Integration von Verteilungskonzepten in ein adaptives Workflow-Management-System

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von:

Jochen Zeitler

1. Gutachter: Prof. Dr. Peter Dadam
2. Gutachter: Prof. Dr. Michael Weber

1999

Vorwort

Diese Diplomarbeit entstand aus der Idee, die umfangreichen Konzepte des ADEPT-Modells zur flexiblen Ausführung von Workflows mit Forschungsarbeiten zu verteilten WfMS zu verbinden. Zur Demonstration der erarbeiteten Verfahren wurden diese in den Prototyp des ADEPT-WfMS integriert.

Das ADEPT-Modell entstand 1995 in der Abteilung Datenbanken und Informationssysteme (DBIS) der Universität Ulm auf der Grundlage von Ergebnissen aus Forschungskooperationen im Bereich klinischer Informationssysteme. Seitdem wurde das Modell kontinuierlich weiterentwickelt und 1998 erstmals prototypisch ein ADEPT-WfMS realisiert.

Besonders danken möchte ich Thomas Bauer, Clemens Hensinger und Manfred Reichert für die ausgezeichnete Betreuung der Diplomarbeit. Ihre Bereitschaft zu intensiven Diskussionen und der kritischen Durchsicht der Arbeit machten das Gelingen der Arbeit erst möglich. Dies gilt auch für die fruchtbare Zusammenarbeit mit Herrn Hensinger bei der Erweiterung des ADEPT-WfMS. Weiterer Dank gebührt Prof. Dr. Peter Dadam und Prof. Dr. Michael Weber für die fachliche Unterstützung dieser Arbeit sowie allen anderen Mitarbeitern der Abteilung Datenbanken und Informationssysteme für das hervorragende Arbeitsklima.

Ulm, im Dezember 1999 Jochen Zeitler

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In dieser Arbeit werden verschiedene Konzepte und Realisierungsaspekte für die verteilte Steuerung von Workflows im ADEPT-WfMS behandelt.

1.1 Unterstützung von unternehmensweiten Prozessen durch WfMS

Kostensenkung und Effizienzsteigerung sind heute mehr denn je aktuelle Themen in Unternehmen. Im Zuge der Rationalisierung von Abläufen bewährt sich seit Jahren die Geschäftsprozeßmodellierung und -optimierung [JBS97]. Hierbei werden bestehende Abläufe analysiert, modelliert und verbessert. Die Wirkung dieser Maßnahmen entfaltet sich erst dann voll, wenn auch die Durchführung der Abläufe ausreichend durch EDV-Systeme unterstützt wird.

Für diese Aufgabe wurden *Workflow-Management-Systeme* (WfMS) entwickelt. Sie integrieren vorhandene Datenbanken, Betriebssysteme und Anwendungssoftware und schaffen so eine gemeinsame Basis für die Verarbeitung betrieblicher Abläufe.

Bereits erhältliche WfMS bieten eine relativ starre Prozeßunterstützung an. Abweichungen vom vordefinierten Ablauf (englisch: Workflow) sind zur Laufzeit nicht oder nur mit erheblich eingeschränkter Unterstützung z.B. in Bezug auf Korrektheits- und Robustheitsaspekte durch das System möglich. In der Praxis ist ein hohes Maß an Flexibilität jedoch unbedingt erforderlich, da nicht alle möglichen Sonderfälle bei der Modellierung berücksichtigt werden können. Dies würde einerseits zu einer kaum mehr überschaubaren Anzahl komplexer Ausführungszweige führen und andererseits daran scheitern, daß nicht immer alle denkbaren Abweichungen im voraus bekannt sind.

Eine ebenso wichtige Forderung an WfMSe ist die nach Skalierbarkeit ([BD97], [BD98a], [BD99a] und [BD99b]). Prozeßorientierte WfMSe müssen die Ausführung von großen unternehmensweiten oder unternehmensübergreifenden Workflows mit einigen Tausend Benutzern und einer noch größeren Zahl an gleichzeitig aktiven WF-Instanzen erlauben (Production Workflows). Kommerzielle Systeme bieten hierfür keine oder nur eine sehr eingeschränkte Unterstützung. Es existieren zwar einige Forschungsprototypen verteilter WfMSe ([AMG95], [BMR94], [CGP+96], [MWW+98], [SNS99] u.a.). Keiner dieser Ansätze unterstützt jedoch die dynamische Adaption von Workflows zur Laufzeit.

Ziel dieser Arbeit ist es daher, in einem WfMS Adaptierbarkeit und Skalierbarkeit zu vereinen. Wie wir später sehen werden, konkurrieren beide Forderungen miteinander, etwa im Hinblick auf die Performanz des Systems oder den Synchronisationsaufwand bei verteilter WF-Steuerung. Hier gilt es, einen vernünftigen Kompromiß zu finden. Bisher gibt es noch kein WfMS, das diesen beiden Forderungen gleichermaßen genügt.

1.2 Einordnung der Diplomarbeit und Aufgabenstellung

Die Diplomarbeit baut auf dem ADEPT-Modell, einem in der Abteilung Datenbanken und Informationssysteme der Universität Ulm entwickelten und an den Standard der Workflow Management Coalition (WfMC) [WMC99] angelehnten prozeßorientierten adaptiven WF-Modell auf [RD98]. ADEPT bietet einen strukturierten Ansatz zur Modellierung von Workflows und zeichnet sich durch vielfältige Möglichkeiten zur dynamischen Adaption von Workflows aus. Im Zentrum des $ADEPT_{flex}$ Ansatzes steht eine vollständige und minimale Menge von Änderungsoperationen, mit der sich laufende Instanzen unter Erhalt ihrer Korrektheit und Konsistenz modifizieren lassen. Eine wichtige Erweiterung bietet $ADEPT_{distribution}$ [BD97], ein Modell zur verteilten Steuerung von Workflows. Ziel des $ADEPT_{distribution}$-Ansatzes ist es, eine in Bezug auf die Gesamt-Kommunikationskosten des Systems möglichst günstige, verteilte Ausführung großer Workflows zu erreichen.

In dieser Arbeit sollen Konzepte und Realisierungsaspekte beider Modelle - $ADEPT_{flex}$ und $ADEPT_{distribution}$ - zu einem Ganzen verbunden werden. Wichtigstes Ziel ist wieder die effiziente, flexible und robuste Ausführung komplexer Workflows. Dabei wird zunächst die verteilte Ausführung von Workflows ohne Berücksichtigung dynamischer WF-Änderungen betrachtet. Später werden dann Aspekte dynamischer Änderungen bei verteilter WF-Steuerung näher behandelt. Für beide Fälle werden Anforderungen und Probleme identifiziert sowie unterschiedliche Lösungsansätze diskutiert. Geeignete Varianten werden herausgearbeitet und im Detail behandelt. Ein wichtiges Anliegen bei der

Entwicklung von Lösungskonzepten bilden Optimierungen mit dem Ziel der Reduzierung des Kommunikationsaufkommens.

Eine Auswahl der konzeptuell entwickelten Verfahren wird schließlich in den ADEPT-Prototyp integriert, so daß dieser die verteilte Ausführung von Workflows einschließlich dynamischer Änderungen ermöglicht.

1.3 Gliederung der Arbeit

Die vorliegende Arbeit besitzt drei Schwerpunkte, die sich auf mehrere Kapitel verteilen: Die verteilte Ausführung von ADEPT-Workflows, in Verbindung mit dynamischen Änderungen neu dazu kommende Probleme und die Integration der entwickelten Konzepte in den ADEPT-Prototyp.

Zunächst wird in Kapitel 2 eine Einführung gegeben, in der Grundlagen zu WfMS, zum ADEPT-Ansatz und zur verteilten Ausführung von Workflows behandelt werden. Dabei werden auch elementare Algorithmen definiert, die für die weiteren Verfahren benötigt werden. Kapitel 2 ist deshalb wichtig für das grundlegende Verständnis der Arbeit.

Die verteilte Ausführung von ADEPT-Workflows und die damit verbundenen Herausforderungen, wie die effiziente Durchführung von Migrationen[1], sind Thema von Kapitel 3 und 4. Zunächst werden lediglich statische Serverzuordnungen zugelassen, in Kapitel 4 wird die Betrachtung auf variable Serverzuordnungen (vgl. [BD99a], [BD00]) ausgeweitet. In jedem Kapitel werden zunächst grundsätzliche Anforderungen erläutert und anschließend Verfahren, einschließlich verschiedener Optimierungen, entwickelt und diskutiert.

In Kapitel 5 wird intensiv auf die besonderen Herausforderungen der verteilten Ausführung in Verbindung mit dynamischen Änderungen eingegangen. Es werden Protokolle entwickelt, die auf den in Kapitel 3 und 4 entwickelten Konzepten aufbauen und die für die Erweiterung eines adaptiven WfMS um Möglichkeiten zur verteilten WF-Ausführung genutzt werden können.

Ausgewählte Konzepte der Kapitel 3 bis 5 wurden in den ADEPT-Prototyp integriert. Hierauf wird in Kapitel 6 im Detail eingegangen. Die Implementierung erfolgte in Java. Über die konzipierten Java-Klassen und deren Zusammenarbeit sowie die Bedienung des Systems gibt der Anhang Auskunft.

In Kapitel 7 werden die entwickelten Konzepte gegen andere verteilten WfMS abgegrenzt und es wird eine kurze Zusammenfassung und ein Ausblick auf mögliche Erweiterungen gegeben.

[1] Hiermit ist die Übertragung von zur Ausführung benötigten WF-Informationen und die anschließende Übergabe der Kontrolle der Ausführung eines Workflows an einen anderen Server gemeint.

2 Grundlagen

In diesem einführenden Kapitel werden allgemeine Anforderungen an WfMS aufgezeigt und zum Verständnis der Arbeit erforderliche Konzepte des ADEPT-Modells erläutert. Schließlich wird auf die Besonderheiten der verteilten Ausführung von Workflows eingegangen.

2.1 Anforderungen an WfMS

Mit einem WfMS sollen Unternehmensabläufe unterstützt und weitgehend automatisiert werden. Das System muß Kenntnis von den Abläufen und den zu verarbeitenden Daten haben, um eine vorgegebene Reihenfolge der Tätigkeiten und die Erfüllung aller Voraussetzungen zu deren Durchführung zu gewährleisten. Dies bedeutet insbesondere, daß alle benötigten Daten vorhanden sein müssen. Entsprechend den Gegebenheiten der Ablauforganisation dürfen bestimmte Aufgaben nur von den dazu berechtigen Mitarbeitern in den passenden Abteilungen des Unternehmens bearbeitet werden. Manche Aufgaben können auch automatisch von in dem System eingebundenen Programmen erledigt werden.

Mit unterschiedlichen, nicht zusammenhängenden Programmen können zwar die Tätigkeiten zum Erreichen des Unternehmenszieles durchgeführt werden. Hier ist jedoch keine Reihenfolge vorgesehen, so daß sich jeder Beteiligte selbst darum kümmern muß, alle für ihn bestimmten Tätigkeiten richtig durchzuführen und mit den anderen Mitarbeitern zu kommunizieren. Im Falle von Fehlern oder Abweichungen von gewohnten Abläufen gibt es keine automatische Unterstützung. Veränderungen zum Beispiel in der Ablaufstruktur müssen an allen Komponenten separat durchgeführt und allen Beschäftigten mitgeteilt werden. Damit existieren viele Teilsysteme, so daß das Zusammenspiel schwer zu durchschauen und zu warten ist. Alle diese Faktoren stellen eine enorme Fehlerquelle dar und beeinflussen die Produktivität extrem negativ.

Es soll jedoch kein großes monolithisches Anwendungsprogramm erstellt werden, daß alle festgelegten Anforderungen erfüllt. Das Programm wäre sehr komplex und damit nur schwer und unter hohen Kosten zu warten. Kleine Änderungen in der Ablaufstruktur beispielsweise durch im Lauf der Zeit identifizierte Verbesserungsmöglichkeiten machen jedesmal eine Neucompilierung des gesamten Programms nötig. Dies stellt nicht nur einen immensen Aufwand dar, sondern ist auch extrem fehlerträchtig. Aufgrund der Umstellung wird der gewohnte Ablauf für einige Zeit beeinträchtigt, was noch höhere Kosten als für die Wartung des Programms zur Folge haben kann.

Statt dessen bieten WfMSe einen Rahmen für die Ausführung von durch Vorlagen modellierten Abläufen. Die Idee eines prozeßorientierten WfMS ist es, die Ablauflogik, d.h. Bearbeitungsreihenfolgen, Informationsflüsse und organisatorische Zuordnungen, vom Anwendungscode zu trennen. Damit ist dem System die Information über zulässige Abläufe bekannt, so daß diese gesteuert und überwacht werden können. Potentiellen Bearbeitern von Schritten werden diese aktiv präsentiert. Durch Einträge in den individuellen Arbeitslisten bekommt jeder Beteiligte seine Aufgaben automatisch zugeteilt. Nachdem eine Aktivität zur Bearbeitung ausgewählt wurde, wird diese von allen Arbeitslisten entfernt, damit sie nicht mehrfach ausgeführt wird. Die zur Durchführung benötigten Eingabedaten werden bereitgestellt, alle Ausgabedaten werden dem WfMS bekannt gegeben.

Die Grundlage für die Ausführung eines Arbeitsablaufes ist ein zuvor definierter Kontrollfluß. Mit dessen Hilfe wird zur Ausführungszeit die Reihenfolge der Aktivierung der Arbeitsschritte festgelegt. In der Regel wird der Ablauf mit Kontrollkonstrukten für beispielsweise sequentielle, parallele und bedingte Ausführung von Schritten oder Schleifen modelliert. Ein gutes Modell bietet eine hohe Modellierungsmächtigkeit bei gleichzeitiger Möglichkeit zum Verifizieren der Korrektheit. Dies bedeutet insbesondere die Erreichbarkeit aller Aktivitäten und die Verklemmungsfreiheit. Anhand einer Ablaufvorlage können beliebig viele Instanzen erzeugt werden, die vom Startknoten bis hin zum Endknoten ausgeführt werden.

Die zweite wichtige Voraussetzung ist die Modellierung des Informationsflusses. Hierbei muß gewährleistet werden, daß alle Aktivitäten mit den benötigten Daten versorgt werden. Die Modellierung kann mittels Ein- und Ausgabeparametern erfolgen, so daß potentielle Lese- und

Schreibzugriffe auf ein bestimmtes Datum schon bei der Modellierung erkennbar sind. So können auch Zugriffskonflikte wie paralleles Schreiben erkannt werden. Alle in einem Geschäftsprozeß anfallenden Daten können nach [WMC99] in drei Kategorien eingeteilt werden: *Anwendungsdaten*, *WF-relevante Anwendungsdaten* und *WF-Kontrolldaten*.

