonstruiert wird und zwar in demselben Zustand wie zum Zeitpunkt des Abbruchs.

Falls sich der ConTract zum Zeitpunkt des Abbruchs im Zustand SUSPENDING_SOFT bzw. SUSPENDING_HARD befand, wird die Suspendliste vervollständigt. Dazu wird der Status jeder zum Zeitpunkt des Abbruchs aktiven Steptransaktion vom TM abgefragt und anschließend ein Eintrag mit der entsprechenden Step-Instanz in die Suspendliste eingefügt. Sobald die Suspendliste vollständig ist, werden dieselben Aktionen wie auch im normalen Fall durchgeführt, um den Vorgang der Suspendierung abzuschließen. D.h der Contract wird in den Zustand SUSPENDED_INTERN überführt und externalisiert, falls von der suspend-Nachricht gefordert. Die ConTract-Abwicklung wird im Zustand SUSPENDED_EXTERN angehalten.

Hatte der ConTract schon vor dem Abbruch den Zustand SUSPENDED_INTERN erreicht,

```
AS_ERROR_CODE CM_restart( IN  ADT_gid   ConTract_id,
                          OUT int      *ConTract_state)
  { Initialisiere die ConTract-Daten;
    Restauriere Zustand aus dem jungsten Checkpoint;
    Lese Log vom letzten Checkpoint bis zum Ende;
    Restauriere den Zustand der dynamischen ConTract-Daten zum Zeitpunkt des Abbruchs;

    ConTract_state := RUNNING_FORWARD;
    Erzeuge neuen Checkpoint ;
    for (jede Step-Instanz aus der Liste der aktiven Step-Instanzen)
      { state = stepinst.state;
        if ((state == active) or (state == prepared)) {
           get_TA_State( ta_state, stepinst.gtrid);
           Protokolliere den aktuellen Zustand der Step-Transaktion;
        }
        if (state == aborted) {
            fuege die Step-Instanz in der Liste der auszuf"uhrenden Step-Instanzen;
        }
        else if (state == comitted) {
          Fuege Step-Instanz in der Liste der ''commiteten'' Step-Instanzen;
        }
      }
  Fordere neue Transaktions-Id's fuer alle Steps, die abgebrochen wurden;
  Protokolliere neue Aenderungen;
  Aktiviere alle abgebrochene Steps;
  for (jede nicht bearbeitete commitete Transaktion)
    { Triggere neue Steps; }
}
```

Abbildung 8.1: Wiederanlauf vom Zustand RUNNING_FORWARD

ist die aus den ersten Schritten rekonstruierte Suspendliste schon vollständig und die Liste der aktiven Step-Instanzen leer. In diesem Fall wird nur der Übergang zum Zustand SUSPENDED_EXTERN, falls gefordert, vollzogen.

Falls der ConTract schon vor dem Abbruch den Zustand *SUSPENDED_EXTERN* erreicht hat, wird der Wiederanlauf überhaupt nicht angestoßen. Die weitere Abwicklung des ConTracts kann durch die Bearbeitung der resume-Nachricht ohne weiteres fortgesetzt werden.

Das in diesem Abschnitt beschriebene Wiederanlaufverfahren in der 8.2 in der Form eines Pseudokodes angegeben.

8.2 Wiederanlauf nach Unterbrechung im Zustand RESUMING

Im Abschnit 3.2.4 ist beschrieben worden, daß bei der Reaktivierung eines suspendierten ConTracts der ConTract-Manager zunächst den Zustand der internen Datenstrukturen re-

```
AS_ERROR_CODE CM_restart_from_suspend(
                          IN  ADT_gid   ConTract_id,
                          OUT int      *ConTract_state)
  {
     readLog_ConTract_state(ConTract_state)
    if (ConTract_state == SUSPENDED_EXTERN)
      return();
    Initialisiere die ConTract_Daten;
    Restauriere Zustand aus dem jungsten Checkpoint;
    Lese Log vom letzten Checkpoint bis zum Ende;
    Restauriere den Zustand der dynamischen Daten zum Zeitpunkt des Abbruchs;
    Erzeuge neuen Checkpoint ;
    if (ConTract_state == SUSPENDING_SOFT OR
        ConTract_state == SUSPENDING_HARD){
        for (jede stepinst aus der Liste der aktiven Step-Instanzen){
           state = stepinst.state;
           if ((state == active) or (state == prepared)) {
             get_TA_State( ta_state, stepinst.gtrid);
             Vermerke Step-Instanz in die logliste;
             fuege Step-Instanz in die Suspendliste;
            }
        }
        Vermerke (ConTract_state = SUSPENDED_INTERN) in die logliste;
        ConTract_state = SUSPENDED_INTERN;
    }

    if (ConTract_state == SUSPENDED_INTERN)
      if (suspendmodus == permanent){
          externalisiere ConTract-Daten;
          Vermerke (ConTract_state = SUSPENDED_EXTERN) in die logliste;
          ConTract_state = SUSPENDED_EXTERN;
      }
}
```

Abbildung 8.2: Wiederanlauf vom Zustand SUSPENDING bzw. SUSPENDED

konstruiert - falls der ConTract permanent suspendiert war - und erst dann die Triggerung bzw. Reaktivierung der Step-Instanzen aus der Suspend-Liste anstößt. Unmittelbar bevor der ConTract-Manager die auszuführenden Step-Instanzen aktiviert, setzt er den ConTract in den Zustand RUNNING_FORWARD. Weil aber bis zu diesem Zustandübergang keine dynamischen Daten protokolliert werden, beschreiben die protokollierten Daten, den Zustand, in dem sich der ConTract vor dem Empfang des *resume*-Komandos befand. Dieser Zustand kann entweder SUSPENDED_EXTERN oder SUSPENDED_INTERN sein.

Dies nutzen wir hier aus, um den Wiederanlauf wie den Vorgang im normalen Betrieb zu gestalten. Falls sich der ConTract vor seiner Reaktivierung im Zustand SUSPENDED_EXTERN befand, brauchen wir nur den Vorgang wie im normalen Betrieb anzustoßen (siehe Abschnit 3.2.4. Falls der ConTract nur temporär suspendiert worden ist, führen wir den Wiederanlauf, wie im vorigen Abschnit für den Zustand SUSPENDED_INTERN beschrieben wurde. Sobald dieser Zustand rekonstruiert wird, stoßen wir die Reaktivierung des ConTracts wie sie im Abschnit 3.2.4 für diesen Zustand beschrieben wurde. Die Abbildung 8.3 verdeutlicht diesen Vorgang.

```
AS_ERROR_CODE CM_restart_from_resuming(
                        IN  ADT_gid   ConTract_id,
                        OUT int      *ConTract_state)
{
  readLog_ConTract_state(ConTract_state)
  if (ConTract_state == SUSPENDED_INTERN){

     Initialisiere die ConTract_Daten;
     Restauriere Zustand aus dem jungsten Checkpoint;
     Lese Log vom letzten Checkpoint bis zum Ende;
     Restauriere den Zustand der dynamischen Daten zum Zeitpunkt des Abbruchs;
     Erzeuge neuen Checkpoint ;
     for (jede stepinst aus der Liste der aktiven Step-Instanzen){
        state = stepinst.state;
        if ((state == active) or (state == prepared)) {
           get_TA_State( ta_state, stepinst.gtrid);
           Vermerke Step-Instanz in die logliste;
           fuege Step-Instanz in die Suspendliste;
        }
     }
   }
 CM_resume();
}
```

Abbildung 8.3: Wiederanlauf vom Zustand RESUMING

8.3 Korrektheits-Überlegungen für das Wiederanlaufverfahren

In diesem Abschnitt wollen wir erläutern, daß der beschriebene Wiederanlauf die Forderung nach zuverlässiger Kontrollflußabwicklung erfüllt. Dabei wird hier auf einen aufwendigen, formalen Beweis verzichtet. Dem Leser soll die Korrektheit dieses Verfahrens durch eine einfache Fallunterscheidung plausibel gemacht werden.

Wir führen die Diskussion nur für den Fall, daß der ConTract in den Zustand RUNNING_FORWARD unterbrochen wurde. Für die restlichen Fälle beschränken wir uns auf folgende Bemerkung:
Beim Wiederanlauf eines während der Suspendierung bzw. der Reaktivierung nach einer Suspendierung unterbrochenen ConTracts rekonstruiert der ConTract-Manager zunächst den Zustand der dynamischen ConTract-Daten und beendet den Vorgang wie auch im normalen Fall. Weil die Korrektheit der Abläufe im normalen Betrieb gegeben ist, genügt

für die Korrektheit der Vorgänge beim Wiederanlauf nur die Garantie, daß alle Daten, die der ConTract-Manager zum Aufbau seiner internen Datenstrukturen benötigt, im Log vollständig enthalten sind. Dies garantiert das im vorigen Kapitel beschriebene Protokollierungsverfahren.

Für die Korrektheit des Wiederanlaufverfahrens für den Fall, daß einer ConTract während der Vorwärts Abwicklung unterbrochen worden ist, genügt es folgendes zu zeigen:

- Zusicherung 1: Alle abgebrochene Step-Instanzen werden wiederholt.
- Zusicherung 2: Eine Triggerung oder Reaktivierung, die unterbrochen bzw. verhindert wurde, wird vom ConTract-Manager erneut bzw. vollständig durchgeführt.
- Zusicherung 3: Keine erfolgreich beendete Step-Instanz wird mehrfach aktiviert.

Betrachten wir irgendeine Step-Transaktion. Diese kann beim Wiederanlauf im Zustand "commited", "aborted" [1] sein. Den Zustand "prepared" betrachten wir hier nicht, da der ConTract-Manager solche Step-Transaktionen nicht weiter behandelt. Diesen Zustand kann der ConTract-Manager schon protokolliert oder aber auch nicht protokolliert haben. Damit ergeben sich die vier Fälle, die in der Tabelle aus der Abbildung 8.4 registriert sind.