Anwendungsdaten werden von in den Workflow eingebundenen Anwendungsprogrammen erzeugt und haben keinen Einfluß auf den Ablauf des Workflow. Sie müssen nicht zwingend für das WfMS bekannt sein, allerdings können dann alle Vorteile, die sich aus der Verwaltung durch das WfMS ergeben, nicht genutzt werden. Dies betrifft die Möglichkeit zur Überprüfung der Korrektheit des Datenflusses und der Berücksichtigung der Daten bei Migrationen, welche bei der verteilten Ausführung eines WF notwendig sind. Aus diesen Gründen wird in den folgenden Kapiteln angenommen, daß alle mit dem WF zusammenhängenden Daten dem WfMS bekannt sind.

WF-relevante Anwendungsdaten müssen dem WfMS zugänglich sein, da sie das Fortschreiten des Workflows beeinflussen. Sie dienen u.a. zu Verzweigungsentscheidungen.

Zu den WF-Kontrolldaten gehören zum Beispiel die Ablaufvorlage, die Statusinformation und die Historie. Sie werden nicht explizit durch Ausgabeparameter von Aktivitäten des WF, sondern nur bei der Modellierung bzw. automatisch bei der Ausführung geschrieben.

Das System bietet nicht nur Unterstützung für den gewünschten Ablauf und den Informationsfluß, sondern auch für viele weitere im Zusammenhang notwendige Funktionalität. So ermöglicht es die Abbildung der Unternehmensstruktur mit allen Beschäftigten, damit diese Information zur Laufzeit verwendet werden kann, um die Schritte den richtigen Bearbeitern zuzuordnen. Dabei werden häufig Rollen (z.B. „Arzt“) und Organisationseinheiten (z.B. „Chirurgie“) als Kriterium verwendet. Organisatorische Veränderungen können unabhängig von den Abläufen vorgenommen werden. In den meisten Fällen, beispielsweise bei Neueinstellungen, bleiben alle vorhandenen Abläufe weiterhin ausführbar. Flexible Systeme bieten zusätzlich auch Stellvertreterregelungen, falls ein Beschäftigter zeitweise nicht verfügbar ist.

Wünschenswert für ein WfMS ist daneben ein integriertes Ressourcenmanagement. So kann sichergestellt werden, daß nur solche Aktivitäten angeboten werden, über dessen Ressourcen der Bearbeiter verfügt. Dies können Arbeitsmittel wie bestimmte Geräte sein, dies kann aber auch beispielsweise ein Raum sein, der frei sein muß. Damit können auch konkurrierende Zugriffe erkannt und Konflikte vermieden werden.

Im Falle von Fehlern bei der Ausführung kann vom System Unterstützung geboten werden, zumindest jedoch können diese protokolliert und eine zuständige Stelle benachrichtigt werden. Grundsätzlich besteht auch die Möglichkeit, bereits durchgeführte Schritte bis zu einer bestimmten Stelle im Workflow zurückzusetzen und alle bis dahin nach außen sichtbar gemachten Änderungen zurückzunehmen. Falls dies nicht möglich sein sollte, muß eine Aktivität ausgeführt werden, die das ursprüngliche Ergebnis der zurückzusetzenden Aktivität kompensiert. Vor allem bei realen Handlungen wird dies nötig. So muß z.B. „Verband abnehmen“ durchgeführt werden, wenn „Verband anlegen“ rückgängig gemacht werden soll.

Weitere wichtige Anforderungen an WfMSe sind:

- Skalierbarkeit:

  Ein WfMS muß auch für die Ausführung von sehr langen Workflows, die sich durch das ganze Unternehmen hindurch ziehen, geeignet sein. Da hierbei alle Beschäftigten des Unternehmens in den Workflow einbezogen werden können, muß die Verteilung der Ausführung auf mehrere Server möglich sein. Anderenfalls kann bei größeren Unternehmen mit einer hohen Zahl von Benutzern und gleichzeitig aktiven WF-Instanzen leicht eine Überlastung von WF-Servern und Kommunikationssystem auftreten. Dies gilt erst recht für unternehmensübergreifende Workflows, bei denen die Kommunikationswege noch kritischer sind und die in Teilabschnitten nur von einem WfMS im jeweiligen Unternehmen kontrolliert werden dürfen.

- Performanz:

  Bei Produktions-WfMSen müssen alle Bearbeiter so rechtzeitig mit ausführbaren Aktivitäten versorgt werden können, daß bei ihnen keine Leerlaufzeiten entstehen. Dies muß auch zu Zeitpunkten mit Spitzenlast ohne spürbare Verzögerungen möglich sein.

- Fehlertoleranz:

  Ein während der Ausführung einer Aktivität auftretender Fehler darf die Ausführung anderer Aktivitäten nicht beeinträchtigen. Der Bearbeiter, bei dem der Fehler aufgetreten ist, muß mit der Ausführung anderer ihm angebotener Aktivitäten fortfahren können.

- Ausfallsicherheit/Verfügbarkeit:

  Um eine hohe Verfügbarkeit zu gewährleisten, können Standardkonzepte wie Cluster [CDK95] verwendet werden. WF-Instanzen sollten nur so kurz wie möglich gesperrt werden.

- Datensicherheit/Dauerhaftigkeit:

  Wie bei einem DBMS müssen die Daten persistent gesichert werden.

- Datenschutz:

  Nur die berechtigten Benutzer dürfen Zugriff zu den Daten erhalten, wenn sie Arbeitsschritte ausführen, welche die Daten lesen.

- Erlernbarkeit, einfache Bedienung [CGP+96]:

  Die Ausführung von Arbeitsschritten darf keine neuen Anforderungen an die Benutzer stellen. Ein Benutzer benötigt eine übersichtliche, seinem Einsatzgebiet angemessene Oberfläche. Das System muß bei nicht selbsterklärenden Interaktionen Unterstützung bieten.

  Genauso müssen die Modellierungswerkzeuge anschaulich und leicht bedienbar sein. Die reale Ablaufstruktur muß erkennbar sein.

2.2 Begriffe

Die in dieser Arbeit verwendeten Begriffe entsprechen weitgehend den von der Workflow Management Coalition (WfMC) vorgeschlagenen [WMC99]. In der WfMC haben sich vor allem bedeutende Hersteller von WfMSen zusammengeschlossen, um die WF-Technologie voranzubringen. Neben einer einheitlichen Terminologie wurde ein Referenz-Workflow-Modell entwickelt, das mit standardisierten Schnittstellen bessere Zusammenarbeit unterschiedlicher WfMSe und leichtere Einbindung von Anwendungen ermöglichen soll.

Mit Geschäftsprozeß oder Ablauf werden die Arbeitsvorgänge in einem Unternehmen bezeichnet, die zur Erreichung des Geschäftszieles dienen. Sie sind aus einer geordneten Menge von Aktivitäten zusammengesetzt und im Kontext der Organisationsstruktur des Unternehmens zu betrachten.

Ein automatisierter Geschäftsprozeß, der elektronisch verarbeitet wird, heißt Workflow. Er wird mit einem WfMS modelliert und ausgeführt. Das Ergebnis der Modellierung wird Prozeßdefinition, Prozeßvorlage oder Prozeßschema genannt. Oft wird auch äquivalent der Begriff Ablaufvorlage bzw. Ablaufschema oder nur kurz Vorlage oder Template verwendet. Zur Ausführung muß aus der Vorlage eine Prozeßinstanz (Instanz, Prozeß, Workflow) erzeugt werden. Sie besteht aus zusammenhängenden Aktivitäten, synonym auch mit Knoten oder (Arbeits-)Schritt bezeichnet. Die Aktivitäten werden von Bearbeitern, auch Akteure genannt, ausgeführt.

2.3 Zum ADEPT-Modell

Das ADEPT-Modell wird seit 1995 in der Abteilung Datenbanken und Informationssysteme der Universität Ulm entwickelt. In die Entwicklung sind praktische Erfahrungen aus Projekten beispielsweise in Zusammenarbeit mit der Uniklinik Ulm eingeflossen. Ziel dabei war es, eine mächtige Beschreibungsmöglichkeit für betriebliche Abläufe zu bieten, die eine graphisch anschauliche Darstellung ermöglicht.

ADEPT ist ein prozeßorientiertes WfMS. Deshalb muß in jedem Workflow stets eine strikte Trennung zwischen Daten- und Kontrollfluß herrschen. Weitere zur Modellierung notwendige Konzepte wie die passende Abbildung von Organisationsstrukturen werden unterstützt. Durch den Austausch von Informationen über eine Datenbank wird die Integration in bestehende Systeme ermöglicht.

Besonderen Wert wurde auf die Möglichkeit zur automatischen Überprüfung von Korrektheitskriterien gelegt. Daher können mit ADEPT keine beliebigen Graphen erzeugt werden, statt dessen müssen die angebotenen Basiskonstrukte verwendet werden. Somit können vom System verschiedene Analyseverfahren angewendet werden, die bereits zur Modellierungszeit auf Fehler oder Unstimmigkeiten hinweisen und so letztlich für einen reibungslosen Ablauf der Instanzen sorgen.

2.3.1 ADEPT-Konstrukte

Die Konstrukte bauen auf einer Blockstruktur auf und sorgen so für Übersichtlichkeit und eine einfachere Wartung. Um alle in der Praxis auftretenden Abläufe modellieren zu werden können, werden aus imperativen Programmiersprachen bekannte Konzepte übernommen.

Ein Workflow besteht aus einer Anzahl nacheinander abzuarbeitender Schritte oder Blöcke, die durch Kontrollkanten miteinander verbunden sein müssen.

Eine einfache Sequenz ist in Abbildung 2-1 zu sehen. Jedes Rechteck stellt eine Aktivität dar. In der Abbildung treten nur Kontrollkanten vom Typ „CONTROL_E“ auf. Wenn im folgenden von Kontrollkanten gesprochen wird, ist immer dieser Kantentyp gemeint. Eine andere Interpretation ordnet diesem Begriff alle Kantentypen zu, die auf den Kontrollfluß direkten Einfluß haben.

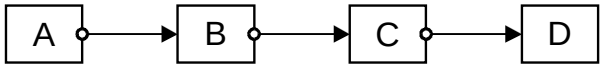


Abbildung 2-1: Sequenz in ADEPT.

Um gleichzeitiges Bearbeiten mehrerer Aktivitäten oder Entscheidungen zwischen Alternativen zu ermöglichen, wurden Verzweigungen eingeführt (vgl. Abbildung 2-2). Sie bestehen aus Verzweigungs- bzw. Splitknoten, mindestens zwei Zweigen und einem Synchronisations- bzw. Joinknoten. Für einen Verzweigungsblock gibt es drei verschiedene Möglichkeiten:

a) Bei der parallelen Verzweigungen werden alle Zweige gestartet und unabhängig voneinander ausgeführt. Aktivität A ist hier ein AND-Verzweigungsknoten, Aktivität C ein AND-Joinknoten.

b) Bei der bedingten Verzweigung wird nur ein Zweig ausgeführt. Welcher dies ist, wird über einen Wert, der beim OR-Verzweigungsknoten A zwingend vorhanden sein muß, und den ausgehenden Kanten zugeordnete Werte entschieden.

c) Bei der bedingten Verzweigung mit finaler Auswahl werden zunächst alle Zweige ausgeführt. Der Zweig, der zuerst beendet wird, wird quasi im Nachhinein als gewählte Alternative betrachtet. Die anderen Zweige werden verworfen, das System muß für einen Zustand sorgen, als ob sie nicht ausgeführt worden wären. Danach kann mit der Ausführung des OR-Synchronisationsknotens C fortgefahren werden. In der Arbeit soll diese Variante der Verzweigung allerdings ausgenommen werden.

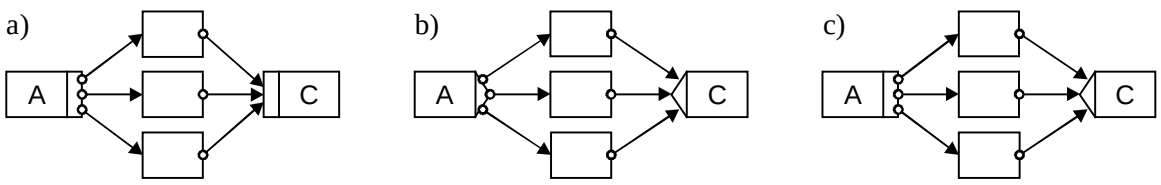


Abbildung 2-2: Verzweigungen in ADEPT.

Um mehrfache Wiederholungen adäquat darstellen zu können, gibt es ein Schleifenkonstrukt (vgl. Abbildung 2-3). Ein Schleifenblock besitzt immer einen Schleifenanfangs- und -endknoten, die über eine spezielle Schleifenkante vom Typ „LOOP_E“ miteinander verbunden sind. Der Schleifenendknoten muß über einen Wert verfügen, anhand dessen entschieden wird, ob eine erneute Schleifeniteration durchgeführt wird oder die Schleife über die vom Endknoten ausgehende Kontrollkante verlassen wird.

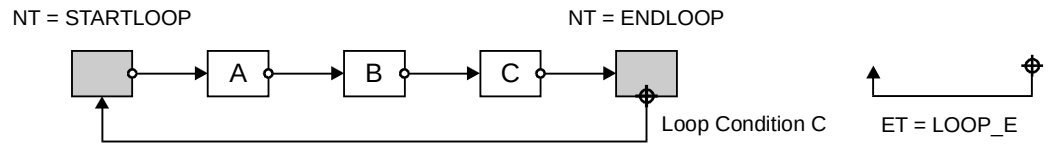


Abbildung 2-3: Schleife in ADEPT.

Neben dem Kontrollfluß wird in ADEPT auch der Datenfluß modelliert. Dazu gibt es im ADEPT-Modell Datenelemente, über die Daten ausgetauscht werden. Diese sind global für einen Prozeß gültig. Von den Datenelementen werden Versionen beliebiger Datenobjekte, d.h. sowohl einfacher Variablen als auch komplexer Strukturen, verwaltet.