Fall	Step-Transaktion	Log	Aktion beim Wiederanlauf
1	commited	nicht vorhanden	Triggerung
2		vorhanden	Keine weitere Maßnahmen erforderlich
3	aborted	nicht vorhanden	Reaktivierung
4		vorhanden	Keine weitere Maßnahmen erforderlich

Abbildung 8.4: Zusammenfassung der Recovery-Aktionen des Contract-Managers

Ist eine Step-Transaktion im Zustand "commited" nicht im Log vermerkt, bedeutet das, die Triggerung der zugehörinen Step-Instanz ist nicht abgeschlossen bzw. verhindert ist, da der ConTract-Manager diesen Zustand nur dann protokolliert, wenn auch die Triggerung vollständig durchgeführt wurde. Der Contract-Manager findet diese Step-Transaktion in seinem Log im Zustand "running", da er jede Step-Transaktion bzw. Step-Instanz protokolliert, bevor er sie ausführen läßt. Anhand dieses Protokolleintrags fragt der ConTract-Manager den Transaktion-Manager nach dem neuen Zustand dieser Step-Transaktion. Sobald er die Antwort erhält, stoßt er die Triggerung der zugehörigen Step-Instanz an. Damit erfüllt er die zweite Zusicherung.

Findet der ConTract-Manager die Step-Transaktion im Zustand "commited" in seinem Log (Fall 2), heißt es, er hat schon die Triggerung der zugehörigen Step-Instanz vollständig durchgeführt und die dadurch entstandenen neuen Step-Instanzen aktiviert. Daher behandelt er sie nicht mehr weiter. Damit wird die wiederholte Aktivierung derselben Step-Instanzen verhindert (Zusicherung 3).

Die Behandlung einer Step-Transaktion, die sich im Zustand "aborted" befindet, erfolgt ähnlich. Falls sie im Log nicht vermerkt ist (Fall3), bedeutet das, daß ihre Reaktivie-

[1] An dieser Stelle wird vorausgesetzt, daß der Transaktions-Manager die Recovery aller (Step-)Transaktionen durchgeführt hat, bevor der ConTract-Manager mit seinem Wiederanlauf beginnt.

rung noch nicht stattgefunden hat bzw. abgebrochen wurde. Daher läßt sie der ConTract-Manager erneut ausführen, sobald er ihren neuen Zustand vom Transaktions-Manager erfahren hat (Zusicherung 1, 2).
Ist sie im Log vermerkt (Fall 4), dann heißt es, der ConTract-Manager hat sie schon reaktiviert. Er behandelt sie nicht mehr weiter (Zusicherung 3).

Im weiteren kann man aus der Erfüllung der Zusicherungen 1 und 3 folgern, daß jede Step-Instanz genau einmal ausgeführt wird.

8.4 Bewertung

In diesem Abschnit werden die Kosten geschätzt, die das Wiederanlaufverfahren verursacht. Auch hier wird nur der Fall betrachtet, daß der Wiederanlauf für einen ConTract angestoßen wird, der während der Vorwärts Abwicklung unterbrochen wurde.

Wenn wir die in den vorausgegangenen Abschnitten beschriebenen Verfahren betrachten, dann stellen wir fest, daß folgende Ein/Ausgabeoperationen erfolgen:

- Lesen der Daten, die während des letzten Checkpoints gesichert wurden.
- Lesen der nach dem letzten Checkpoint protokollierten Sätze.
- Erzeugen eines Checkpoints
- Protokollierung aller veränderten ConTract-Daten vor der Aktivierung der neu erzeugten Step-Instanzen.

Die Kosten für die Initializierung der ConTract-Datenstrukturen, sowie die Kosten für den Aufbau des Prädikat-Transitionsnetzes betrachten wir hier nicht, da sie konstant für jede Aktivierung sind.

Für die weitere Betrachtung setzen wir voraus, daß die Kosten für eine Ausgabe und eine Eingabe gleich sind. Damit können wir für das Lesen der Daten, die im letzten Checkpoint gesichert wurden, die Formel (4) aus dem Abschnit 7.5 heranziehen. Dieselbe Formel können wir auch für die Abschätzung der Kosten bei der Erzeugung des neuen Checkpoints verwenden. Damit ergeben sich für diese zwei Fälle zusammen

```
2(2s+m+1)b/g [BLTR] (5)
```

wobei die Zahl der aktiven Step-Instanzen in beiden Fällen gleich groß angenommen wird (m).

Falls der ConTract-Manager nach dem letzten Checkpoint weitere l Steps aktiviert hat, dann muß der ConTract-Manager weitere 4l Sätze aus dem Log lesen. Als letzte kommen die Kosten, durch die Protokollierung vor der Aktivierung der neuen Step-Instanzen, verursacht. Bei p auszuführende Step-Instanzen sind es weitere

```
4pb/g [BLTR]
```

Damit ergeben sich beim Wiederanlauf insgesammt

```
[2(2s+m+1) + 4l + 4p] b/g  [BLTR]   (6)
```

Wie man aus der Formel erkennen kann, sind die Kosten beim Wiederanlauf umso höher, je größer die Anzahl der ausgeführten Steps ist.

Die Verringerung der Kosten kann dadurch erreicht werden, daß die Checkpoints in kürzeren Abständen erzeugt werden. Dann wäre die Anzahl der Sätze, die der ConTract-Manager nach dem letzten Checkpoint lesen muß geringer. Damit würden aber die Kosten während der Protokollierung ansteigen. Diese Entscheidung wäre nicht sinvoll, weil man davon ausgehen sollte, daß der Wiederanlauf ein seltenes Ereignis ist.

Eine weitere Möglichkeit, die Kosten beim Wiederanlauf zu senken, ergibt sich aus der Tatsache, daß der ConTract-Manager nur die jeweils letzte Änderung seiner Objekte benötigt, um dessen Zustand zu rekonstruieren. Würde man den ConTract-Manager einen mengenorientierten Zugriff auf das Log erlauben, könnte er nur die Sätze auswählen, die jeweils die letzte Änderung eines Objektes beschreiben. Die Implementierung dieses Ansatzes beschreiben wir im nächsten Kapitel.

8.5 Abschließende Bemerkung

Für das in diesem Kapitel vorgestellte Wiederanlaufverfahren wurde angenommen, daß die benötigten Log-Daten beim Restart vollständig im Log vorliegen. Das kann aber bei realen Anwendungen nicht immer garantiert werden, da Systemfehler während der Übertragung von Log-Daten auf den nichtflüchtichen Speicher auftreten können.

Wie schon beschrieben wurde, werden die während der Abwicklung eines ConTracts erzeugte Log-Informationen zunächst auf einen Puffer zwischengespeichert und unmittelbar vor dem Übergang zum neuen Ausführungszustand stabil gespeichert. In anderen Worten heißt es, daß falls die Daten auf dem stabilen Speicher vorhanden, dann hat der Zustandsübrgang stattgefunden andernfalls nicht.

Probleme tretten nun auf, wenn ein Systemfehler die Übertragung der Protokolldaten vom Puffer auf dem nichtflüchtigen Speicher unterbricht. In diesem Fall ist ein Teil der Log-Sätze, die im stabilen Speicher vorhanden sind, für den Wiederanlauf nicht relevant, da der von diesen Daten beschriebene Zustand zum Zeitpunkt des Abbruchs noch nicht eingetreten war.

Solche Situationen lassen sich durch die Realisierung atomarer (ATOMIC) flushlog-Operationen vermeiden. Falls die gewählte Implementierung keine atomare Übertragung von Log-Daten auf das externe Speichermedium vorsieht, muß der ConTract-Manager beim Wiederanlauf nur den Teil des Logs betrachten, der die relevanten Log-Daten enthält.

Kapitel 9

Implementierung eines Protokolls für den ConTract-Manager

9.1 Protokolldaten des ConTract-Managers

Es ist zweckmäßig an dieser Stelle eine kurze Zusammenfaßung der Daten vorzunehmen, die während der ConTract-Abwicklung protokolliert werden müssen. Aus der Beschreibung des Protokollierungs- und Wiederanlaufverfahrens in den Kapiteln 7 und 8 kann man folgende Daten erkennen, die der ConTract-Manager zum Rekonstruieren seines Zustands beim Wiederanlauf benötigt:

- Der **ConTract-Zustand**.
- **Events**: Jede Zustandsänderung eines Events soll im Log enthalten sein. Dieser Zustand wird durch die Anzahl der empfangenen "Tokens" bestimmt.
- **Stepinstanzen**: Für jede Stepinstanz müssen folgende Informationen protokolliert werden:
  - Die Step-Id des zugehörigen Steps.
  - Die Zustandsänderung: Dabei sind folgende Werte für den Zustand einer Step-Instanz relevant: "running", committed", "aborted", "prepared" und "deleted".
  - Die Versionsnummer.
  - Der Parallelitätsindex.
  - Die Transaktions-Id, unter deren Schutz die Stepinstanz ausgeführt wird.
- **Dynamische Step-Daten** : Informationen über die *"history"* eines Steps. Dazu gehören die höchste Version, die Anzahl der aktivierten und die Anzahl der beendeten Step-Instanzen dieses Steps.
- **Nachrichten** : Nachrichten, die zu Zustandsübergängen des ConTracts führen, müssen protokolliert werden, um beim Restart den Vorgang nachvollziehen zu können (siehe Abschnitt 3.2.3).

- **Checkpoints** : Die Nummer des letzten Checkpoints muß protokolliert werden, um beim Restart auf die Daten zugreifen zu können, die während bzw. nach diesem Checkpoint gespeichert wurden.

9.2 Struktur des Logs

9.2.1 Die Struktur eines Log-Satzes

Es wurde eine allgemeine Struktur für den Log-Satz gewählt, die die im Inhalt und Struktur verschiedene Protokolldaten aufnehmen kann. Die gewählte Struktur soll auch mengenorienterten Zugriff auf das Log unterstützen. Sie besteht aus zwei Teilen. Im sogennanten "Kopf" werden Informationen zur Identifizierung des Records, über seine Herkunft (d.h von welchem ConTract-Manager er erzeugt wurde), über seine Länge etc. aufgenommen. Im zweiten Teil, dem "Rumpf", werden die eigentlichen Daten in Form eines Strings gespeichert. Die Strukur eines Log-Satzes des ConTract-Manager wird in der Abbildung 9.1 dargestellt.

Struktur eines Log-Satzes		
Attribut	Typ	Beschreibung
lsn	long	Logsequenznummer.
client	char(60)	ConTract-Manager, der den Log-Satz erzeugt hat
key	CHAR(3)	Kennzeichnet die Art des Satzes.
itemid	char(60)	Identifikation des beschriebenen Elementes innerhalb des Contracts. Z.B Event_id, Step_id, Stepinst_id etc.
length	INTEGER	Gibt die Länge des Strings an, der die eigentliche Informationen enthält
rec	CHAR(200)	Die protokollierten Informationen

Abbildung 9.1: Die Struktur eines Log-Satzes des ConTract-Managers

Die Logsequenznummer dient der eindeutigen Kennung des Log-Satzes. Sie wird von Log-Manager beim Schreiben des Satzes im Log vergeben. In der prototypischen Realisierung ist die lsn ein Zähler, der die Log-Sätze des Contracts aufsteigend nummeriert.
Die ConTract_id wird für die Speicherung in die Log-Relation in einem ASCII-String formatiert und in die Variable *cid* abgelegt. Damit kann man die Log-Sätze, die zu einem bestimmten ConTract-Ablauf gehören selektieren. Durch die ConTract-Id und die Log-Sequenznummer ist jeder Log-Satz im Apricot-System eindeutig identifizierbar.
Die Variable *key* gibt den Typ der im Log-Satz gespeicherten Information an. Diese Variable kann folgende Werte annehmen:

Wert	Bedeutung
CTS	Satz beschreibt den ConTract-Zustand.
MSG	Satz beschreibt eine Nachricht.
CKP	Satz enthält Angaben über einen Checkpoint.
EVT	Satz enthält Informationen über ein Event.
STD	Satz enthält die dynamischen Daten eines Steps.
SIS	Satz beschreibt eine aktive Stepinstanz.
SIA	Satz beschreibt eine Stepinstanz im Zustand aborted.
SIC	Satz beschreibt eine Stepinstanz im Zustand commited.
SID	Satz beschreibt eine Stepinstanz die vollständig bearbeitet worden ist (aus dem ADT_Script gelöscht)

Itemid ist die Identifikation des im Log-Satz gespeicherten Objekts. Dabei wird dieselbe Kennung (Id) verwendet, über die diese Objekte auch vom ConTract-Manager identifiziert werden. Da für diese Id's verschiedene Datentypen verwendet werden, z.B long für Step-Id bzw. Event-Id, ADT_gid für Step-Instanzen etc., werden sie für das Eintragen in die Relation in einen String umgewandelt. Dieser String kann für bestimmte Log-Einträge leer sein z.B MSG, CTS.

Der String *logrecord* enthält entsprechend dem Typ des protokollierten Objektes folgende Struktur:

Typ	Stringstruktur
CTS	"state: %d, log_mgr_id: %s"
MSG	"message: %s , parameter: %s"
CKP	"checkpointnr: %d db_name: %s"
EVT	"rec_Tokens: %d"
STD	"version_cnt: %d, nof_done_inst: %d, nof_run_inst: %d"
SIS	"step_id: %d, gtrid: %s, version: %d, par_index: %d"
SIA	"step_id: %d, gtrid: %s, version: %d, par_index: %d"
SIC	"step_id: %d, gtrid: %s, version: %d, par_index: %d"
SID	"step_id: %d, gtrid: %s, version: %d, par_index: %d"

9.2.2 Struktur des Logs

Das Log, das der ConTract-Manager zur Speicherung seiner Zustandsinformationen verwendet, besteht aus einer Hierarchie zweier Speichertypen.

Der **Log-Puffer** wird im Hauptspeicher realisiert. Er dient zur Minimierung der Kosten, die die Protokollierung verursacht. In der prototypischen Implementierung des ConTract-Systems wird der Log-Puffer als eine verkettete Liste von Log-Sätzen realisiert. Die Verwaltung dieser Liste erfolgt nach dem FIFO-Prinzip. Wegen der Kürze der Log-sätze können wir einen Puffer mit unendlichen Länge annehmen.

Zur stabilen Speicherung der Log-Daten, die während der ConTract-Abwicklung anfallen, werden SQL-Relationen verwendet. Folgende Gründe haben dazu geführt, daß die SQL-Relationen den Vorzug gegenüber Files erhielten:

- Die Kürze und Struktur der Log-Sätze bieten sich für eine einfache und effiziente Verwaltung durch das Datenbaksystem an.

- SQL-Relationen bieten die Möglichkeit einer mengenorientierten Schnittstelle. Statt nur einzelne können mehrere Sätze durch eine Operation geschrieben bzw. gelesen werden. In SQL-Anfragen kann man Auswahlkriterien zum mengenorientierten Zugriff einfach formulieren.

- Die Entwicklung und Implementierung von Verfahren, zur Speicherplatzbelegung und -verwaltung entfallen dadurch. Diese Aufgaben werden vom Datenbanksystem übernommen.

- Die atomaren *flushlog()*-Operationen kann man mit Hilfe von Datenbaktransaktionen einfach implementieren.

Um die Kosten für die Protokollierung und den Wiederanlauf weiter zu minimieren, haben wir für die verwendeten Log-Relationen eine weitere Unterteilung unternommen.

In der *config_table* speichert der ConTract-Manager bei der Aktivierung bzw. Terminierung eines ConTracts einen Satz, der Informationen über diesen ConTract enthält, wie z.B seinen Namen, die ConTract_Id, den Rechnerknoten, auf dem der ConTract bearbeitet wird, Startzeit etc. Damit kann der ConTract-Prozessing-Monitor beim Wiederanlauf die ConTracts bestimmen, die durch einen Fehler unterbrochen worden sind, um sie zu reaktivieren. Die Struktur dieser Tabelle ist in der Abbildung 9.2 dargestellt.

Struktur der Konfigurations-Relation		
Attribut	Typ	Beschreibung
ConTract_Id	CHAR(60)	Id des ConTracts in Form eines Strings
ConTract_Name	CHAR(60)	Bezeichnung des ConTracts
node	CHAR(60)	Rechnerknoten, der den ConTract verarbeitet.
ConTract-Managerid	CHAR(60)	Identifikation des ConTract-Managers, der den ConTract abwickelt.
client	CHAR(60)	Identifikation des ConTract-Clients, der den ConTract gestartet hat.
start_time	TIME	Startzeit des Anstosses der Abwicklung
termination_time	TIME	Zeitpunkt der Terminierung des ConTracts
state	long	Zustand des ConTracts.
log_data	CHAR(60)	Ein Verweis auf die Log-Daten

Abbildung 9.2: Struktur der Kofigurationsrelation

Die Bedeutung der einzelnen Attribute geht aus der Tabellenbeschreibung deutlich hervor, daher wird auf weitere Erklärungen verzichtet. Es wird lediglich bemerkt, daß im Attribut *state* nicht jeder einzelne interener Zustandsübergang eingetragen werden muß. Hierfür sind die Zustände relevant, die auf den Eintrit eines besonderen Ereignisses, wie z.B permanente Suspendierung, Migration, Kompensation, Wiederanlauf,etc hinweisen. Der Hinweis auf die Log-Daten könnte sich auch in gewisse Fällen (wie z.B Migration, etc) als besonders hilfreich erweisen.

In einer weiteren Relation, die *log_table*, speichert der ConTract-Manager seine Log-Daten während der ConTract-Abwicklung. Die Abbildung 9.3 zeigt die Struktur dieser Relation.

Struktur der Log-Relation		
Attribut	Typ	Beschreibung
lsn	long	Log-Sequenznummer. Sie dient der eindeutigen Kennung des Log-Satzes. Wird von LM beim Schreiben des Satzes im Log aufsteigend vergeben.
ConTract_Id	CHAR(60)	Id des ConTracts, während dessen Abwicklung der Log-Satz erzeugt wurde.
key	CHAR(3)	Kennzeichnet die Art des Satzes.
itemid	char(60)	Identifikation des beschriebenen Elementes innerhalb des Contracts. Z.B Event_id, Step_id, Stepinst_id etc.
length	INTEGER	Gibt die Länge des Strings an, der die eigentliche Informationen enthält.
rec	CHAR(200)	Die protokollierten Informationen.

Abbildung 9.3: Struktur der log_table

Der Primärschlüssel dieser Relation setzt sich aus den Attributen *lsn* und *ConTract_Id* zusammen.

Der dritte Teil der Log-Relationen enthält nur die Daten, die der ConTract-Manager während eines Sicherungspunktes (checkpoint) protokolliert. Dieser Teil des Logs besteht aus mehreren Relationen, jeweils eine für jedes Objekt, das zu den dynamischen ConTract-Daten gehört (z.B Events, Steps, etc). Diese Relationen werden im Anhang A ausführlich beschrieben.

Die hier beschriebene Unterteilung des Logs bietet folgende Vorteile:

- Beim Wiederanlauf braucht der ConTract-Prozessing-Monitor nur in der (globalen) Relation *config_table* zu suchen, um die abgebrochenen ConTracts zu bestimmen. Damit ist die Menge der Log-Sätze, die er zu betrachten braucht, viel geringer, als wenn er eine größere Relation mit allen Protokolldaten durchsuchen würde.