Zugriffe auf ein Datenelement müssen durch dessen Verbindung mit Lese- oder Schreibkanten zu den betreffenden Aktivitäten explizit modelliert werden. Um eine korrekte Datenversorgung zu gewährleisten, muß vor einem Lesezugriff auf ein Datenelement immer ein Schreibzugriff stattgefunden haben. Die Eingabedaten einer WF-Instanz werden als vom Startknoten geschriebene Parameter der entsprechenden Datenelemente modelliert, Ausgabewerte erscheinen in dem Modell als Lesezugriffe des Endknotens.

Grundsätzlich gilt, daß nur Datenelementeinträge für eine lesende Aktivität sichtbar sind, deren Schreiber im Kontrollfluß vor dem Leser liegen. Hierbei sind auch Vorgänger über Schleifenkanten erlaubt, so daß in einer vorherigen Schleifeniteration geschriebene Werte zugreifbar sind. Somit werden immer die dem Kontext der Aktivität entsprechenden Datenelementeinträge gelesen („Datenkontextmanagement“). Dies müssen nicht immer die zeitlich gesehen zuletzt geschriebenen sein. Wenn Aktivität D aus Abbildung 2-4 das Datenelement d_1 liest, so erhält er immer den von A geschriebenen Wert, egal ob die ebenfalls schreibende Aktivität B schon vor dem Start von D beendet wurde. Daten, die in einem anderen als dem betrachteten parallelen Zweig geschrieben wurden, sind damit in der Regel dort nicht sichtbar.

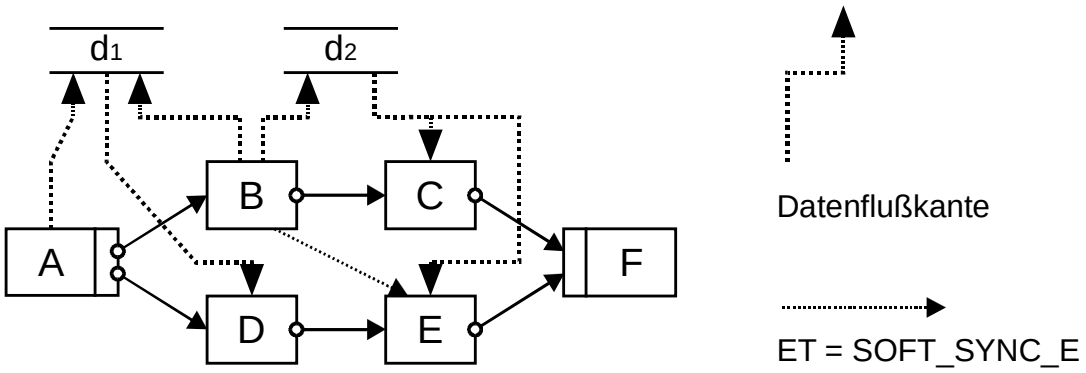


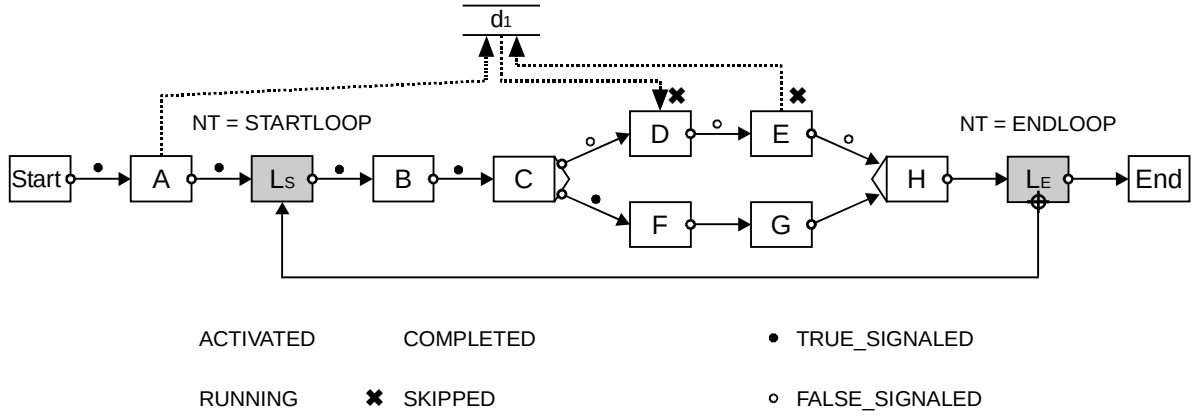
Abbildung 2-4: Modellierung des Datenfluß in ADEPT.

Allerdings sind Abhängigkeiten zwischen Aktivitäten der Zweige erlaubt. Sie werden über besondere Synchronisationskanten modelliert, abgekürzt auch nur Sync-Kanten genannt. Aktivität E darf lesend auf d_2 zugreifen, da B jetzt Vorgänger von E ist. Diese Kanten werden wie die in der Abbildung vorhandene Kante zwischen B und E vom Typ „SOFT_SYNC_E“ gestrichelt dargestellt.

In Verbindung mit parallelen Verzweigungen gibt es eine weitere Besonderheit. Es nicht erlaubt, daß ein Datenelement in unterschiedlichen Zweigen parallel beschrieben wird, da sonst der am Ende der Verzweigung gültige Wert zufällig von je nach Instanz unterschiedlichen Aktivitäten geschrieben worden sein konnte. Eine klare Semantik, die für alle Instanzen einer Vorlage zu gleichen Leser-Schreiber-Beziehungen führt, wäre nicht mehr vorhanden.

2.3.2 Ablauf eines Workflow

Bis jetzt wurden lediglich WF-Vorlagen betrachtet. Wird aus einer Vorlage eine Prozeßinstanz abgeleitet, kommt noch Information über den Ausführungszustand dazu. Dies sind Knoten- und Kantenmarkierungen, Werte für die Datenelemente und ein ausführliches Protokoll über den Ablauf, die Ausführungshistorie. In Abbildung 2-5 wird beispielhaft eine Prozeßinstanz dargestellt. Im Anschluß wird auf die darin vorkommende Symbolik und die wesentlichen Schritte im Ablauf eingegangen.



Ausführungshistorie

1	START(Start, 1), END(Start, 1)	8	START(H, 1), END(H, 1)
2	START(A, 1), END(A, 1)	9	START(L_E, 1), END(L_E, 1)
3	START(L_S, 1), END(L_S, 1)	10	START(L_S, 2), END(L_S, 2)
4	START(B, 1), END(B, 1)	11	START(B, 2), END(B, 2)
5	START(C, 1), END(C, 1)	12	START(C, 2), END(C, 2)
6	START(D, 1), END(D, 1)	13	START(F, 1)
7	START(E, 1), END(E, 1)		

Historie zum Datenelement d_1

1 A, 1, COMPLETED, „Hallo“
2 E, 1, COMPLETED, „Welt“

Abbildung 2-5: Zustand einer WF-Instanz in ADEPT.

Bei der Instanziierung eines Prozesses erhalten alle Knoten den Status NOT_ACTIVATED und alle Kanten den Zustand NOT_SIGNALED (jeweils ohne Symbol). Der Startknoten der Instanz wird mit COMPLETED markiert und „START“- und „END“-Eintrag in die Ausführungshistorie geschrieben. Immer wenn ein Knoten diese Markierung erhält, werden alle seine ausgehenden Kanten auf TRUE_SIGNALED gesetzt, es sei denn es handelt sich um den Splitknoten einer bedingten Verzweigung oder einen Schleifenendknoten. Im ersten Fall werden alle Knoten der nicht gewählten Zweige auf SKIPPED und alle zugehörigen Kanten auf FALSE_SIGNALED gesetzt. Nach Beendigung einer Schleife wird entweder die ausgehende Schleifen- oder die Kontrollkante mit TRUE_SIGNALED markiert.

Wenn alle eingehenden Kanten einer Aktivität den Zustand TRUE_SIGNALED oder FALSE_SIGNALED erhalten haben[2], wechselt der Status der Aktivität über in ACTIVATED und sie ist damit startbar. Wird sie von einem Benutzer ausgewählt und gestartet, geht ihr Zustand über in RUNNING, ihr Iterationszähler wird um 1 erhöht und in der Ausführungshistorie wird ein „START"-Eintrag geschrieben. Bei erfolgreicher Beendigung wird ein „END"-Eintrag in der Ausführungshistorie geschrieben, die Aktivität gelangt in den Zustand COMPLETED und so können wieder die Ausgangskanten mit TRUE_SIGNALED markiert werden. Die Ausführungsregeln finden sich im Detail in [Hen97].

Im Beispiel ist noch ein interessanter Fall für die Datenversorgung von Aktivitäten enthalten. Der Knoten D muß in seiner ersten Iteration den von A geschriebenen Wert „Hallo" lesen. Würde er allerdings in einer weiteren Schleifeniteration später erneut ausgeführt (dann mit der Iterationsnummer 2), so könnte er das von E geschriebene „Welt" lesen, obwohl E ja ein Nachfolger von D über Kontrollkanten ist. Allerdings ist E ein Vorgänger von D über die Schleifenkante und kann daher zur Datenversorgung von Aktivitäten in folgenden Schleifeniterationen dienen. Wichtig ist daher, die Berechnung der Vorgänger unter Berücksichtigung der Ausführungshistorie korrekt durchzuführen.

Im Kontext von ADEPT ist die Korrektheit und Konsistenz einer Ausführungshistorie durch die Reihenfolge der Einträge und deren Vollständigkeit bestimmt. Die Einträge zu Aktivitäten $a \in A$, die in der Graphstruktur in einer Reihenfolgebeziehung stehen, müssen in der Ausführungshistorie diese Reihenfolgebeziehung widerspiegeln (unter Berücksichtigung von Schleifen, „<" bedeutet „steht in der Ablaufhistorie vor"). Folgende Bedingungen lassen sich für die Reihenfolge formulieren (vgl. [Rei00]):

Definition 2-1: Korrektheit von Ausführungshistorie.

1. $\forall a \in A\ \forall i \in$ Integer gilt: START(a, i) < END(a, i)
2. $\forall a \in A\ \forall i, j \in$ Integer mit i<j gilt: START(a, i) < START(a, j)
3. Sei $a, b \in A$ und $i, j, k \in$ Integer mit $a \in pred^*(b)$ und $(L_S, L_E) = minLoop(a, b)$.
   Falls L_S = NULL oder $a \in \{L_S, L_E\}$ oder $b \in \{L_S, L_E\}$, dann gilt: END(a, i) < START(b, j).
   Falls $L_S \neq$ NULL und $END(L_S, k) < END(a, i) < START(L_E, k)$
   und $END(L_S, k) < START(b, j) < START(L_E, k)$, dann gilt: END(a, i) < START(b, j).

Zu jeder Aktivität müssen für jede Iteration, in der sie ausgeführt wurde, die entsprechenden Historieneinträge vorhanden sein.

2.3.3 Algorithmen

Im Umgang mit WF-Instanzen häufig benötigte Funktionen werden in den beiden untenstehenden Tabellen aufgelistet. Sie beziehen sich alle auf eine konkrete Instanz, so daß dieser Parameter nicht aufgeführt ist.

Weiterhin gilt einheitlich für alle Funktionen: Mengenwertige Funktionen geben, wenn kein Ergebnis gefunden werden konnte, die leere Menge ∅ zurück. Eine Funktion, die nur einen Wert bzw. ein Tupel zurückgibt, liefert in diesem Fall NULL bzw. ein Tupel mit NULL-Werten.

Historien werden immer als Kopie übergeben, so daß die Funktion daran Änderungen vornehmen darf, ohne daß sich Auswirkungen auf die Instanz ergeben.

Die Methoden zur Vorgänger- und Nachfolgerbestimmung sind in [Hen97] ausführlich beschrieben. Bisher noch nicht an anderer Stelle formulierte Funktionen werden im Anschluß an Tabelle 2-1 und Tabelle 2-2 algorithmisch in Pseudocode angegeben. Hiervon ausgenommen sind succ_part, pred_part

[2] Ausgenommen sind OR-Joinknoten und Schleifenanfangsknoten. Hier genügt es, wenn eine eingehende Kontroll- bzw. Schleifenkante und alle eingehenden Sync-Kanten signalisiert sind.

und succ_readers, welche nur von einer nicht gewählten Verfahrensalternative bzw. Optimierungen benötigt werden.

Ebenfalls nicht näher ausgeführt werden selbsterklärende Funktionen, die mit einer geeigneten Datenstruktur leicht berechnet werden können (z.B. alreadySent, readers, correspondingNode).

Name der Funktion	**Aufgabe**
Vorgänger- und Nachfolgerbestimmung	
pred(n)	Bestimmt direkte Vorgänger der Aktivität n. Berücksichtigt werden Kontroll- und Sync-Kanten.
pred*(n)	Bestimmt alle (transitiven) Vorgänger der Aktivität n über die Graphstruktur. Berücksichtigt werden Kontroll- und Synchronisationskanten.
succ*(n)	Wie pred*, nur Nachfolger statt Vorgänger.
l_pred*(n)	Wie pred*, jedoch werden jetzt zusätzlich Schleifenkanten berücksichtigt.
l_succ*(n)	Wie succ*, jedoch werden jetzt zusätzlich Schleifenkanten berücksichtigt.
c_pred*(n)	Wie pred*, jedoch werden jetzt ausschließlich Kontrollkanten berücksichtigt.
c_succ*(n)	Wie succ*, jedoch werden jetzt ausschließlich Kontrollkanten berücksichtigt.
pred_h(n, H)	Bestimmt alle Vorgänger der Aktivität n über die Graphstruktur und die Ausführungshistorie. Damit werden ggf. auch mehrere Iterationen einer Schleife berücksichtigt. Es werden auch solche Aktivitäten berücksichtigt, die über Kontroll- und Sync-Kanten keine Vorgänger sind, allerdings über die Schleifenkante einer zuvor durchlaufenen Schleife Vorgänger sind. Bei den zurückgelieferten Vorgängeraktivitäten wird ebenfalls die Iterationsnummer mitgeteilt. Die Einträge der Rückgabeliste sind genauso geordnet wie die korrespondierenden Einträge der Historie. Definiert im folgenden Algorithmus 2-1.
pred_i(n, i, H)	Wie pred_h(n, H), jedoch muß als Eingabeparameter die Iterationsnummer i der Aktivität n mit angegeben werden, damit nur die Einträge berücksichtigt werden, die vor dieser Iteration ausgeführt wurden. Definiert im folgenden Algorithmus 2-2.
succ_i(n, i, H)	Wie pred_i(n, i, H), nur statt Vorgängern Nachfolger. Dabei können natürlich nur die in der Historie vermerkten bereits abgeschlossenen Aktivitäten berücksichtigt werden. Definiert im folgenden Algorithmus 2-3.
succ_part(n)	Bestimmt alle Nachfolger über Kontroll-, Sync- und Schleifenkanten der gleichen Serverklasse[3] bis zu Aktivitäten einer anderen Serverklasse, d.h. der zurückgelieferte Teilgraph darf nicht durch Aktivitäten anderer Serverklassen unterbrochen sein. So wird die mit dem Knoten zusammenhängende Partition, allerdings nur in Kantenrichtung, bestimmt. Die potentiellen Migrationspunkte, welche die Partition begrenzen, sind durch die zur Modellierungszeit vorberechneten und bei dynamischen Änderungen aktualisierten Serverklassen gegeben.
pred_part(n)	Wie succ_part, nur Vorgänger statt Nachfolger.
Den Datenfluß betreffende Funktionen	
succ_readers(n, d)	Bestimmt über die Graphstruktur alle das Datenelement d lesenden Nachfolger von n einschließlich n, die potentiell die gleiche Version des in d enthaltenen Datenelementeintrags lesen.
lastWriter(n, d, H)	Bestimmt die Aktivität und deren Iterationsnummer, die ausgehend von Aktivität n als letzte das Datenelement d geschrieben hat. Arbeitet vom Prinzip wie pred_h(n, H), d.h. die Historie wird berücksichtigt. Ein modifiziertes ReadDataElement aus [Rei00], bei dem nicht der Wert selbst zurückgeliefert wird, sondern der Schreiber dieses Wertes und dessen Iteration, kann allein mit der Ausführungshistorie

[3] Die Funktion wird in Abschnitt 3.6 benötigt. Bis dahin werden Begriffe wie „Serverklasse“ erklärt.

Name der Funktion	Aufgabe
	nicht verwendet werden. Dieser Algorithmus geht nämlich von der aktuell vorhandenen Datenhistorie aus. Wenn eine Aktivität eingesetzt würde, die noch nicht ausgeführt ist und vor der ein Schreiber liegen würde, der ebenfalls noch nicht ausgeführt ist, wird mit ReadDataElement keine Fehlermeldung, sondern der letzte verfügbare Wert zurückgeliefert! Definiert im folgenden Algorithmus 2-4.
alreadySent(d, n, i, s)	Gibt an, ob der von Knoten n in Iteration i geschriebene Datenelementeintrag zu d schon an den Server s geschickt wurde.
readers(d)	Alle Aktivitäten, die auf das Datenelement d lesend zugreifen.
writers(d)	Alle Aktivitäten, die auf das Datenelement d schreibend zugreifen.
Weitere Funktionen	
corresponding SplitAct(N)	Bestimmt die gemeinsame AND-Splitaktivität von allen übergebenen Aktivitäten. Diese ist sozusagen das „kleinste gemeinsame Vielfache“. Bei verschachtelten Blöcken wird der von der Struktur her am weitesten innen liegende[4] gemeinsame Splitknoten zurückgegeben.
corresponding SplitActivity(n)	Bestimmt zu einem Joinknoten n den korrespondierenden Splitknoten.
corresponding JoinActivity(n)	Bestimmt zu einem Splitknoten n den korrespondierenden Joinknoten.
outermostJoinAct(n)	Falls sich die Aktivität n innerhalb einer parallelen Verzweigung befindet: Bestimmt zum übergebenen Knoten n den nächsten folgenden Joinknoten der Eingangssemantik ALL_Of_ALL auf der höchstmöglichen Ebene der Blockstruktur. Zwischen Knoten n und dem Resultatknoten darf kein paralleler Splitknoten der gleichen oder einer höheren Ebene wie der Resultatknoten liegen, da ein solcher Splitknoten bereits zu einem der Aktivität n über den Kontrollfluß nachfolgenden Verzweigungsblock gehört. Ansonsten, d.h. insbesondere natürlich wenn Aktivität n in der obersten Ebene der Blockstruktur liegt, wird NULL zurückgegeben.
correspondingStart Loop(n)	Gibt den passenden Schleifenanfangsknoten zum Schleifenendknoten n zurück. Dies ist der direkte Vorgänger von n über die Schleifenkante.
minLoop(n_1, n_2)	Gibt ein Tupel (L_S, L_E) bestehend aus dem Schleifenanfangs- und dem Schleifenendknoten der kleinsten Schleife zurück, so daß n_1 und n_2 im Schleifenkörper enthalten sind. Dabei ist „klein“ bei einer Schleife so definiert, daß eine Schleife kleiner als eine andere ist, wenn sie weniger Aktivitäten enthält.

Tabelle 2-1: In der Arbeit verwendete Funktionen im Umgang mit Workflows.

Name der Funktion	Aufgabe
correspondingNode(H_E)	Gibt zum Historieneintrag H_E den Knoten und dessen Iterationsnummer zurück.
correspondingHistory Entry(H, n, i)	Sucht in H den zum Knoten n in Iteration i entsprechenden Historieneintrag H_E und gibt diesen zurück, falls vorhanden.
readAndDeleteOldestHistory Entry(H)	Liest den ältesten Historieneintrag H_E von H und gibt ihn zurück. Dabei wird H_E aus H entfernt. Es wird der erste Eintrag aus der die (Teil)Historie H repräsentierenden geordneten Liste genommen. Dies muß nicht unbedingt der zeitlich gesehen älteste sein, falls die entsprechenden Knoten zu anderen Einträgen aus H von der Graphstruktur her parallel zu correspondingNode(H_E) sind.
readAndDeleteLatestHistory Entry(H)	Wie readAndDeleteOldestHistoryEntry(H), jedoch wird jetzt der aktuellste (=neueste) Eintrag zurückgegeben.
append(H, H_{Teil})	Hängt die Teilhistorie H_{Teil} an die Historie H an. Dies kann auch nur ein einzelner Historieneintrag sein.

[4] Am weitesten innen liegender Block = Block auf der niedrigsten Ebene. Die Hauptsequenz des Workflow stellt die höchste Ebene der Blockstruktur dar. Die Aktivitäten eines jeden Zweiges befinden sich auf einer tieferen Ebene als die zugehörigen Split- und Joinknoten.

Name der Funktion	Aufgabe
	H_{Teil} wird ans Ende der Liste, welche H repräsentiert, angehängt.
latestHistoryEntry(H)	Findet in H den aktuellsten Historieneintrag H_E und gibt diesen zurück.
latestHistoryEntry ForServer(H, S)	Findet in H den aktuellsten Historieneintrag H_E mit Server(H_E) = S und gibt diesen zurück. Dabei ist Server(H_E) der Server, auf dem der Knoten correspondingNode(H_E) ausgeführt wurde.
$H_2 = H_1 \setminus \{ (n, i) \}$	Die korrespondierenden Einträge aus H_1 zu der übergebenen Menge von Knoten werden nicht ins Ergebnis H_2 übernommen, alle anderen Einträge schon.
$H_2 = H_1 \cap \{ (n, i) \}$	In H_2 werden nur die korrespondierenden Einträge zu der übergebenen Menge von Knoten übernommen. Alle anderen Einträge aus H_1 werden für das Ergebnis nicht berücksichtigt.

Tabelle 2-2: In der Arbeit verwendete Funktionen und Operationen im Umgang mit der Historie von Workflows.

Der Algorithmus pred_h zur Bestimmung aller Vorgänger eines Knotens n über die Graphstruktur und die Ausführungshistorie ist angelehnt an die Formulierung von „ReadDataElement“ in [Rei00]. Bei diesem Algorithmus wird ausgehend von einem Eintrag der Datenhistorie geprüft, ob dieser für einen Leserknoten n_{read} in der aktuellen Iteration ein gültiger Wert ist. Bei pred_h wird jedoch keine Datenhistorie, sondern die Ausführungshistorie verwendet, um daraus nur die Einträge herauszufiltern, die garantiert vor dem Knoten n in der aktuellen Iteration stattgefunden haben.

Algorithmus pred_h

input
n∈N: der Knoten, dessen Vorgänger bestimmt werden sollen
// es werden die Vorgänger von n in der Iteration, die dem zu n korrespondierenden
// Historieneintrag aus H mit der höchsten Iterationsnummer entspricht, bestimmt
H: Kopie eines Teilabschnittes der Ausführungshistorie der Instanz

output
P: { n∈N, i∈Integer }: die berechneten Vorgänger, in der gleichen Ordnung wie die Historie

begin
P = ∅;

repeat
// den ältesten Eintrag aus H herausnehmen und in HE festhalten
HE = readAndDeleteOldestHistoryEntry(H);
{n_H, i_H} = correspondingNode(HE);
if ($n_H \in pred^*(n) \cup n$) **then**
// Vorgänger über Kontrollkanten, damit gehört der Eintrag zum Ergebnis
P = P ∪ {n_H, i_H};
else if ($n_H \in l\_pred^*(n) \cup n$) **then**
// Vorgänger über Schleifenkanten, weitere Prüfung ist notwendig
// Prüfen, ob der Eintrag in einem früheren Schleifendurchlauf erfolgte oder aber innerhalb
// eines parallelen Teilzweiges derselben Iteration (in diesem Fall darf er nicht in das
// Ergebnis aufgenommen werden).
if (∃ Eintrag in (H ∪ HE) für Beendigung eines Schleifenendknotens L_E und es gilt:
($L_E \in c\_succ^*(n_H)$ **and** $L_E \in c\_succ^*(n)$)) **then**
// L_E nach n_H und vor n ausgeführt, da nur Historie von n_H bis n betrachtet!
// Damit muß n_H in einem früheren Schleifendurchlauf als n stattgefunden haben.
P = P ∪ {n_H, i_H};
until H = ∅;

end

Algorithmus 2-1: Bestimmung der Vorgänger unter Berücksichtigung der Ausführungshistorie.

Der folgende Algorithmus entspricht dem schon beschriebenen pred_h, mit dem Unterschied, daß noch zusätzlich die Iterationsnummer des Knotens angegeben ist. Falls noch kein Historieneintrag zu (n, i) vorhanden ist, können wie bisher alle schon vorhandenen Historieneinträge überprüft werden. Sollte ein Eintrag gefunden werden, darf nur der ältere Teil der Historie bis einschließlich dem gefundenen Eintrag für den Aufruf von pred_h verwendet werden.

Algorithmus pred_i

input
n∈N: der Knoten, dessen Vorgänger bestimmt werden sollen
i∈Integer: die Iterationsnummer von n
H: Kopie der vorhandenen Ausführungshistorie der Instanz

output
P: { n∈N, i∈Integer }: die berechneten Vorgänger, in der gleichen Ordnung wie die Historie

begin
HE = correspondingHistoryEntry(H, n, i);
if (HE ≠ NULL) **then**
H = alle Historieneinträge aus H vom ältesten Eintrag bis einschließlich HE;
P = pred_h(n, H);

end

Algorithmus 2-2: Bestimmung der Vorgänger unter Berücksichtigung der Ausführungshistorie und der Iterationsnummer.

Mit succ_i werden die Nachfolger von Knoten n in Iteration i unter Berücksichtigung von Graphstruktur und Ausführungshistorie bestimmt. Dabei können natürlich nur die in der Historie vermerkten bereits abgeschlossenen Aktivitäten berücksichtigt werden.

Algorithmus succ_i

input
n∈N: der Knoten, dessen Nachfolger bestimmt werden sollen
i∈Integer: die Iterationsnummer von n
H: Kopie der vorhandenen Ausführungshistorie der Instanz

output
S: { n∈N, i∈Integer }: die berechneten Nachfolger, in der gleichen Ordnung wie die Historie

begin
HE = correspondingHistoryEntry(H, n, i);
if (HE ≠ NULL) **then**
H = alle Historieneinträge aus H von HE bis zum aktuellsten Eintrag (jeweils einschließlich);
S = ∅;
// den aktuellsten (neuesten) Eintrag aus H herausnehmen
{n_H, i_H} = correspondingNode(readAndDeleteLatestHistoryEntry(H));
if ($n_H \in succ^*(n) \cup n$) **then**
// Nachfolger über Kontrollkanten, damit gehört der Eintrag zum Ergebnis
$S = S \cup \{n_H, i_H\}$;
else if ($n_H \in l\_succ^*(n) \cup n$) **then**
// Nachfolger über Schleifenkanten, weitere Prüfung ist notwendig
// Prüfen, ob der Eintrag in einem späteren Schleifendurchlauf erfolgte oder aber innerhalb
// eines parallelen Teilzweiges derselben Iteration (in diesem Fall darf er nicht in das
// Ergebnis aufgenommen werden).
if (∃ Eintrag in H für Beendigung eines Schleifenendknotens L_E und es gilt:
($L_E \in c\_succ^*(n_H)$ **and** $L_E \in c\_succ^*(n)$)) **then**
// L_E nach n und vor n_H ausgeführt, da nur Historie von n bis n_H betrachtet!

```
            // Damit muß n_H in einem späteren Schleifendurchlauf als n stattgefunden haben.
            S = S ∪ {n_H, i_H};
    else return ∅;
end
```

Algorithmus 2-3: Bestimmung der Nachfolger unter Berücksichtigung der Ausführungshistorie und der Iterationsnummer.

Mit lastWriter wird die Aktivität und deren Iterationsnummer, die ausgehend von Aktivität n als letzte das Datenelement d geschrieben hat, bestimmt. Der Algorithmus arbeitet vom Prinzip her wie pred_h, d.h. die Ausführungshistorie wird berücksichtigt.

Algorithmus lastWriter(n, d, H)

input

$n \in N$: der Knoten, von dem ausgehend der letzte Schreiber bestimmt werden soll

$d \in D$: das Datenelement, für das der letzte Schreiber bestimmt werden soll

H: Kopie der vorhandenen Ausführungshistorie der Instanz

output

$n_{Write} \in N$: der Knoten, der das Datenelement d zuletzt geschrieben hat

$i_{Write} \in$ Integer: die Iterationsnummer von n_{Write}

begin

$P = pred\_h(n, H)$;

(n_{Write}, i_{Write}) = aktuellster Eintrag in P mit $n_{Write} \in writers(d)$;

end

Algorithmus 2-4: Bestimmung des letzten Schreibers eines Datenelements unter Berücksichtigung der Ausführungshistorie.