- Die *config_table* kann man entsprechend erweitern, um die gesamte Konfiguration des ConTract-Systems aufzunehmen.

- Während eines Checkpoints kann der Contract-Manager die Daten, die er sichern muß, direkt in die entsprechende Relationen schreiben, ohne sie in das allgemeine Log-Satzformat umwandeln zu müssen. Entsprechend kann der ConTract-Manager diese Daten während des Vorgangs der Internalisierung aus den Log-Relationen direkt in seine Datenstrukturen kopieren. Weil während dieser Vorgänge eine relativ große Datenmenge bearbeitet wird, kann man auf diese Weise enorme Leistungseinbußen verhindern.

- Durch die Unterteilung in mehrere Relationen kann man verhindern, daß das Log schlagartig wächst.

An dieser Stelle muß bemerkt werden, daß die Relationen, die der ConTract-Manager zur

Sicherung des ConTract-Zustands während eines Checkpoints verwendet, bei einer späteren Implementierungeines globalen Log im APRICOTS-System eines globalen Logs entfallen können.

9.3 Schnittstelle des ConTract-Managers zum Log

Die Abbildung 9.4 enthällt die Schnittstellenfunktionen des ConTract-Managers zum Log. Der ConTract-Manager hängt durch einen *writelog()*-Aufruff einen Satz in den Log-Puffer an. Diese Routine liefert als Ergebnis die Log-Sequenznummer des neuen Satzes zurück. Sie ist größer als jede andere lsn, die für Log-Sätze des bearbeiteten ConTracts vergeben wurde.

Der ConTract-Manager kann die Übertragung der im Log-Puffer geschriebene Sätze auf die Log-Tabelle durch den *flushLog()*-Aufruf veranlassen. Dabei werden alle Log-Sätze, deren Log-Sequenznummer kleiner als die überegebene *lsn* sind, in die Log-Tabelle gespeichert.
In einigen Situationen ist es notwendig, einen Satz direkt, d.h. ohne Zwischenspeicherung im Puffer, in die Log-Tabelle zu schreiben. Für diese Fälle besitzt der ConTract-Manager eine weitere Routine, die *flushlogrec()*. Diese Routine hat den Nebeneffekt, daß auch alle im Log-Puffer befindlichen Daten in die Log-Tabelle übertragen werden. Beide flushlog-Aufrufe bewirken, daß der Log-Puffer wieder frei wird.

```
long readLog( IN  ConTract_Id  ADT_gid,
              IN  rectype    long,
              IN  lsn         LSN,
              OUT recPtr     ptr_logrec_list);

long writeLog ( IN  ConTract_Id  ADT_gid,
                IN  recPtr      ptr_LOGREC,
                IN  reclength   long,
                IN  flush       boolean,
                OUT lsn         LSN);

long flushLog( IN  ConTract_Id  ADT_gid);

long flushLogrec( IN  ConTract_Id  ADT_gid);

long checkpoint( IN  ConTract_Id  ADT_gid);
```

Abbildung 9.4: Log-Schnittstelle des ConTract-Managers

Der ConTract-Manager liest Sätze aus dem Log durch den Aufruf *readLog()*. Dieser Aufruf ermöglicht dem ConTract-Manager, nicht nur einen, sondern mehrere Sätze durch einen einzigen Log-Zugriff zu lesen. Durch den Parameter *rectype* kann der ConTract-Manager spezifizieren, welche Objekte (Events, Steps, etc.) er aus dem Log entnehmen will. Die Werte dieses Parameters entsprechen den Werten, die der Parameter *key* eines Log-Satzes annehmen kann (siehe Abschnitt **??**. Die übergebene Log-Sequenznummer gibt ein weiteres

Kriterium zum mengenorientierten Zugriff. Es sollen nur die Sätze betrachten werden, deren *lsn* größer ist als die übergebene. Damit wird es z.B möglich, nur auf Log-Daten zuzugreifen, die der ConTract-Manager nach dem letzten Checkpoint protokolliert hat.

Die Routine Checkpoint sichert den Zustand aller dynamischen ConTract-Daten.

9.4 Zusammenfassung

In diesem Kapitel wurde ein erster Ansatz für die Implementierung eines Protokolls skizziert, das dem ConTract-Manager ermöglichen soll, den Zustand eines von ihm bearbeiteten ConTracts beim Wiederanlauf zu rekonstruieren. Weil das Ziel dabei war, die Anforderungen nach einer *robusten* Kontollflußabwicklung zu erfüllen, war, hat man sich beim Entwurf dieses Protokolls an der Arbeitsweise und die Datenstrukturen des ConTract-Managers orientiert.

Durch die Implementierung dieses Protokolls ist sichergestellt, daß ein ConTract auch nach einer durch einen Systemfehler herbeigeführten Unterbrechung in endlicher Zeit terminieren kann. Voraussetzung dafür ist, daß der Fehler, der die Unterbrechung verursacht hat, den Zustand der auf den nichtflüchtigen Teil des Speichers befindlichen Objekte nicht manipuliert hat. Im einzelnen hat man folgende Anforderungen erfüllen können:

- Durch die Protokollierung jedes aktivierten ConTracts in die *config_table* kann ein in der APRICOTS-Umgebung aktivierter Vorgang niemals spurlos verschwinden. Der ConTract-Prozessing-Monitor hat dadurch die Möglichkeit beim Wiederanlauf des Systems, die unterbrochenen Vorgänge zu reaktivieren.

- Durch die Verwendung des nichtflüchtigen Speichers zur Protokollierung sind die Log-Daten und damit der Zustand eines ConTracts gegen viele häufig auftretende Fehler geschützt.

- Die verwendete Speicherhierarchie, die Unterteilung des nichtflüchtigen Logs und die Generierung von Sicherungspunkten tragen dazu bei, daß die Leistungsfähigkeit des Systems sowohl während des normalen Betriebs, als auch während des Wiederanlaufs nicht spürbar vermindert wird.

- Checkpoints können auch dazu beitragen, den für die Protokollierung benötigten "on-line"-Speicher in Grenzen zu halten: Daten, die der ConTract-Manager vor dem letzten Checkpoint protokolliert hat, können auf den Archivspeicher verlagert werden.

- Die Verwendung von Transaktionen ermöglicht die Realisierung *atomarer flushlog*-Operationen, die auch dazu beitragen, die Komplexität des Wiederanlaufverfahrens und damit auch die Kosten beim Wiederanlauf zu minimieren.

Wie oben bereits erwähnt, beinhaltet das Fehlermodell, das man beim Entwurf dieses prototypischen Logs zugrundegelegt hat, nur einen Teil der Fehler, die in realen Systemen auftreten können. Das bedeutet, daß durch dieses Protokoll der Zustand eines ConTracts bzw. ConTract-Managers nicht gegen alle möglichen Fehler geschützt werden kann. Aus

diesem Grund hat der in diesem Kapitel skizzierte Anzatz einige Fragen zur Diskussion offengelassen. Diese Fragen betreffen die Verfügbarkeit und Zuverlässigkeit des Log-Services, sowie die Verknüpfung der weiteren APRICOTS-Komponenten zum Log, etc.
Auf alle diese Fragen muß man eine Antwort geben, wenn man das Ziel erreichen will, die *forward Recovery* nach jedem möglichen Fehler durchführen zu können. Die Realisierung der verschiedenen Lösungsansätze für die Erfüllung dieses Ziels, sollte in der Diskussion zur Implementierung eines *globalen* Log-Services in das APRICOTS-System miteinbezogen werden.

Kapitel 10

Überlegungen zur Realisierung eines globalen Logs in APRICOTS

In diesem Kapitel werden einige Aspekte diskutiert, die man bei der Implementierung eines globalen Log-Services berücksichtigen sollte. In den ersten Abschnitten wird auf die Notwendigkeit eines globalen Logs und auf die Vorteile hingewiesen, die sich aus der Realisierung einer globalen Instanz zur Verwaltung der Log-Daten ergeben. Anschließend wird ein Überblick über die Verfahren gegeben, die man zur Realisierung eines zuverlässigen, verteilten Log-Services anwenden kann. Im weiteren werden kurz die Konsequenzen skizziert, die sich für den ConTract-Manager im Bezug auf das in den vorausgegangenen Kapiteln beschriebene Protokollierungs- und Wiederanlaufverfahren aus der Implementierung eines globalen Logs ergeben. Als letztes wird das prinzipielle Vorgehen beim Wiederanlauf des ConTract-Systems behandelt.

10.1 Warum ist ein globales Log notwendig?

Jede APRICOTS-Komponente muß gewisse Informationen über ihre Aktivitäten dauerhaft protokollieren, damit sie beim Wiederanlauf nach einem Fehler ihren Zustand rekonstruieren kann. Für die Protokollierung dieser Informationen stehen zwei Möglichkeiten zur Verfügung: Ein globales Log für alle Komponenten, oder jede Komponente soll ihr eigenes Log führen. Obwohl die zweite Möglichkeit, die Portierung neuer Komponenten, die spezielle Anforderungen an die Verwaltung der Protokolldaten stellen, enorm vereinfacht, sprechen für die Realisierung eines globalen Logs folgende Gründe:

- Die Systemadministration wird durch die Verwaltung eines einzigen Logs viel einfacher.
- Man kann Komponenten implementieren, die eine weniger komplexe Struktur aufweisen, weil man ihnen die Aufgaben der Log-Verwaltung entnimmt.
- Es erlaubt den Aufbau eines leistungsfähigeren Systems, da man für die Recovery nur die Log-Sätze eines Logs betrachten muß. Würde z.B. jeder Resource-Manager

sein eigenes Log führen, wären die Log-Sätze einer Transaktion über mehrere Logs verteilt. Der Transaktions-Manager müßte dann beim Wiederanlauf mehrere Logs für die Recovery einer einzigen Transaktion bearbeiten. Damit würden die Kosten für die Transaktions-Recovery enorm steigen. Zusätzlich wäre dann auch bei rein lokalen Aktivitäten ein Zweiphasen-Commit-Protokoll zur Koordinierung der Protokollierungsoperationen notwendig. Das würde aber zu Verzögerungen während des normalen Ablaufs führen.