2.3.4 Erweiterungen des Basismodells

Zum Basismodell von ADEPT wurden auch Erweiterungen entwickelt. Eine grundlegende Erweiterung behandelt das Konzept der dynamischen Änderungen $ADEPT_{flex}$ [Hen97]. Damit werden zur Laufzeit auf Wunsch Abweichungen von der modellierten Vorlage ermöglicht, ohne dabei jedoch die Korrektheit des gesamten Ablaufs zu gefährden. Im Rahmen von Kapitel 5 wird eine Einführung dazu gegeben und anschließend Konzepte zum Zusammenspiel mit der verteilten Ausführung von Workflows erarbeitet.

Für Themen wie Synchronisation von dynamischen Änderungen existieren bereits Konzepte [Wei97]. Die darin gewonnen Erfahrungen gelten natürlich ebenfalls im verteilten Fall.

Es existiert ein Organisationsmodell wird. Dieses entspricht im wesentlichen dem Modell der WfMC. Das Vorhandensein eines Organisationsmodells ist Voraussetzung für die Ausführung, wenn die Arbeitsschritte nicht lediglich bestimmten Benutzern fest verdrahtet zugeordnet sein sollen.

Zeitliche Aspekte in der Ausführung von Workflows werden in [Gri97] ausführlich behandelt. Sie sollen in dieser Diplomarbeit jedoch unberücksichtigt bleiben, da es sich um eine zur Verteilung orthogonale Problemstellung handelt.

Zur Demonstration des ADEPT-Konzepts wurde von Praktikanten und Mitarbeitern der Abteilung DBIS der Universität Ulm ein Prototyp entwickelt. Er enthält alle zur Modellierung und Ausführung eines Workflows notwendigen Komponenten und kann daher als WfMS betrachtet werden. Seine Grundlage ist eine relationale Datenbank, die Software wurde in der Sprache Java erstellt.

Im Rahmen von Praktika an der Universität Ulm wurden die genannten Erweiterungen bereits zu großen Teilen in den ADEPT-Prototyp integriert, so daß für die Entwicklung der Verteilungsschicht

darauf aufgebaut werden kann. Details zur innerhalb dieser Diplomarbeit durchgeführten Implementierung der Verteilungsschicht werden in Kapitel 6 beschrieben.

2.4 Verteilte Ausführung eines Workflow

Ein besonders kritischer Punkt bei der Ausführung von sehr großen, unternehmensweiten Workflows ist das anfallende Kommunikationsaufkommen zwischen dem WF-Server und dessen Klienten. In einem zentralen System mit mehreren hundert Benutzern kann es leicht zu Engpässen kommen. Auch wenn mehrere Server in verschiedenen Teilnetzen zur Ausführung von sehr umfangreichen WF-Instanzen parallel eingesetzt werden, fällt in allen Teilnetzen noch eine hohe Kommunikationsmenge an. Der Grund dafür ist, daß die Benutzer über alle Teilnetze verteilt sind und so meist Arbeitsschritte auf einem Server eines anderen Teilnetzes ausführen.

Mit der verteilten Ausführung eines Workflows sollen die Kommunikationskosten im betrachteten Netzwerk und die Last eines WF-Servers verringert werden. Die Idee dabei ist, einen Workflow in Abschnitte zu zerschneiden, so daß Teilmengen der Aktivitäten bei der Ausführung von unterschiedlichen Servern kontrolliert werden. Die Verteilung soll die Kommunikation über Teilnetzgrenzen hinweg weitestgehend reduzieren. Im Idealfall befinden sich die möglichen Bearbeiter einer Aktivität im Teilnetz des Servers, der diese Aktivität kontrolliert. Abbildung 2-6 zeigt einen solchen Workflow.

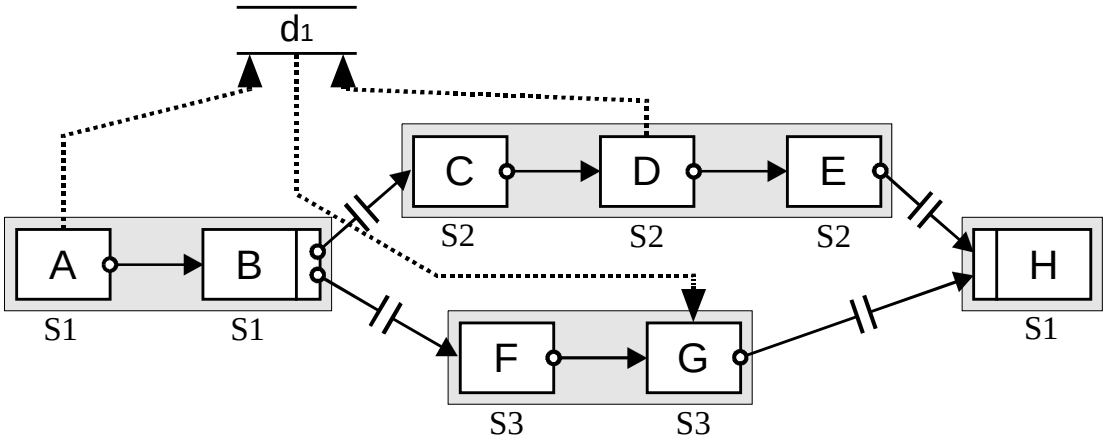


Abbildung 2-6: Auf mehrere Server verteilter Workflow mit Migrationen.

Eine über Kontroll-, Sync- oder Schleifenkanten zusammenhängende Teilmenge von Aktivitäten, die vom gleichen Server kontrolliert werden, wird dabei *Partition* genannt. Im Beispiel sind dies {A,B}, {C,D,E}, {F,G} und {H}, die jeweils mit einem grauen Kasten als zusammengehörig gekennzeichnet sind. Zwischen aufeinander folgenden Aktivitäten unterschiedlicher Partitionen finden *Migrationen* statt. Dabei muß der *Migrationszielserver* auf den aktuellen Stand gebracht und mit allen benötigten Daten versorgt werden, um mit der Ausführung des Workflow fortfahren zu können. Eine Migration findet immer entlang einer *Migrationskante* statt, in der Abbildung ist dies beispielsweise die unterbrochen gezeichnete Kontrollkante zwischen E und H. Die zugehörige Migration von S2 nach S1 wird mit $M_{E,H}$ bezeichnet. Als *Migrationspunkt* wird im folgenden die *Migrationsquellaktivität* verstanden.

Die gleichzeitig an der Ausführung der parallelen Zweige beteiligten Server S2 und S3 arbeiten unabhängig voneinander. Beide verfügen nur über einen Teil der Gesamtinformation. S2 kennt keine Ausführungsinformation der Aktivitäten F und G, S3 besitzt dementsprechend keine Information über den Status von C, D und E. Das Datenelement d_1 ist auf beiden Servern in unterschiedlichen Versionen vorhanden. Die auf S2 geschriebene Version ist zunächst nur dort bekannt. Bei den beiden Migrationen $M_{E,H}$ und $M_{G,H}$ zum Joinknoten H wird die Information der Teilzweige wieder zusammengeführt. Die dazu notwendigen Verfahren werden im einzelnen in Kapitel 3 beschrieben.

Zur Bestimmung der optimalen Verteilung der Aktivitäten auf die Server werden in [BD97], [BD98b] und [BD00] Verfahren vorgeschlagen. Ziel dabei ist es, mögliche Überlastungen von Systemkomponenten wie Server, Teilnetzen oder Gateways zu entdecken und zu verhindern. Ein Kostenmodell erlaubt es, geeignete Abschätzungen vorzunehmen. Mit Hilfe dieses Modells können zur Modellierungszeit unterschiedliche Varianten von Serverzuordnungen für einen Workflow verglichen und bewertet werden, so daß die Variante mit den geringsten Gesamtkosten als die beste übernommen werden kann.

Es sollen folgende Annahmen gelten:

- Basis für $ADEPT_{distribution}$, die verteilte Variante des ADEPT-Systems, ist das prozeßorientierte WfMS ADEPT. Dies bedeutet insbesondere, es existiert ein Organisationsmodell, in dem Personen sowohl mit Organisationseinheiten (OE) als auch Rollen assoziiert werden können. Die Auswahl möglicher Bearbeiter einer Aktivität erfolgt zur Laufzeit über ein Auswahlprädikat. Der tatsächliche Bearbeiter ergibt sich somit meist „zufällig“ aus den möglichen Bearbeitern einer Aktivität.
- Das WfMS besteht aus mehreren Teilnetzen (Domains) mit je einem WF-Server. Diese sind untereinander austauschbar, d.h. jeder Server kann die Ausführung aller existierenden Workflows übernehmen.

  Unter dieser Bedingung kann keine Lastverteilung zwischen Servern innerhalb des gleichen Teilnetzes betrachtet werden. Diese Fragestellung ist aber auch orthogonal zur angestrebten Minimierung der Kommunikationskosten.
- Jeder WF-Server kann alle angemeldeten Klienten jedes Teilnetzes bedienen. Ein Benutzer ist jedoch i.d.R. einem oder mehreren Teilnetzen fest zugeordnet.
- Die potentiellen Bearbeiter einer Aktivität befinden sich nicht unbedingt im selben Teilnetz.
- Zwischen der Modellierung und der Ausführung des Workflow werden keine bedeutenden Änderungen an der Organisationsstruktur wie auch der Netzwerktopologie vorgenommen.

Für die Berechnung einer günstigen Verteilung müssen zumindest folgende Kosten berücksichtigt werden:

- Kosten für den Parametertransfer beim Starten und Beenden von Aktivitäten.

  Am Server fallen nur Kosten an, wenn die Aktivität auch von diesem kontrolliert wird. Im Teilnetz hingegen entstehen auch dann Kosten, wenn ein Klient eine Aktivität auf einem anderen Server startet oder beendet. In diesem Fall wird auch das Gateway belastet mit der Übertragung von Ein- und Ausgabeparametern belastet.
- Kosten für das Aktualisieren von Arbeitslisten.

  Für alle angemeldeten Benutzer müssen von Zeit zu Zeit durch Löschen oder Hinzufügen von neuen Einträgen die Arbeitslisten aktualisiert werden. Immer wenn ein Bearbeiter eine Aktivität selektiert und damit für sich übernimmt oder eine Aktivität neu aktiviert wird, sind die Arbeitslisten aller potentiellen Bearbeiter der Aktivität betroffen. Die Änderung soll sofort mitgeteilt werden.

  Hierfür gibt es allerdings verschiedene Verfahren, die zu unterschiedlichen Kosten führen [BD98a]. Es sollen nur zwei Varianten genannt werden: Polling vs. aktives Übertragen und Übertragung der kompletten Liste oder nur von Differenzinformation. Wie beim Parametertransfer sind i.d.R. mehrere Teilnetze betroffen.
- Kosten für die Kommunikation von Aktivitäten mit externen Datenquellen.

  Immer wenn die Datenquelle nicht im gleichen Teilnetz wie die Aktivität liegt, fällt Kommunikation über Teilnetzgrenzen an. Der WF-Server selbst wird nicht belastet.
- Migrationskosten.

Wenn beim Übergang von zwei Aktivitäten ein anderer Server die Kontrolle übernimmt, müssen zwischen den WF-Servern Daten ausgetauscht werden.

Bei der Berechnung der Gesamtkosten muß noch die Ausführungshäufigkeit der Aktivitäten bzw. die Häufigkeit für eine Migration berücksichtigt werden, bevor aus den Kostenformeln eine Summe gebildet werden kann. Wenn alle gleichzeitig vorhandenen Instanzen aller WF-Typen berücksichtigt werden, kann für Server, Teilnetze oder Gateways die Belastung errechnet werden. So können mögliche Überlasten entdeckt werden.

Um zu einer optimalen Verteilung zu kommen, müssen zunächst die Serverzuordnungen so bestimmt werden, daß die drei zuerst genannten Kostenarten minimiert werden. Jetzt sind allerdings sehr viele Migrationen notwendig. Daher sollte für jede Migration überprüft werden, ob die zu erwartenden Kosten ohne die Migration niedriger wären. Das genaue Verfahren wird in [BD98b] und [BD00] beschrieben.

Wenn die Verbindungsqualität zwischen den Servern unterschiedlich ist, so kann dies über Gewichte für die Gateways berücksichtigt werden. Sie werden in einer Kommunikationskostenmatrix festgehalten. Die Matrix wird darüber hinaus zur Bestimmung der günstigsten Verbindung benötigt, wenn bei Migrationen Daten von mehreren unterschiedlichen Quellen bezogen werden können. Dazu wird die folgende Funktion guenstigsterServer definiert:

guenstigsterServer: $s \in S \times MS \in P(S) \rightarrow s_{opt} \in S$
Bestimme aus der Kommunikationskostenmatrix den bezogen auf s günstigsten Server aus allen Servern der Menge $MS \subseteq S$ ohne s. Bei Kostengleichheit muß ein zusätzliches Kriterium herangezogen werden oder nach dem Zufall entschieden werden.

Bei den in den folgenden Kapiteln verwendeten Algorithmen sollen Verfahren Anwendung finden, die die Erfordernisse von verteilten Systemen angemessen berücksichtigen. Zentrale Komponenten sollen vermieden werden, damit keine Engpässe, sog. bottlenecks, entstehen können. Statt dessen soll in Situationen, an denen Informationen an einer Stelle zusammengetragen werden müssen, immer ein zur Aufgabe passender Koordinator bestimmt werden.

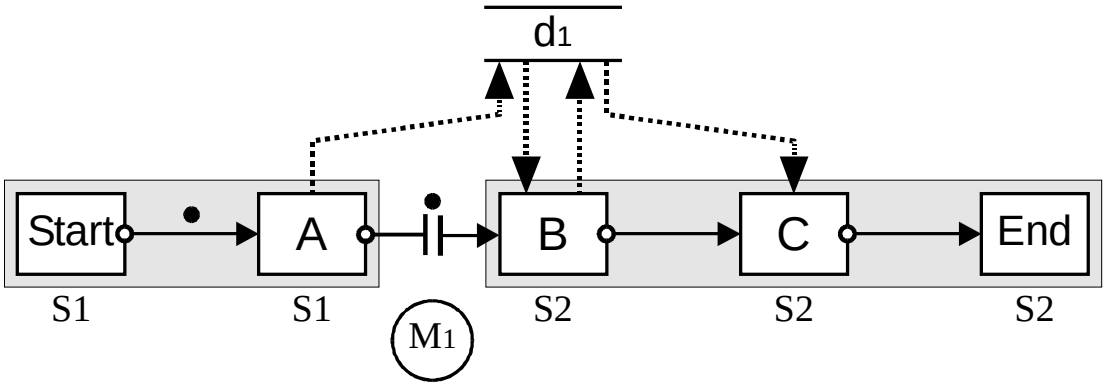
3 Migrationen bei statischen Serverzuordnungen

Im folgenden Kapitel wird die verteilte Ausführung von Workflows untersucht, die abschnittsweise von verschiedenen Servern kontrolliert werden. Den Aktivitäten der Partitionen wurden zur Modellierungszeit Server zugeordnet. Es werden Verfahren entwickelt, um die bei der Ausführung angefallenen Daten weiterzugeben, wenn die Kontrolle an einen anderen Server übergeht.