10.2 Der globale Log-Manager

Der Log-Manager ist die Komponente des APRICOTS-Systems, der die administrativen Aufgaben für das Log übernimmt. Zusätzlich bietet er jeder anderen Komponente eine Schnittstelle an, die ihr den Zugriff auf das Log erlaubt. In der Regel könnte jede Komponente mit einer Schnittstelle versehen werden, über die sie direkt auf das Log zugreifen kann. Aus folgenden Gründen ist es aber vorzuziehen, eine separate Komponente für die Verwaltung des Logs vorzusehen:

- Da mehrere Komponenten auf den selben Speicherbereich zugreifen, müssen diese Zugriffe synchronisiert werden, um die Konsistenz und Integrität der Daten zu bewahren. Eine Komponente, die als Monitor auf dem Log fungiert, ist es eine gute Lösung dafür.

- Der Log-Manager ermöglicht eine Abstraktion von unterschiedlich strukturierten Objekten, die von den Komponenten verwaltet werden, die auf das Log zugreifen. Er bietet eine allgemeine Log-Satzstruktur, und garantiert dabei die korrekte Verwaltung dieser Struktur. Über seine wohldefinierte Schnittstelle, können weitere Komponenten angeschlossen werden, die neue Objekttypen einführen und verwalten.

- Er garantiert durch sichere Protokolle [12] zum Schreiben und Lesen hohe Zuverlässigkeit und Verfügbarkeit des Log-Service. Dies ist notwendig, weil das Log die einzige Kopie ist, die dauerhafte Änderungen an Objekten enthält.

- Beim System-Restart ist keine Systemeinheit funktionsfähig. Der Log-Manager besitzt als einzige Komponente Mechanismen, die ihm ermöglichen, den Log wieder in einen konsistenten Zustand zu bringen. Auf diesen Zustand können die übrigen Systemkomponenten aufsetzen, um ihre Recovery durchzuführen.

10.3 Der Log-Manager in der APRICOTS-Architerktur

Durch den Log-Manager realisiert man einen weiteren Resource-Manager im ConTract-System, dem die Rolle als Verwalters der im Log gespeicherten Objekte zugesprochen wird. Seine Besonderheit liegt darin, daß seine Dienste von allen anderen Komponenten des APRICOTS-Systems in Anspruch genommen werden. Daher ist eine intensive Interaktion zwischen dem Log-Manager und den anderen Komponenten zu erwarten, was mit hohen Kommunikationskosten und Verzögerung der Vorgänge im System verbunden ist.

Um diese Kosten zu minimieren, ist es sinnvoll, das Modul des Log-Managers bzw. Teile davon in den Code jeder anderen Komponente einzubinden. Auf diese Weise ergibt sich nur eine *prozeßlokale* [1] Kommunikation zwischen Log-Manager und den anderen Komponenten. Prozeßlokale Kommunikation stellt die einfachere und billigste Möglichkeit dar, im APRICOTS Aufträge zu senden. Die Abbildung 10.1 stellt diesen Vorschlag graphisch dar.

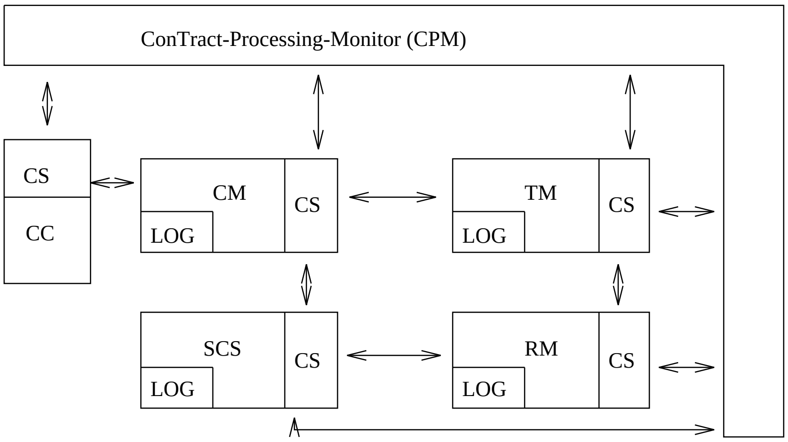

Abbildung 10.1: Der Log-Manager in der APRICOTS-Umgebung

10.4 Implementierungsalternativen eines globalen Log-Services

In diesem Abschnit wird auf gewisse Probleme bzw. Alternativen hingewiesen, die im Rahmen der Diskussion für die Implementierung eines globalen Log-Services beachtet werden sollen. Dabei wird nicht auf die Details eingegangen, sondern es werden nur gewisse Punkte angesprochen. Für weitere Details wird der Leser auf die angegebene Literatur verwiesen.

10.4.1 Struktur des Logs

Ein sehr wichtiger Aspekt bei der Organisation eines globalen Logs ist die Verfügbarkeit der Log-Daten. Die Konsistenzeigenschaft des ConTract-Modells fordert, daß ein ConTract fortgesetzt werden soll, auch wenn ein Knoten für immer aus dem System verschwindet. Um die Verfügbarkeit der Log-Daten in so einem Fall sicherzustellen, ist es notwendig, die Daten auf verschiedene Medien mit unabhängigen Fehlerhäufigkeiten zu speichern.

Eine Möglichkeit dafür ist die Speicherung der Daten auf gespiegelten Platten. Dazu kann man einen Log-Server implementieren, der zwei verschiedene Platten beschreibt. In diesem Fall ist aber die Verfügbarkeit des Log-Servers ein kritischer Faktor für die Verfügbarkeit der anderen Systemkomponenten. Fällt der Log-Server aus, können auch die restlichen Komponenten nicht mehr protokollieren bzw. weiter arbeiten. Dies kann vermieden werden,

[1] Siehe [17]

indem man die Log-Daten repliziert. In diesem Fall verwendet man N Log-Server, wobei, jeder Log-Satz auf mindestens W verschiedene Server gespeichert wird. W wird in der Regel als (N+1)/2 gewählt. Damit kann das System mehr als N/2 Knoten- bzw. Log-Server Ausfälle verkraften.

Das Lesen eines Log-Records kann aber teuer sein, da man in der Regel aus (N-W+1) Log-Server lesen muß, um sicher zu sein, daß er den geforderten Satz erhält. Das heißt, man muß zum Lesen eines Satzes mehrere readLog-Nachrichten senden. Dies kann man umgehen, wenn der Log-Client (bzw. Log-Manager) Informationen im Hauptspeicher darüber hält, welche Server welche Log-Sätze gespeichert haben. Damit kann nur einer davon zum Lesen ausgewählt werden.

Ein weiteres Problem bei der mehrfachen Speicherung der Log-Daten tritt auf, wenn durch einen Systemfehler ein Log-Satz durch weniger als W Server geschrieben wurde. Das bedeutet beim Restart sollte der Log-Manager sein Log-Ende finden können und die Konsistenz der Log-Daten wiederherstellen. Dies kann erfolgen, indem der Log-Manager die Replizierung oder Eliminierung der partiell geschriebenen Log-Sätze veranlaßt.

Während die Spiegelung der Log-Sätze einfache Verfahren zur Verwaltung des Logs erlaubt, erhöht die Replizierung der Daten die Verfügbarkeit des Logs. Die Vorteile bzw. Nachteile dieser Verfahren kann man durch die Implementierung hybrider Logs ausnutzen bzw. eliminieren. Hierbei schreibt man die Log-Sätze auf gespiegelten Platten, dafür wurden aber die Einträge in der Directory auf mehrere Log-Server repliziert. Weitere Details über die obengenannten Verfahren findet man in [8].

10.4.2 Kommunikation

Die Kommunikation zwischen dem Log-Manager und den restlichen APRICOTS-Komponenten, sowie zwischen dem Log-Manager und den Log- bzw. Disk-Servern zur Speicherung der Log-Daten, ist ein wichtiger Faktor für die Leistungsfähigkeit des Systems.

Die Schnittstelle des Log-Managers muß sowohl synchrone als auch asynchrone Nachrichten anbieten. Während die Pufferung eines Log-Satzes nicht zur Blockierung der Komponente führen soll, die den Log-Satz geschrieben hat, muß bei flushlog-Operationen die Verarbeitung erst dann weitergehen, wenn die Übertragung der Daten auf dem stabilen Speicher abgeschlossen ist. Dabei soll das Kommunikationsprotokoll die Übertragung hoher Datenarten unterstützen, was im Falle von Transaktionen mit hohen Änderungsraten notwendig ist.

Für die Implementierung eines replizierten Logs ist weiterhin von Vorteil, wenn das Kommunikationssystem Multicast-Protokolle zur Verfügung stellt, um parallele *writes* auf mehrere Log-Server zu unterstützen. An dieser Stelle muß man bemerken, daß sichere Kommunikatiosprotokolle die Voraussetzung für einen zuverlässigen Log-Service sind.

10.4.3 Log-Satz Struktur

Ein weiterer Aspekt der Implementierung eines globalen Log-Services ist die Darstellung der Daten auf den nichtflüchtigen Speicher. Die Struktur eines Log-Satzes soll vom Typ der gespeicherten Objekte abstrahieren. Damit können weitere Komponenten im System angeschlossen werden, die verschiedene Datenobjekte verwalten.

Der wahlfreie Zugriff auf einzelne Sätze, sowie als auch der mengenorientierte Zugriff (Cursor) auf Sätze eines bestimmten Resource-Managers sollen unterstützt werden. Im weiteren soll es möglich sein, den Log vorwärts wie rückwärts zu lesen.

Zur Realisierung des WAL-Verfahren sollen eindeutige monoton wachsende LSN zur Indentifizierung der Log-Sätze vergeben werden. Die Monotonieeigenschaft der LSN's kann zum direkten Zugriff auf die Log-Sätze ausgenutzt werden. SQL-Tabellen können den Zugriff auf die Log-Sätze optimieren. Eine "low-level"-Schnittstelle zum nichtflüchtigen Speichermedium soll ebenfalls zur Verfügung gestellt werden, um die Log-Informationen beim Wiederanlauf auf jeden Fall wiederauffinden zu können.

Um das Recovery-Verfahren zu vereinfachen, kann man feste Disk-Partitionierungen zur Speicherung bestimmter für den Wiederanlauf wichtiger Informationen wie z.B. den Log-Anchor reservieren. [2]. Die Verwendung von Disk-Partitionen kann auch zur Minimierung der Kosten bei der Übertragung von Log-Records auf den stabilen Speicher beitragen. Die WADS-Technik, die GRAY und REUTER in [12] beschreiben, nutzt solche Partitionen aus.

10.4.4 Verwaltung der Log-Daten

Die Verwaltung des für das Log verwendeten Speichers ist ein weiterer Faktor, der die Leistungsfähigkeit des Log-Services bestimmt. Die Aktivitäten des Systems können blockiert werden, falls der nichtflüchtige Speicher mit Log-Daten gefüllt ist. Um dies zu vermeiden, muß der Log-Manager nur soviele Log-Daten "online" halten, wie für das UNDO von Transaktionen im normalen Betrieb und für den Wiederanlauf des Systems benötigt werden. Die restlichen Daten kann man archivieren, und so den freigewordenen Speicherplatz für die Protokollierung neuer Daten verwenden. Dazu soll der Log-Manager die kleinste LSN (low_water_mark) von allen aktiven Transaktionen kennen, die zum UNDO- bzw. REDO-dieser Transaktion im Fehlerfall benötigt wird. Alle Log-Sätze, die vor dem Satz mit dieser LSN protokolliert werden, können auf den Archivspeicher übertragen werden.

Um die Kosten für das Archivieren, sowie für die Reservierung neuen Speicherbereichs für die Log-Daten zu minimieren, kann man die Platten, die Log-Daten enthalten, gestaffelt archivieren, so daß jederzeit nur eine Platte archiviert wird, während die anderen für die Speicherung weiteren Log-Sätze zur Verfügung stehen.

10.4.5 Sicherheit

Eine weitere Frage, die im Bezug auf einen verteilten Log-Service diskutiert werden soll, ist die Sicherung der Log-Daten. Hierbei müssen Maßnahmen getroffen werden, um den

[2]Der Log-Anchor beschreibt den aktuellen Zustand der Log-Tabellen bzw. Files [12]

Zugang unberechtigter Personen zu den Speichermedien zu verhindern.

Weiterhin sollte die Korruption und Verfälschung der Log-Daten während der Übertragung im Netz nahezu ausgeschlossen werden. Spezielle Verschlüsselungsverfahren (public_key, privat_key, End-to-End-Encryption), sichere Paßworte, sowie der Einsatz spezieller Server (z.B. Kerberos) sind mögliche Ansätze, die man dabei berücksichtigen sollte.

10.5 Konsequenzen aus der Implementierung eines globalen Logs

Im Kapitel 9 haben wir eine Lösung für die Implementierung eines Logs für den ConTract-Manager beschrieben. Den Ansatz dabei haben wir entwickelt, indem wir die Arbeitsweise und Datenstrukturen des Contract-Manager berücksichtigt haben. Aus diesem Grund ist dieses Verfahren nicht, bzw. nicht ohne weitere Modifikationen anwendbar, wenn man beabsichtigt, ein globalen Log-Service zu implementieren.

10.5.1 Zugriff auf das Log

Der mengenorientierte Zugriff auf Objekte eines bestimmten Typs ist nicht mehr möglich, da der Log-Manager keine Kenntnis über den Typ dieser Objekte hat (Datenabstraktion). Man kann z.B. nicht durch eine einzige Anfrage auf alle Step-Instanzen zugreifen, die im Zustand "commitet" sind, weil der globale Log-Manager nicht wissen kann, ob es sich bei einem Log-Satz um eine Stepinstanz, ein Event etc. handelt. Aus demselben Grund ist es auch nicht möglich, jeweils die letzte Änderung an ein Objekt, z.B. Event, aus dem Log zu entnehmen. Das bedeutet, der ConTract-Manager muß beim Wiederanlauf alle Log-Sätze (nach dem letzten Sicherungspunkt) bearbeiten, um den Zustand seiner Datenstukturen zu rekonstruieren (DO-REDO-Protokoll). Das hat aber zur Folge, daß sich der Aufwand beim Wiederanlauf eines ConTracts erhöht.

10.5.2 Pufferung der Log-Daten

Eine weitere Konsequenz betrifft die Pufferung der Log-Daten. Im Abschnitt 7.4.1 haben wir beschrieben, daß alle Änderungen während der Triggerung einer Step-Instanz zunächst in einem Puffer gesammelt werden und unmittelbar vor dem darauffolgenden *schedule_stepinst_list*-Aufruf in einer atomaren Operation stabil protokolliert werden. Durch die Verwendung eines globalen Logs kann es aber vorkommen, daß manche dieser Änderungen stabil protokolliert werden, noch bevor die Triggerung beendet ist. Das ist der Fall, wenn der Transaktions-Manager oder eine andere Komponente die Übertragung des Log-Managers-Puffers auf den stabilen Speicher veranlaßt hat. Wird der ConTract nach dieser Übertragung unterbrochen, dann findet der ConTract-Manager in seinem Log Daten, die zum Rekonstruieren eines falschen Zustands führen würden.

Betrachten wir dazu das Beispiel aus der Abblidung 2.1. Nehmen wir folgende Situation an:

Die Steps S4, S5 werden ausgeführt während Step S3 schon beendet und die Triggerung der entsprechenden Step-Instanz abgeschlossen ist. Durch diese Triggerung hat das Start-Event von S6 ein Token erhalten. Trifft jetzt die *step_finished*-Meldung für den Step S4 ein, wird bei der nachfolgenden Triggerung die Anzahl der Tokens für das Start-Ereignis des Steps S6 auf 2 angehoben. Zu diesem Zeitpunkt schreibt der Log-Manager seinen Puffer aus, bevor der ConTract-Manager die Triggerung abgeschlossen hat, d.h. bevor er den Step S4 im Zustand "commited" protokolliert hat. Wenn an dieser Stelle der ConTract unterbrochen wird, dann begegnet der ConTract-Manager beim Wiederanlauf der Situation, die durch die folgende Tabelle beschrieben wird:

Objekt	Log-Eintrag	Bemerkung
S3	committed	Die zugehörige Step-Transaktion ist "committed". Die Triggerung ist abgeschlossen
S4	running	Die zugehörige Step-Transaktion ist "commited". Die Triggerung ist nicht abgeschlossen.
S5	running	Die zugehörige Step-Transaktion ist abgebrochen
Start-Ereignis von S6	Anzahl "Tokens" = 2	

Der ConTract-Manager muß die Triggerung bzw. Reaktivierung von S4 und S5 anstoßen. Während der Triggerung von S4 erhält aber das Start-Ereignis des Steps S6 noch ein Token und damit wird dieser Step aktiviert. Dies sollte aber gerade nicht passieren, daß S6 aktiviert, wird bevor S5 erfolgreich beendet ist.

10.5.3 Integration des ConTract-Managers

Eine weitere Frage, die man in der Diskussion über die Realisierung eines globalen Logs berücksichtigen sollte, betrifft den Wiederanlauf des ConTract-Managers: Wie soll der Wiederanlauf eines ConTract-Managers organisiert werden ? Soll der ConTract-Manager seine Recovery eigenständig oder im Rahmen der Transaktions-Recovery durchführen ? Letzteres würde allerdings bedeuten, daß der ConTract-Manager als ein Ressource-Manager betrachtet wird, der in das UNDO-REDO-Protokoll mit einbezogen wird.

Da wir in den vorausgegangenen Abschnitten bzw. Kapiteln die Ansätze zur Realisierung der ersten Möglichkeit diskutiert haben, wenden wir uns auf der zweiten Möglichkeit zu.

Um die Recovery des ConTract-Managers als Teil der Transaktions-Recovery durchzuführen, muß der ConTract-Manager im normalen Betriebs auch seine Log-Sätze im Rahmen der Protokollierung der einzelnen Transaktionen bzw. Step-Transaktionen mitprotokollieren. Der ConTract-Manager aber, wie schon im Abschnitt 7.1 diskutiert wurde, protokolliert keine Änderungen im Rahmen einer Transaktion, sondern nur Ereignisse die nach bzw. vor der Änderung des Zustands einer Transaktion (Step-Transaktion) eintreten. Das bedeutet, daß man eine Möglichkeit finden muß, die Protokollsätze des ConTract-Managers in der Protokollierung im Rahmen der Änderungen der Step-Transaktionen zu integrieren.

Dazu gäbe es folgende Möglichkeit: Der ConTract-Manager führt die Triggerung einer erfolgreich ausgeführten Step-Instanz bevor er das Ende der (EOT_request-Aufruf) zugehörigen Step-Transaktion fordert. Dabei protokolliert er die Änderungen als ein Teil der Ausführung dieser Step-Transaktion. Sobald die Triggerung beendet ist, fordert er das "commiten" der Step-Transaktion . Im Rahmen der Abwicklung des Zweiphasen-Commit-Protokolls, aktiviert er die Nachfolger-Step-Instanzen bzw. die neuen Step-Transaktionen.