3.1 Statische Serverzuordnungen

Bei Verwendung *statischer Serverzuordnungen* wird für jede Aktivität der Prozeßvorlage festgelegt, durch welchem Server sie kontrolliert werden soll. Jede aus dieser Vorlage abgeleitete Instanz übernimmt diese Zuordnungen. Der für eine bestimmte Aktivität festgelegte Server hat zum Zeitpunkt der Ausführung dieser Aktivität die Kontrolle über einen Ausschnitt des Workflows. Im Falle einer parallelen Verzweigung können auch mehrere Server gleichzeitig an der Ausführung der Instanz beteiligt sein.

Für die Identifikation von Servern muß eine systemweit eindeutige Kennzeichnung, wie z.B. seine IP-Adresse, benutzt werden. Das folgende Beispiel zeigt einen sequentiellen Workflow mit statisch zugeordneten Servern.



Ausführungshistorie
1 START(Start, 1, S1), END(Start, 1, S1)
2 START(A, 1, S1), END(A, 1, S1)

Historie zum Datenelement d_1
1 A, 1, COMPLETED, „Hallo“

Abbildung 3-1: Sequenz mit statisch zugeordneten Aktivitäten und einer Migration.

Im folgenden werden aus Gründen der besseren Lesbarkeit WF-Server nicht durch Angabe ihrer IP-Adresse identifiziert, sondern durch symbolische Namen wie „S1“, „S_{Source}“ oder „S_{Target}“. Neu gegenüber dem Modell ohne Migrationen ist, daß die Server, auf denen eine Aktivität ausgeführt wurde, jetzt auch in der Ausführungshistorie vermerkt werden. Ein im folgenden verwendeter verkürzter Historieneintrag ist demnach wie folgt aufgebaut (vgl. Abbildung 3-1):

AKTION(Aktivität, Iterationsnummer, Server-ID), wobei AKTION $\in$ {START, END}.

An der Historie zu den Datenelementen muß nichts gegenüber dem bisherigen Verfahren, wie es in Kapitel 2 beschrieben ist, verändert werden.

Migrationen finden beim Übergang zwischen zwei Aktivitäten statt, d.h. beim Signalisieren der sie verbindenden Kante, wenn die Nachfolgeaktivität einem anderen Server zugeordnet ist als die gerade beendete Aktivität. Sie sind numeriert und werden graphisch durch die Verwendung von unterbrochenen Kantensymbolen hervorgehoben. Eine solche Kontrollkante verbindet in Abbildung 3-1 die Knoten A und B. Im dargestellten Beispiel ist „M_1“ die einzige Migration. Sie wird auch entsprechend der Quell- und Zielaktivitäten mit $M_{A,B}$ bezeichnet. Die unterschiedlichen Partitionen -

im Beispiel sind dies die beiden von S1 bzw. S2 kontrollierten WF-Abschnitte - werden durch Hinterlegen der enthaltenen Aktivitäten mit einer gemeinsamen grauen Fläche gekennzeichnet.

Manche der verwendeten Beispiele mögen etwas konstruiert wirken. Nichtsdestotrotz dienen sie der Illustration wichtiger Sachverhalte und sollen weniger Ausschnitte aus einem realen Workflow widerspiegeln. Falls z.B. in einer Abbildung Migrationen scheinbar nur für eine Aktivität durchgeführt werden, ist zu bedenken, daß diese Aktivität ja auch für einen ganzen Block von Aktivitäten mit identischer Serverzuordnung stehen könnte. Der Übersichtlichkeit halber wird die Darstellung auf das Wesentliche reduziert. Für die Erklärung überflüssige Aktivitäten und Datenelemente werden ausgeblendet.

Die diskutierten Verfahren basieren auf folgenden Annahmen:

1. Nur statische Serverzuordnungen sind erlaubt.
2. Es gibt keine dynamischen Änderungen.

3.2 Fragestellungen

In diesem Kapitel wird untersucht, welche zusätzlichen Aufgaben ein verteiltes WfMS mit statischen Serverzuordnungen und Migrationsunterstützung gegenüber einem zentralen WfMS leisten muß. Ein wesentliches Ziel einer solchen verteilten Ausführung ist, das Kommunikationsaufkommen im Gesamtsystem möglichst gering zu halten (vgl. [BD00]).

Zunächst wird untersucht, welche Informationen bei einer Migration übertragen werden müssen. Dabei wird davon ausgegangen, daß die benötigten Prozeßvorlagen aufgrund der Verwendung eines Replikationsmechanismus bereits auf den WF-Servern vorhanden sind. Es verbleiben noch die WF-Kontrolldaten und die WF-Daten, d.h. WF-relevante Daten und im WfMS verwaltete Anwendungsdaten, die bei einer Migration zu den Zielservern transportiert werden müssen[5]. Beide Datenkategorien werden in den folgenden Diskussionen getrennt behandelt. Für den Transfer dieser Daten zwischen Servern werden generelle Alternativen diskutiert und im Anschluß eine geeignete Variante ausgewählt und im Detail vorgestellt (incl. Algorithmen).

Anhand von Beispielen werden die Verfahren und ihre Algorithmen veranschaulicht. Darüber hinausgehende Verbesserungen, die nur in wenigen Spezialfällen interessant sind, werden lediglich angesprochen. In einem kurzen Resumee werden die Verfahren schließlich noch einmal tabellarisch miteinander verglichen.

3.3 Transfer von WF-Kontrolldaten

In diesem Abschnitt behandeln wir Aspekte der Übertragung von Kontrolldaten (WF-Status, Ausführungshistorie) bei der Durchführung einer Migration.

3.3.1 Generelle Alternativen

Zur Übertragung von WF-Kontrolldaten gibt es unterschiedliche Möglichkeiten, von denen im folgenden die wichtigsten diskutiert werden. WF-relevante Daten und Anwendungsdaten werden an dieser Stelle noch ausgeklammert, sie werden in den Abschnitten 3.5 und 3.6 ausführlich behandelt.

1. Bei der Minimallösung werden dem Migrationszielserver nur die ID der Quell- und der Zielaktivität der Migration und notwendige Kontextinformationen (WF-Instanz-ID) übertragen. Anhand dieser Information weiß der Zielserver dann, an welcher Stelle im Ablaufgraphen sich die Ausführung gerade befindet. Besitzt die Zielaktivität lediglich eine Eingangskontrollkante, so kann der Zielserver nach Abschluß dieser Migration mit der Ausführung fortfahren. Da es Knoten mit mehr als einer Eingangskante gibt (OR- bzw. AND-Joinknoten, Knoten mit eingehenden Synchronisationskanten), muß die Quellaktivität der Migration mit angegeben werden, so daß bekannt ist, über welche Eingangskante gerade migriert wurde. Damit können die für den lokalen Fall definierten Markierungsregeln übernommen und nach dem Markieren aller Eingangskanten

[5] An dieser Stelle werden dynamische WF-Änderungen (siehe Kapitel 5) noch ausgeklammert.

mit der WF-Kontrolle fortgefahren werden. Würde die Information über die Quellaktivität fehlen, müßte ein Kantenzähler eingeführt werden und Veränderungen an den Algorithmen vorgenommen werden, um die in den Markierungsregeln für Aktivitäten definierte Semantik beizubehalten.

Ein inhärentes Problem dieser einfachen Lösung ist das Fehlen von Zusatzinformation, z.B. über die Bearbeiter und Start- und Endezeitpunkte von Aktivitäten. Darüber hinaus ist der Zustand von WF-Aktivitäten nicht mehr von jedem Server aus feststellbar. Damit existieren auf vielen Servern Aktivitäten, von denen nicht bekannt ist, in welchem Zustand sie abgeschlossen wurden. Dies kann später bei der Sicherstellung der Datenversorgung zu Problemen führen (siehe Abschnitt 3.6). Auch einfache Statusanfragen wären nicht ohne weiteres möglich.

2. Besser wäre es daher, die Statusmarkierungen von Aktivitäten und Kanten der WF-Instanz zu übertragen und die Datenbank des Zielservers damit zu aktualisieren. Allerdings müssen dann bei Joinknoten die unterschiedlichen Informationen aus den Teilzweigen richtig zusammengeführt werden. Dies läßt sich bewerkstelligen, indem alle erhaltenen Informationen zu Knoten im Zustand „NOT_ACTIVATED“ und Kanten im Zustand „NOT_SIGNALED“ ignoriert werden. Damit werden nur die für die weitere WF-Ausführung erforderlichen Information in die Datenbank aufgenommen, und es werden keine bereits erhaltenen Werte überschrieben. Wenn allerdings die Möglichkeit des partiellen Zurücksetzen der WF-Kontrolle mit einbezogen werden soll, wird eine aufwendigere Lösung notwendig, da dann auch bereits abgeschlossene Knoten wieder in den Zustand „NOT_ACTIVATED“ versetzt werden müssen.

   Wie bei dem in 1. beschriebenen Minimalverfahren fehlen auch hier wichtige Zusatzinformationen. Die Markierung des Graphen und der Status der Aktivitäten lassen sich nach einer Migration auf dem Zielserver vollständig rekonstruieren. Ohne die Ausführungshistorie kann jedoch bei Schleifen mit enthaltenen bedingten Verzweigungen die Versorgung mancher Aktivitäten mit den korrekten Daten nicht garantiert werden. Wurden in einer bedingten Verzweigung enthaltene Schreiberaktivitäten in der vorigen Schleifeniteration nicht ausgeführt, in vorigen Iterationen jedoch schon, kann die tatsächliche Reihenfolge der Schreibzugriffe auf das entsprechende Datenelement nicht mehr festgestellt werden. Es ist nur die Information über den Ablauf der letzten Iteration bekannt. Wenn nicht mehr festgestellt werden kann, welche Aktivität ein Datenelement zuletzt beschrieben hat, ist auch nicht mehr bekannt, von welchem Server ein benötigtes Datenelement zu beziehen ist (siehe Abschnitt 3.6.3).

3. Eine korrekte, wenn auch nicht gerade effiziente Möglichkeit wäre es, alle Informationen aus der Datenbank zu übertragen, die zu der zu migrierenden WF-Instanz gehören. Darin ist dann auch die komplette Ausführungshistorie enthalten. Damit ist die Vollständigkeit garantiert, allerdings wird möglicherweise eine große Datenmenge kommuniziert. Auch bei diesem auf den ersten Blick naiven Verfahren entsteht ein Zusatzaufwand, um an Synchronisationspunkten einen korrekten Abglich aller eintreffenden Daten durchzuführen.

4. Um die in 1.-3. genannten Unzulänglichkeiten zu beheben, könnte man die Information zu schon beendeten Aktivitäten (mit den entsprechenden Kantenmarkierungen) und zusätzlich die Ausführungshistorie übertragen. Die Datenmenge ist zwar größer als im Fall von 2., aber dafür ist die Vollständigkeit der Daten garantiert. An Joinknoten muß ebenfalls ein Verfahren zum korrekten Mischen der erhaltenen Information, auch für die Ausführungshistorien, durchgeführt werden. Eine Schwäche dieses Ansatzes ist allerdings noch, daß Kontrolldaten redundant übertragen werden.

5. Wenn man die Strukturierung von WF-Modellen in ADEPT und die definierten Ausführungs- und Markierungsregeln konsequent ausnutzt, ergibt sich, daß nicht notwendigerweise immer die komplette Instanzinformation übertragen werden muß. So ist es niemals notwendig, die Markierungen des Graphen, d.h. dessen aktuellen Zustand, zu übertragen, da sich diese Information immer aus der Ausführungshistorie rekonstruieren läßt. Darüber hinaus enthält die Ausführungshistorie schon alle im folgenden wichtigen Informationen wie z.B. Start- und Endzeitpunkte und den tatsächlichen Bearbeiter einer Aktivität. Die bei Schleifenkonstrukten wichtige Iterationsnummer, die angibt, wie oft jede Aktivität ausgeführt wurde, ist ebenfalls im Historieneintrag vorhanden. Diese die Anwendungssemantik optimal ausnutzende Lösung wird im folgenden näher beschrieben.

3.3.2 Übertragung von Ausführungshistorien

Angenommen, jeder WF-Server schickt bei einer Migration die komplette bisher bei ihm vorhandene Ausführungshistorie an den Zielserver. Bei einem sequentiellen Workflow ergeben sich keine Probleme. Der Zielserver kann seine bisherige Ausführungshistorie verwerfen, und die Markierungen seines Graphen gemäß der neu erhaltenen Ausführungshistorie setzen.

Ein Problem tritt immer dann auf, wenn sich parallele Zweige unterschiedlicher Server an einem Punkt synchronisieren, da für diesen Fall zwei unterschiedliche Versionen der Ausführungshistorie aufeinander treffen. Das Zusammenführen der Information kann bei einem AND-Joinknoten erforderlich werden, aber auch schon innerhalb einer Verzweigung bei Knoten mit einmündenden Synchronisationskanten. Im folgenden Beispiel sind zwei dieser Stellen enthalten, und zwar die Migration $M_{B,E}$ über die Sync-Kante zwischen Aktivität B und E und die Migrationen $M_{C,F}$ bzw. $M_{E,F}$ vor dem Joinknoten F, bei dem die beiden parallelen Zweige des Graphen von Abbildung 3-2 zusammengeführt werden.