Ist das Zweiphasen-Commit-Protokoll erfolgreich abgewickelt, sind keine weitere Maßnahmen erforderlich. Andernfalls müssen die Ergebnisse der triggernden Step-Transaktion zurückgesetzt werden. Das bedeutet, der ConTract-Manager muß die Änderungen an seinen Daten zurücksetzen und die Ausführung der neu aktivierten Step-Instanzen abbrechen. Erst dann kann er die Fehlerbehandlung (bzw. Reaktivierung) für die triggernde Step-Instanz anstoßen. Diese Art der Organisation weist aber große Nachteile während des normalen Betriebs sowie während des Wiederanlaufs auf.

Aus der obigen Beschreibung erkennt man, daß die Behandlung einer abgebrochenen Step-Transaktion enorme Kosten verursachen kann. Die Teilnahme des ConTract-Managers an der Abwicklung des Commit-Protokolls, führt ebenfalls zur Verzögerung einer ConTract-Abwicklung.

Wird der ConTract während der Triggerung durch einen Fehler unterbrochen, müssen beim Wiederanlauf nicht nur die dynamischen ConTract-Daten zurückgesetzt werden, sondern auch die beendete Step-Instanz bzw. ihre Step-Transaktion zurückgesetzt und erneut wiederholt werden. Im Vergleich dazu braucht der ConTract-Manager in diesem Fall nur die Triggerung anzustoßen, wenn das Verfahren angewendet wird, das in den Kapiteln 7 bis 9 beschrieben ist.

Ein weiterer Nachteil ist die Notwendigkeit der Implementierung eines UNDO-Protokolls für den ConTract-Manager. Dies kann entfallen, wenn der ConTract-Manager seine eigene Recovery durchführt, da er nur den zuletzt gültigen Wert für seine Objekte braucht, um einen unterbrochenen ConTract fortsetzen zu können.

Aus der obigen Diskussion ergibt sich, daß die Einbeziehung des ConTract-Managers in das UNDO-REDO-Protokoll, das für Transaktions-Recovery in einem Transaktion-Verabeitungssystem notwendig ist, hohe Kosten während der Verarbeitung sowohl im normalen Fall als auch beim Wiederanlauf verursachen kann. Zusätzlich sind komplexe Algorithmen zur Realisierung eines solchen Ansatzes erforderlich. Aus diesen Gründen ist es vorzuziehen, ein eigenes Log für den ConTract-Manager zu implementieren.

10.6 Organisation des Wiederanlaufs im APRICOTS

Dem Enwurf eines Wiederanlaufprotokolls soll eine Untersuchung der Abhängigkeiten vorausgehen, die zwischen den verschiedenen APRICOTS-Komponenten bestehen. Damit kann man vermeiden, daß während des Wiederanlaufs Zyklen auftreten, die zur Blockierung von Komponenten führen würden. Andererseits kann man den Wiederanlauf beschleunigen, wenn man gewisse Komponenten parallel reaktiviert.

Im Kapitel 8 ist für die Korrektheit des beschriebenen Wiederanlaufverfahrens vorausgesetzt, daß das System sich in einem (Step-)Transaktionskonsistenten Zustand befindet und alle benötigten Komponenten betriebsfähig sind. Dies ist die Basis unserer Untersuchung. Dabei nehmen wir an, daß der ConTract-Manager seine eigene Recovery-Prozedur durchläuft. Selbstverständlich ist überhaupt eine Recovery nur dann möglich, wenn das Betriebssystem seine Dienste anbieten kann.

Wie bereits erwähnt, ist der ConTract-Processing-Monito zuständig für das Erzeugen und die Überwachung der APRICOTS-Konfiguration. Nachdem das Betriebssystem den Wiederanlauf abgeschlossen hat, soll er anhand seiner Konfigurationstabelle die APRICOTS-Prozesse reaktivieren. Er muß dabei berücksichtigen, daß er die ConTract-Manager dann startet, wenn die Transaktions-Recovery durchgeführt worden ist. Für die Transaktions-Recovery sind der Transaktions-Manager und die Resource-Manager zuständig. Weil der ConTract-Processing-Monitor keine Kenntnis darüber hat, wie sich die einzelnen Resource-Manager für die Abwicklung einer Transaktion aufrufen, ist er nicht in der Lage, die Reihenfolge für die Reaktivierung der Resource-Manager zu bestimmen. Das bedeutet, daß man gewisse Schnittstellen beim Entwurf der Resource-Manager vorsehen soll, über die sie sich beim Wiederanlauf synchronisieren können. [3] Damit kann der ConTract-Processing-Monitor den Transaktions-Manager und die Resource-Manager parallel reaktivieren. Dabei muß aber sichergestellt sein, daß die Gesamtheit der protokollierten Daten verfügbar ist.

Sobald die Transaktions-Recovery abgeschlossen ist, kann der ConTract-Processing-Monitor die ConTract-Manager-Prozesse reaktivieren, und damit die *forward*-Recovery aller abgebrochenen ConTracts anstoßen. Parallel dazu kann er die Aktivierung weiterer ConTracts zulassen, da die parallele Ausführung der ConTracts im System erlaubt ist.

An dieser Stelle muß bemerkt werden, daß die Dienste des Kommunikationssystems in Anspruch genommen werden können, sobald die Mailboxen der APRICOTS-Komponenten aufgebaut sind. Das ist aber nach dem Restart des Betriebssystems gewährleistet.

Zusammenfassend lassen sich folgende Schritte für den Wiederanlauf angeben:

- Schritt 1: Betriebsystem Restart.
- Schritt 2: Rekonstruieren des Logs.
- Schritt 3: Der ConTract-Prozessing-Monitor muß anhand der der globalen Konfigurations-Relation die APRICOTS-Prozesse in der Reihenfolge starten, wie sie weiter angegeben wird.
- Schritt 4: Den Transaktion-Manager und die Resource-Manager. Diese Komponeneten müssen sich evtl. während ihre Recovery abstimmen [12].
- Schritt 5: Die Step-Computation-Server reaktivieren. Dies kann auch parallel zum vorigen Schritt erfolgen.
- Schritt 6: Als letztes muß der ConTract-Prozessing-Monitor alle ConTract-Manager-Prozesse reaktivieren, und damit die *forward Recovery* aller abgebrochenen ConTracts anstoßen.

[3] Einen solchen Mechanismus beschreiben Gray und Reuter in [12].

10.7 Zusammenfassung

Die Implementierung eines globalen Logs, kann die Bildung eines leistungsfähigeren Systems unterstützen. Dabei sind viele Aspekte zu berücksichtigen, die die Organisation und Verwaltung der Log-Daten betreffen.

Dem Entwurf eines globalen Logs muß aber eine genaue Untersuchung der Anforderungen vorausgehen, die die einzelnen APRICOTS-Komponenten an die Verwaltung der Log-Daten stellen. Damit kann man hohe Kosten sowohl beim normalen Ablauf, als auch beim Wiederanlauf vermeiden.

Im Rahmen dieser Untersuchungen soll konkreter geprüft werden, ob die Integration gewisser Komponenten in einem globalen Service nachteilig auf das Verhalten des Systems insgesamt wirkt. In diesem Fall sollte man für diese Komponente ein privates Log vorsehen. Zur Verbesserung des Gesamtverhaltens des APRICOT-Systems könnte auch die Organisation des Wiederanlaufs beitragen. Aus diesem Grund sollten die gegenseitigen Beziehungen dieser Komponenten beim Entwurf des Wiederanlaufverfahrens detailliert untersucht werden.

Kapitel 11

Resumee

11.1 Zusammenfassung

Der Zweck dieser Arbeit war, Probleme anzusprechen, die man in der Diskussion über die Realisierung einer robusten Verarbeitung verteilter Aktivitäten im APRICOT-System berücksichtigen sollte. Es war nicht beabsichtigt ein implementierungsreifes Lösungskonzept zu entwerfen. Vielmehr sollten unter Berücksichtigung der Eigenschaften des ConTract-Modells Anregungen gegeben werden, die als Basis zur Bildung eines robusten Systems dienen soll, das die Anforderungen vieler Anwendungen in bezug auf fehlertolerante Abwicklung langlebiger, verteilter Vorgänge auf heterogenen Umgebungen genügt. Der Kernpunkt dabei war der Aspekt der forward-Recovery. Das System soll nicht nur Teilergebnisse unvollständig ausgeführter Aktionen zurücksetzen, sondern vielmehr Vorgänge fortsetzen können, die durch nicht vorhersehbare Ereignisse in einem undefinierten Zustand beendet werden. Dabei wurde festgestellt, daß diese Ereignisse

- den Abbruch
- die Blockierung
- und das abrupte Beenden

der Ausführung eines ConTracts bewirken können. Als Ausgangspunkt für die Ansätze zur Behandlung dieser Fehler dienten die Aussagen der Konsistenzeigenschaft des ConTracts-Modells [26].

Aus der Diskussion zur Fehlerbehandlung hat sich folgendes ergeben:

- Um eine Störung akut zu behandeln, muß man in der Lage sein diese zu erkennen. Es muß auszumachen sein, welche die Ursache für diese Störung war.
- Falls die Störung anwendungsbedingt war, muß ein anwendungsspezifischer Recovery-Manager die Situation behandeln können.
- Jede APRICOTS-Komponente muß in der Lage sein, ihren Zustand nach einem Ausfall zu rekonstruieren. Das gilt sowohl im Falle eines Prozeßausfalls, als auch im Falle eines Knotenausfalls.

- Insbesondere soll der ConTract-Manager den Verarbeitungszustand eines ConTracts rekontruieren können, so daß er diesen zuverlässig fortsetzen kann (forward-Recovery).
- Die automatische Replizierung der APRICOTS-Prozesse soll möglich sein, um die Blockierung eines ConTracts zu vermeiden.
- Die redundante Speicherung der Daten ist eine unbedingt erforderliche Maßnahme, um Knotenausfälle zu tolerieren.

Es ist wichtig, den Leser an dieser Stelle daran zu erinnern, daß die gesamte Diskussion in dieser Arbeit auf die Garantie aufbaut ist, daß das Kommunikationssystem die Nachrichten zuverlässig transportiert.

Im weiteren Verlauf dieser Arbeit wurde der Kernpunkt dieser Arbeit (forward-Recovery) nochmals aufgegriffen und ein Verfahren realisiert, das in der gegenwärtige Implementierung des APRICOT-Systems dem ConTract-Manager ermöglicht, den Verarbeitungszustand eines ConTracts zu protokollieren und beim Wiederanlauf zu rekontruieren.

Die anschließenden Überlegungen gelten der Realisierung eines globalen Logs im APRICOTS. Dabei wird auf weitere Aspekte hingewiesen, die man beim Entwurf eines Lösungsansatzes zur fehlertoleranten Abwicklung der im APRICOT-System modellierten Vorgänge behandeln sollte.

11.2 Ausblick

Im Rahmen dieser Arbeit war es nicht möglich alle angesprochenen Probleme im Detail zu betrachten und vollständige Verfahren zu ihrer Lösung anzugeben. Im Hinblick auf die Realisierung des Konzeptes der forward-Recovery sind noch offene Fragen geblieben.

Die erste Frage, die hier nicht behandelt wurde, betrifft die Fehlererkennung. Es müssen Protokolle für das Erkennen aufgetretener Fehler und ihrer Ursachen entwickelt werden.

Die Entwurfs- und Implementierungsaspekte des anwendungsspezifischen Recovery-Managers sowie seine Integration im APRICOTS geben Anlaß zu weiteren Überlegungen.

Die Organisation eines globalen Logs im APRICOTS ist die Frage, die besonderer Aufmerksamkeit bedarf. Es muß herausgestellt werden, ob ein globales Log die Anforderungen aller Komponenten des APRICOT-Systems befriedigt. Vorkehrungen müssen getroffen werden, die die Verfügbarkeit der Log-Daten in den häufig anzutreffenden Fehlersituationen garantieren. Die Archivierung und Sicherheit dieser Daten sind weitere Gesichtspunkte in der Diskussion eines globalen Logs.

Weil in einem Transaktionsverarbeitungssystem sehr intensive Log-Operationen stattfinden, sollte man bei der Wahl der in Frage kommenden Alternativen darauf achten, daß das Log nicht zu einem Engpaß im APRICOT-System führt.

Literaturverzeichnis

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

[2] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. Concurrency Control and Recovery in Database Systems, Addison-Wesley 1987.

[3] Philipp A. Bernstein et al. DECdta-Digital's Distributed Transaction Processing Architecture. Digital Technical Journal, Vol.3, No. 1, Winter 1991.

[4] H. Burkert. Systemarchitektur und Betriebsmodell für das verteilte Transaktionssystem APRICOT. Diplomararbeit Nr. 812: Institut für Parallele und Verteilte Höchstleistungsrechner, Stuttgart, August 1991.

[5] H.Burkert, B. Reichle. Entwicklung eines Daten- und Verarbeitungssystem für den Kontext. Universität Stuttgart, Studienarbeit, 1990.

[6] Spector, A.Z u.a. Camelot: ADistributed Transaktion Facility for Mach and the Internet- An Interim Report, Technikal Report CMU-CS-87-129, Carnegie Melon University, Juni 1987.

[7] Stefano Ceri and Giusepe Pelagatti. Distributted Databases: Principles and Systems. McGraw-Hill 1984.

[8] Dean Spencer Daniels. Distributed Logging for Transaction Processing, CMU-CS-89-114, December 1988.

[9] Carey, M.J. u.a. The Architecture of the EXODUS Extensible DBMS, in Proc. Int. Workshop on Objekt-Oriented Database Systems, Pacific Grove, California , 1986.

[10] David K. Gifford. Weighted Voting for Replicated Data. In Proceedings of the Seventh Symposium on Operating System Principles, Pages 150-162. ACM, December 1979.

[11] Gray, J. The Transaction Concept: Virtues and Limitations. In Proc. of the 7th Int. Conf. on Very Large Database Systems, Cannes, Sept 1981.

[12] J. Gray and A. Reuter. Transaction Processing Systems: Concepts and Techniques. Morgan Kaufman Publishers,1992. Zur Veröffentlichung vorbereitet.

[13] Härder, T. and Reuter, A. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, 15(4),1983.

[14] Härder, T.; Rothermel,K. Concepts for Transaction Recovery in Nested Transactions, SIGMOD 1987.

[15] Meichun Hsu et al. An Execution Model for Active Base Management Systems. Proc. of int. Conf. on Data and Knowledge Bases, June 1988, Jerusalem.

[16] Klein, J., and Reuter, A. Migrating Transactions. Proc. IEEE Workshop on the Future Trends of Distributed Computing Systems, Hong Kong, Sept. 1988.

[17] Dieter Laube. Transaktionsorientierte Kommunikationsmechanismen im verteilten Transaktionssystem APRICOTS. Universität Stuttgart. 26. März 1992.

[18] A. Liebl et al. Sicherheitsaspekte des Betriebssystems UNIX. Informatik-Spektrum (1990), 13:191-203, Springer-Verlag.

[19] Rainer Lutz. Transaktionsverwaltung auf der Basis von Transaktionsbeziehungsnetzen. Universität Stuttgart, Diplomarbeit 1991

[20] Masmanidis, I. ConTract-Programmierung in Apricots: Sprachbeschreibung und Programmieranleitung. Universität Stuttgart, Arbeitspapier 1991.

[21] C. Mohan et al, ARIES; IBM Research Report, RJ6649, 19.01.1989

[22] Härder, T. (Hrsg.) The PRIMA Project, Design and Implementation of a Non-Standard Database System, SFB124 Research Report No. 26/88, Universität Kaiserslautern, 1988

[23] Reuter, A. Fehlerbehandlung in Datenbanksystem, Hnser Verlag 1981.

[24] A. Reuter. Performance Analysis of Recovery Techniques. ACM Transactions on Database Systems, Vol 9, No. 4, December 1984, Pages 526-559

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

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

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

[28] Schütz, H. Darstellung und Verwaltung von Beziehungen zwischen Transaktionen. Universität Stuttgart, Studienarbeit 1991

[29] Joost S. M. Verhofstad. Recovery Techniques for Database Systems. ACM Computing Surveys 10(2), June 1978.

[30] H.Wächter. Transaktionskonzepte und Recovery-Verfahren für Objektbanken. Universität Stuttgart, Diplomarbeit, 1989.

[31] H. Wächter. ConTracts: A Means for Improving Reliability in Distributed Computing. Proc. IEEE Spring Computer Conference 1991.

[32] Wächter H. Managing Transactions with Transaction Dependencies. Technical Report, University of Stuttgart, 1991.

[33] Wahl, P. Interpretative Abwicklung von Prädikat Transitionsnetze. Universität Stuttgart, Studienarbeit 1991.

[34] Michael Wayne Young. A Modular Architecture of Distributed Transaktion Prozessing. Presented at USENIX Technical Conference, Winter 1991, Dallas, Texas.

Anhang

Checkpoint-Relationen des ConTract-Managers

1. **Die Step-Instanzen Relation**

   In dieser Relation werden die aktiven bzw. suspendierten Step-Instanzen protokolliert.. Hierbei muß bemerkt werden, daß die Externalisierung der ConTract-Daten im Rahmen einer permanenten Suspendierung als ein Checkpoint implementiert wird.

stepinstances		
Attribut	Datentyp	Erklärung
checkpoint_nr	long	Nummer des Checkpoints
stepinst_id_dbstr	char[60]	Global eindeutige Kennung der Stepinstanz
gtrid_dbstr	char[60]	Transaktions-Id
cid_dbstr	char[60]	ConTract
step_id	long	Zugeordneter Step (quasi die *Klasse* dieser Instanz)
version	long	Versionsnummer der Stepinstanz
par_index	long	Index für parallele Instanzen einer Version
finished_code	long	Returncode der step_finished Meldung

2. **Die Relation für die dynamische Step-Daten**

   In dieser Relation wird der Zustand der dynamischer Step-Daten protokolliert.

step_dynamic		
Attribut	Datentyp	Erklärung
checkpoint_nr	long	Nummer des Checkpoints
cid_dbstr	char[60]	ConTract
step_id	long	Step
version_cnt	long	Zähler für Versionsnummern der Instanzen dieses Steps
nof_done_inst	long	Anzahl ausgeführter Instanzen
nof_run_inst	long	Anzahl laufender Instanzen

3. **Die Relation für die Event-Daten**

Diese Relation protokolliert den Zustand der Ereignisse.

events		
Attribut	Datentyp	Erklärung
checkpoint_nr	long	Nummer des Checkpoints
cid_dbstr	char[60]	ConTract
event_id	long	Event
rec_token	long	Bereits erhaltene Token

4. **Die Relation für den ConTract-Zustand**

Diese Relation nimmt den internen Zustand eines ConTracts auf.

contract_dynamic		
Attribut	Datentyp	Erklärung
checkpoint_nr	long	Nummer des Checkpoints
cid_dbstr	char[60]	ConTract
contract_state	long	Globaler Ausführungszustand des ConTracts
desired_action	long	Gewünschte Folge von Zustandsänderungen
start_time	long	Startzeit (Beginn der Ausführung)
termination_time	long	Terminierungszeit (Ende der Ausführung)
nof_run_stepinst	long	Anzahl momentan aktiver Steps
cont_name	char[40]	Name des ConTracts