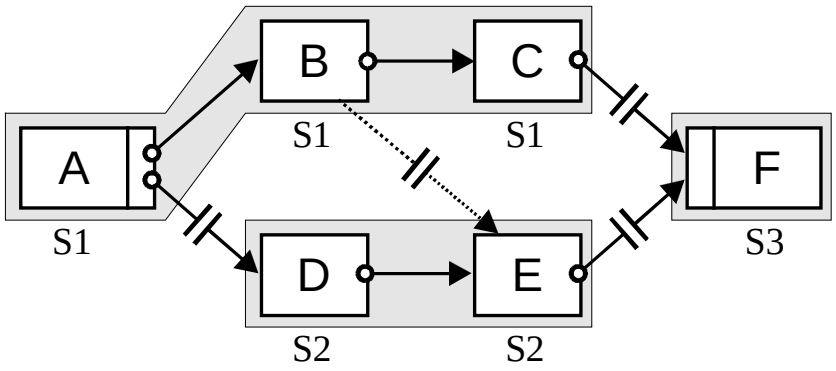


Abbildung 3-2: Synchronisieren der Ausführungshistorien nach Migrationen.

Beim *Zusammenfügen von Historieninformation* verschiedener WF-Server muß gewährleistet werden, daß das Ergebnis wieder eine *korrekte Historie* (siehe Kapitel 2) darstellt.

Definition 3-1: Korrektheit von Ausführungshistorie bei verteilter Ausführung.

Eine Historie ist dann korrekt, wenn sie auch auf einem zentralen System mit nur einem einzigen WF-Server entstanden sein könnte[6].

Mit einer verwandten Aufgabenstellung beschäftigt sich das Projekt TransCoop [WK96] in Bezug auf Concurrency Controlling beim verteilten Editieren komplexer Dokumente: Zwei oder mehr Personen arbeiten - zumindest zeitweise - am selben Dokument. Dabei entstehen dann mehrere gültige Änderungshistorien. Um daraus wieder ein gemeinsames Dokument zu machen, ist ein Verfahren zum Mischen notwendig, so daß wieder eine gültige Änderungshistorie entsteht. Es werden formale Bedingungen für eine korrekte Änderungshistorie aufgestellt und schließlich ein Algorithmus zum Import von Teilhistorien beschrieben. Die Autoren betonen, daß es wichtig ist, die anwendungsbezogene Semantik auszunutzen.

Zurück zu obigem Beispiel (Abbildung 3-2): In einer korrekten Ausführungshistorie müssen die Einträge zu B stets vor den Einträgen zur Aktivität E auftreten. Der Graphausschnitt soll nicht in einer Schleife enthalten sein, anderenfalls müßte die Betrachtung der Korrektheit halber auf jeweils eine Schleifeniteration beschränkt werden.

[6] Der Unterschied zu diesem Fall besteht darin, daß dann bei jeder Operation der gleiche Server vermerkt wäre.

Zunächst betrachten wir die Migration $M_{B,E}$, die beim Signalisieren der Synchronisationskante von B nach E erfolgt. Da die Ausführung von E blockiert wird, bis Aktivität B beendet wird, muß der Server von E vor der Ausführung von E auch schon die Historie mit dem Eintrag zu B erhalten und in die eigene Historie aufgenommen haben. Die Ausführungsreihenfolge zwischen B und D spielt keine Rolle, da beide Knoten in einer parallelen Verzweigung liegen und über die Graphstruktur in keiner Reihenfolgebeziehung zueinander stehen. Da bei Ausführung einer Aktivität die Protokolleinträge immer ans Ende der Historie angehängt werden, kann der Eintrag zu E somit nie vor dem Eintrag zu B stehen. Dies läßt sich auch leicht auf ein System mit mehreren Aktivitäten und beteiligten Servern übertragen. Es kann immer vorausgesetzt werden, daß die Reihenfolge der Historieneinträge zu den entsprechenden Aktivitäten vor dem Verzweigungsknoten korrekt ist. Alle Server, die innerhalb der Verzweigung liegende Aktivitäten kontrollieren, übernehmen diesen gemeinsamen Teil der Historie unverändert.

Bei den Migrationen $M_{C,F}$ und $M_{E,F}$ muß mit einem passenden Verfahren zum Zusammenführen der Historien dafür gesorgt werden, daß auch in der neuen Historie die Reihenfolge der Einträge korrekt bleibt. Dies kann mit einem sehr einfachen Verfahren erreicht werden, welches die längste Sequenz an übereinstimmenden Historieneinträgen der zusammenzuführenden Historien unverändert läßt, für die folgenden Einträge aber keine Reihenfolge garantiert.

Stark vereinfacht kommen bei der Migration $M_{C,F}$ {A,B,C} und bei $M_{E,F}$ {A,B,D,E} oder {A,D,B,E} bei S3 an. Die längste übereinstimmende Sequenz ist somit {A,B}. Die Reihenfolge in der die Historieneinträge von C und E angeordnet werden spielt keine Rolle, da die Aktivitäten im dargestellten Graphen in keiner Ordnungsbeziehung stehen. Gültige Historien für den Server S3 wären demnach {A,B,C,D,E}, {A,B,D,E,C}, {A,B,D,C,E}, {A,D,B,E,C} oder {A,D,B,C,E}. Sollten mehr als zwei Historien integriert werden, kann beim Mischen der Historien nacheinander paarweise vorgegangen werden. Zu beachten ist nur, daß die in der Reihenfolge zu erhaltenden übereinstimmenden Sequenzen unterbrochen sein können, was jedoch der Korrektheit einer Historie nicht schadet.

Wir präsentieren nun einen einfachen Algorithmus für das Zusammenführen von Historienausschnitten an einem Zielserver S_{Target}. Historieneinträge, die auf dem Zielserver bereits vorhanden sind, sind dort bereits in der richtigen Reihenfolge angeordnet. Sie enthalten die oben genannte übereinstimmende Sequenz. Neue Einträge können einfach angehängt werden.

Algorithmus MergeHistories

input
H_{old}: die auf S_{Target} schon vorhandene Historie
H_{new}: die zu integrierende Historie

output
H_{Result}: ist jetzt die vollständige Historie
H_{new}': nur die für S_{Target} neuen Einträge

begin
$H_{Result} = H_{old}$;
$H_{new}' = \varnothing$;

repeat
HE = readAndDeleteOldestHistoryEntry(H_{new});
if not (H_{Result} contains HE) **then**
append(H_{Result}, HE);
append(H_{new}', HE);
until $H_{new} = \varnothing$;

end

Algorithmus 3-1: Zusammenführung von Ausführungshistorien.

Die Komplexität des Algorithmus beträgt $O(|H_{new}|\cdot|H_{old}|)$. Da er nicht innerhalb einer weiteren Schleife aufgerufen wird, ist dies jedoch vertretbar.

Eine geringere Komplexität durch eine Optimierung analog dem Verfahren zur Berechnung eines Sort-Merge-Join ist nicht möglich, da nicht alle Historieneinträge in H_{old} und H_{new} zwingend in der gleichen Reihenfolge vorkommen müssen. Einträge, deren korrespondierende Aktivitäten parallel zueinander sind, dürfen in H_{old} und H_{new} in unterschiedlicher Reihenfolge zueinander auftreten (vgl. Abbildung 3-3).

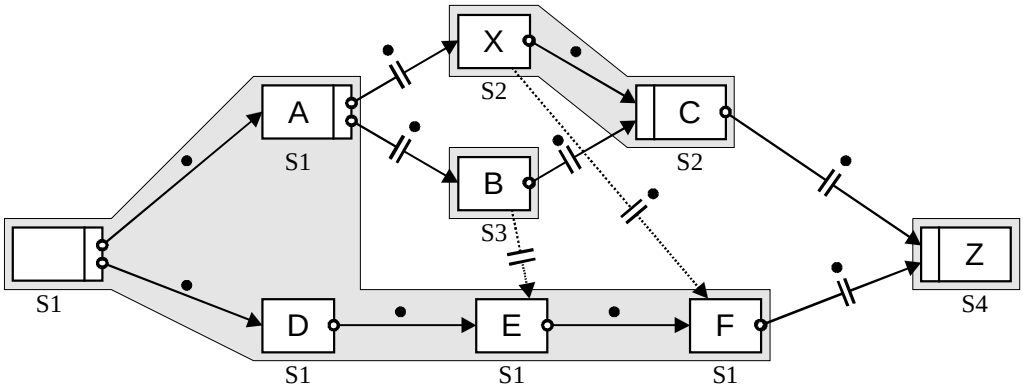


Abbildung 3-3: Optimierung analog Sort-Merge-Join ist wegen Parallelität nicht möglich.

Wenn in dem abgebildeten Workflow zunächst die Migration $M_{F,Z}$ stattfindet, wird auf S4 die Historie {A,D,B,E,X,F} angehängt. Bei der Migration $M_{C,Z}$ muß der Ausschnitt {A,X,B,C} von S4 in seine bereits vorhandene Historie eingearbeitet werden. Die Einträge zu X und B sind hier in der Reihenfolge vertauscht. In diesem zulässigen Szenario wird mit einem Verfahren analog dem Sort-Merge-Join der Eintrag zu B unzulässigerweise noch einmal angehängt. Es muß daher auf die naive Variante zum Einarbeiten der neuen Einträge zurückgegriffen werden, bei der jedesmal die komplette Liste verglichen werden muß.

Daß das Verfahren zum Mischen der Ausführungshistorien korrekt ist, soll nun gezeigt werden: Man nehme einen solchen neuen Historieneintrag N. Dieser kann nur Folgeaktivitäten oder parallelen Aktivitäten zu auf dem Zielserver schon bekannten Aktivitäten entsprechen. Würde N nämlich einer Aktivität entsprechen, die im Graph vor einer auf dem Zielserver bereits bekannten Aktivität liegt, wäre der Eintrag dort schon vorhanden, und dies wäre ein Widerspruch dazu, daß N neu ist.

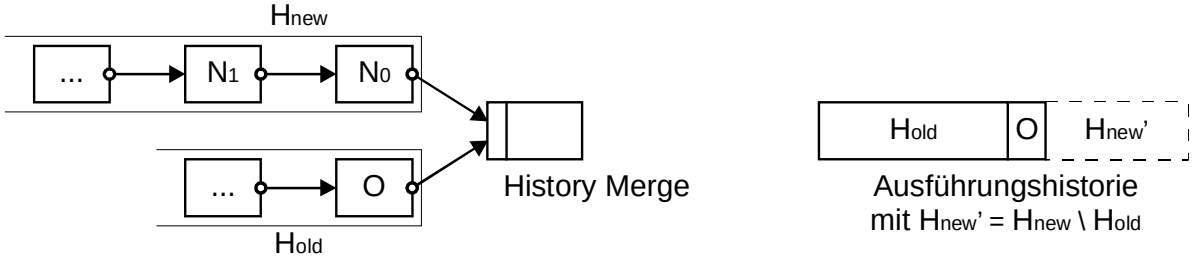


Abbildung 3-4: History Merge an Joinknoten durch Anhängen von neuen Einträgen.

Formal läßt sich der Beweis folgendermaßen ausdrücken (vgl. Abbildung 3-4):

$\forall$ Einträge $N \in Hn_{ew}$, $N \notin Ho_{ld}$ gilt: Aktivität(N) ist parallel zu O $\vee$ Aktivität(N) ist Nachfolger zu O, also $\neg$ (Aktivität(N) ist Vorgänger von O). Falls Aktivität(N) Vorgänger von O $\rightarrow$ $N \in Ho_{ld}$, weil H_{old} alle Einträge zu Vorgängern von O enthält. Dies ist ein Widerspruch zur Annahme, daß $N \notin H_{old}$.

Was bleibt noch zu tun, um auf dem Zielserver den gewünschten Zustand herzustellen? Um keine Inkonsistenz zwischen der Ausführungshistorie und dem WF-Instanzgraphen entstehen zu lassen, müssen die Knoten- und Kantenmarkierungen dieses Graphen noch gemäß der Historie aktualisiert werden. Dies erfolgt, indem für jeden Eintrag HE aus dem im Algorithmus 3-1 berechneten H_{new}' der Graph aktualisiert wird. Dazu müssen der Zustand des HE entsprechenden Knotens gemäß den durch ADEPT definierten Ausführungsregeln aktualisiert werden und dessen ausgehenden Kanten gemäß der Kanten-Markierungsregeln gekennzeichnet werden (siehe [Hen97]).

3.4 Optimierungen bezüglich der Übertragung von WF-Kontrolldaten

Da Migrationen bezüglich der Erzeugung von Kommunikationsaufkommen ohnehin relativ teure Operationen sind, müssen Lösungen gefunden werden, die Kommunikationsmenge zu minimieren. Mit dem vorangehend beschriebenen, einfachen Verfahren „Übertragen der kompletten auf dem Quellserver verfügbaren Ausführungshistorie“ werden die Migrationen mit Fortschreiten in der Prozeßinstanz kontinuierlich teurer, da die Historie sukzessive länger wird.

Ein intelligenteres Verfahren sollte folgende Eigenschaft eines verteilten WfMS ausnutzen: In den meisten Fällen wird ein Server nicht nur einmal an der Ausführung beteiligt. Daher verfügt er bei einer Migration häufig schon über einen Teil der Instanzinformation, nämlich über genau den Zustand aller Vorgängeraktivitäten von Aktivitäten (einschließlich dieser Aktivitäten), bei denen er zuletzt die Kontrolle über die Ausführung der Instanz hatte. Es ist also lediglich notwendig, ihn mit der *Differenzinformation*, d.h. das, was in der Zwischenzeit passiert ist, zu versorgen.

Dabei muß garantiert werden, daß auch alle zur Ausführung benötigten Informationen weitergegeben werden. Der Migrationszielserver muß die vollständige Information zu allen Vorgängern der Zielaktivität erhalten, einschließlich zu Vorgängern über Sync- und Schleifenkanten. Darüber hinaus soll Information vom Ausgangsserver, die für den Migrationszielserver an dieser Stelle nicht relevant ist, weggelassen werden[7].

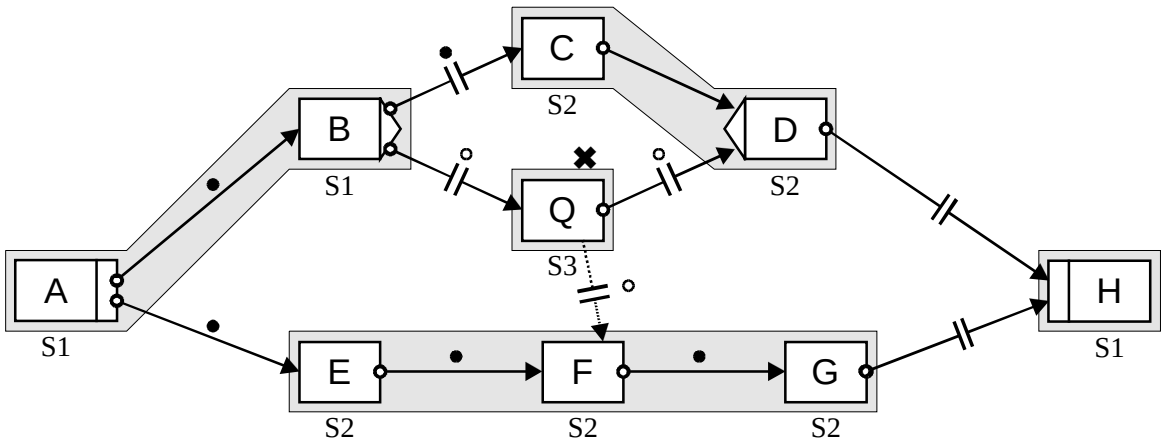


Abbildung 3-5: Übersprungene Aktivitäten in einer bedingten Verzweigung.

Speziell bei bedingten Verzweigungen ist die folgende Optimierung ratsam. Wird in der in Abbildung 3-5 enthaltenen bedingten Verzweigung der Teilzweig mit der Aktivität C ausgewählt, müssen die Aktivitäten des anderen Teilzweiges übersprungen werden. Beim Markieren der entsprechenden

[7] Noch nicht relevant sind Informationen aus parallelen Zweigen, welche nicht in der Vorgängermenge der Ausgangsaktivität liegen. Wenn der Migrationszielserver einen solchen parallelen Zweig kontrolliert oder kontrolliert hat, besitzt er bereits entsprechende Historieneinträge. Diese müssen vor der Übertragung entfernt werden, sie werden vom Zielserver erst später oder auch gar nicht benötigt.

Kanten mit FALSE_SIGNALED müßten, wenn analog dem Markieren mit TRUE_SIGNALED verfahren werden würde, die Migrationen $M_{B,Q}$, $M_{Q,D}$ und $M_{Q,F}$ durchgeführt werden. Dabei werden Informationen zum Server S3 übertragen, die dort gar nicht benötigt werden, da die Aktivität Q nicht ausgeführt wird. Die Versorgung der Aktivität F mit Informationen über vorangehende Aktivitäten ist jedoch notwendig.

Eine Lösung, die Kommunikation vermeidet, ersetzt die Migrationen über die Kanten des nicht gewählten Teilzweiges durch Migrationen vom Verzweigungsknoten B zu allen Aktivitäten, die eine aus einem nicht ausgewählten Teilzweig entspringende eingehende Sync-Kante besitzen. Im Beispiel ist dies die Migration $M_{B,F}$. Aktivität D erhält die benötigten Informationen über den ausgeführten Teilzweig und kann aktiviert werden, sobald die von C ausgehende Kontrollkante signalisiert wird.

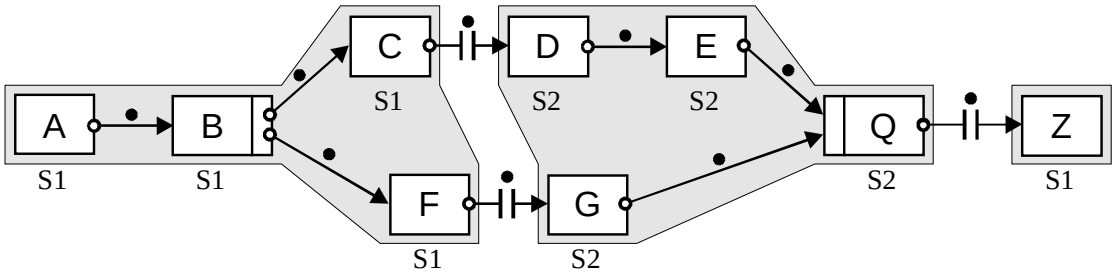
Allgemein läßt sich folgende Regel für Migrationen beim Signalisieren von Kanten angeben:

Definition 3-2: Migrationen beim Signalisieren von Kanten.

Migrationen finden beim Übergang zwischen zwei Aktivitäten statt, d.h. beim Signalisieren der sie verbindenden Kante mit TRUE_SIGNALED, wenn die Nachfolgeaktivität einem anderen Server zugeordnet ist als die gerade beendete Aktivität.
Wird unter derselben Bedingung eine Kante mit FALSE_SIGNALED markiert, so findet nur dann eine Migration statt, wenn es sich um eine Sync-Kante handelt. Die Migration geht von dem OR-Verzweigungsknoten aus, der die Markierung gesetzt hat.

3.4.1 Schicken-Verfahren

Ein im Rahmen dieser Arbeit entwickelter Algorithmus wird nun vorgestellt. Er berechnet für Migrationsquellserver S_{Source} jeweils eine reduzierte Ausführungshistorie, die dann an den Migrationszielserver S_{Target} bei der Migration verschickt wird. Idee dabei ist es, daß S_{Source} in seiner aktuellen Ausführungshistorie nach den Einträgen sucht, an denen der Server S_{Target} zuletzt an der Ausführung der Instanz beteiligt war. Diese Einträge stellen sozusagen die rechten Intervallgrenzen dar für die Information, die S_{Target} mit Sicherheit schon kennt. Falls Migrationen in einer parallelen Verzweigung stattgefunden haben, gibt es sogar mehrere dieser Intervallgrenzen. Das folgende Beispiel soll den Ablauf des Verfahrens verdeutlichen.



Ausführungshistorie bei S2 (exemplarisch)

1 START(A, 1, S1), END(A, 1, S1)
2 START(B, 1, S1), END(B, 1, S1)
3 START(C, 1, S1), END(C, 1, S1)
4 START(D, 1, S2)
5 START(F, 1, S1), END(F, 1, S1)
6 START(G, 1, S2)
7 END(D, 1, S2)
8 END(G, 1, S2)
9 START(E, 1, S2), END(E, 1, S2)
10 START(Q, 1, S2), END(Q, 1, S2)

Abbildung 3-6: Eine Workflowinstanz vor einer Migration.

Wir betrachten die Migration $M_{Q,Z}$ in Abbildung 3-6. Da der Server S1 bereits an der Ausführung dieser Instanz beteiligt war, kennt er einen Teil der Ausführungshistorie, nämlich genau die Historieneinträge zu A, B, C und F. S2 möchte keine unnötige Information verschicken, daher prüft dieser Server anhand der ihm bekannten Historie nach, welche Information S1 in jedem Fall vorliegt. Das sind die Einträge vor C und vor F (jeweils inklusive). Folglich muß S2 genau die Historieneinträge zu den Knoten C und F nachfolgenden Aktivitäten übermitteln.

Aufgrund der Tatsache, daß die Migrationen $M_{C,D}$ und $M_{F,G}$ innerhalb einer parallelen Verzweigung stattgefunden haben, ist es zulässig, daß in der bei S2 vorhandenen Historie nicht alle Einträge von Aktivitäten der ersten, mit A zusammenhängenden Partition, aufeinanderfolgen. S2 findet die Zeilen 4,6,7,8,9 und 10, d.h. die zu den Aktivitäten D, E, G und Q korrespondierenden Einträge, als die zu verschickende Information. Sie sind auch tatsächlich relevant, da sie alle in der Vorgängermenge von Q (einschließlich Q) liegen.

Der grobe Ablauf des Algorithmus für die Migration von S_{Source} nach S_{Target} stellt sich in dem gerade skizzierten Schicken-Verfahren folgendermaßen dar:

- S_{Source} berechnet die Menge M von relevanten Migrationspunkten, d.h. alle Aktivitäten bei denen S_{Target} zuletzt die Kontrolle der WF-Instanz hatte. Die Migrationspunkte aus M dürfen untereinander in keiner Reihenfolgebeziehung stehen (auch nicht über Sync-Kanten), da die Information sonst redundant wäre. Sie stellen sozusagen die rechten Intervallgrenzen dar für die Information, die S_{Target} schon kennt.
- S_{Source} berechnet die Vereinigungsmenge V aller Nachfolgeraktivitäten von M, die er aus der bei ihm vorhandenen Historie kennt. Wenn $M=\emptyset$ ist, d.h. S_{Target} war noch nicht an der Ausführung der Instanz beteiligt, muß die komplett vorhandene Historie bereitgestellt werden. Bevor S_{Source} die korrespondierende Historieninformation an S_{Target} überträgt, läßt er noch die Einträge aus V weg, welche nicht zu einer Vorgängeraktivität der Migrationsquellaktivität in der aktuellen Iteration korrespondieren, also deren Information an dieser Stelle nicht relevant ist. Damit wird verhindert, daß unnötige Historieneinträge aus parallelen Zweigen verschickt werden.

Das Zusammenführen der Historien auf S_{Target} kann schließlich mit dem in Algorithmus 3-1 beschriebenen History-Merge-Verfahren durchgeführt werden.

Abschließend erfolgt eine formale Darstellung des Algorithmus zur Berechnung der zu verschickenden Historienabschnitte. Es werden hier allerdings nicht, wie eingangs beschrieben, die zu übertragenden Historieneinträge direkt bestimmt. Statt dessen werden von der gesamten Historie alle nicht benötigten Einträge gestrichen.

Algorithmus CalculateHistoryToBeSent

input

$n_{Target} \in$ N: Migrationszielknoten
$n_{Source} \in$ N: Ausgangsknoten der Migration
$i \in$ Integer: Aktuelle Iterationsnummer des Knotens n_{Source}
$H_{complete}$: Kopie der auf dem Ausgangsserver vorhandenen Ausführungshistorie

output

H: die zu verschickende Historie

begin

$H = H_{complete}$;
S_{Target} = Server(n_{Target}); // der Zielserver der Migration

repeat

// Bestimme den nächsten Eintrag, den der Zielserver S_{Target} schon kennt
(n_{del}, i_{del}) = correspondingNode(latestHistoryEntryForServer (H, S_{Target}));
if not n_{del} = NULL **then**

```
        // Streiche alle nicht benötigten, weil auf S_Target bekannten, Einträge aus der Historie
        H = H \ ( pred_i(n_del, i_del, H) ∪ (n_del, i_del) );
    until n_del = NULL;
    // Nur die bei dieser Migration schon relevanten Einträge berücksichtigen
    H = H ∩ ( pred_i(n_Source, i, H) ∪ (n_Source, i) );
end
```

Algorithmus 3-2: Berechnung der zu verschickenden Ausführungshistorie.

Handelt es sich, wie in Abbildung 3-6 der Fall, um eine Migration zu einem Knoten mit nur einer Eingangskante, wird die benötigte Historieninformation in einem Schritt übertragen. Im Fall mehrerer Eingangskanten trägt jede Migration einen Teil der Information bei, so daß erst nach dem Signalisieren aller Migrationskanten die vollständige Information vorhanden ist.

Eine Besonderheit könnte für den letztgenannten Fall berücksichtigt werden. Falls mehrere Migrationskanten von einem Quellserver S_{Source} aus zu einem Zielserver S_{Target} führen (vgl. Abbildung 3-7) - dieser Fall kann bei parallelen Verzweigungen auftreten - könnte wie in Abbildung 3-7 dargestellt bei der Migration gewartet werden, bis auch der letzte vom Quellserver kontrollierte Zweig beendet ist. Im Beispiel übermittelt S2 Historieninformation an S3 somit erst, wenn die Aktivitäten E und G beendet sind. Damit wird die Anzahl an Kommunikationen reduziert, und es kann verhindert werden, daß Informationen doppelt verschickt werden. Anderenfalls würde die zu übertragende Historieninformation mehrfach und für jeden Zweig getrennt berechnet werden, wodurch Überlappungen nicht ausgeschlossen werden können. Im Beispiel würden die zu A und B korrespondierenden Historieneinträge zweimal zum Server S3 übertragen, was vermieden werden sollte. Für diese zusätzliche Optimierung müßte der Algorithmus 3-2 so abgeändert werden, daß nicht nur ein Knoten des Quellservers angegeben und berücksichtigt wird, sondern alle Knoten dieses Servers mit Migrationsziel Z. Dies sind die Knoten E und G.

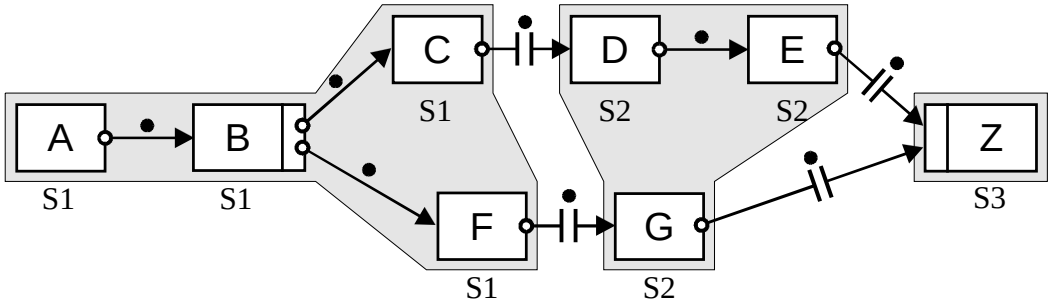


Abbildung 3-7: Eine Workflowinstanz vor einer Migration zu einem Join-Knoten.

Der entsprechende duale Fall eines Knotens mit mehreren Ausgangskanten, z.B. am Beginn einer parallelen Verzweigung, bereitet keine Probleme. Sollten mehrere der Zielknoten vom gleichen Server kontrolliert werden, so genügt eine Übertragung. Der beschriebene Algorithmus darf dann nur mit einem der Zielknoten aufgerufen werden.

Zu erwähnen bleibt, daß Schleifen kein zusätzliches Problem darstellen. Es ist lediglich eine passende Methode zur Berechnung der Vorgängeraktivitäten nötig, die auch die Vorgänger über Schleifenkanten berücksichtigt und somit auch Aktivitäten aus vorangehenden Iterationen der Schleife zurück liefert.

Der oben beschriebene Algorithmus 3-2 hat noch ein Manko, das auch durch die genannte Optimierungsmöglichkeit (Warten an Joinknoten), nicht zu beheben ist. Es kann vorkommen, daß Historieninformation aus parallelen Verzweigungen doppelt übertragen wird. In vielen Fällen wird sogar die komplette Historieninformation zu Aktivitäten, die vor der parallelen Verzweigung liegen, mehrfach übertragen. Da es sich um identische Einträge handelt, werden diese auf dem Zielserver

natürlich dank des History Merging nur einmal berücksichtigt. Das Verfahren ist damit zwar korrekt, jedoch noch nicht optimal bezüglich der Einsparung von zu übertragenden Informationen.

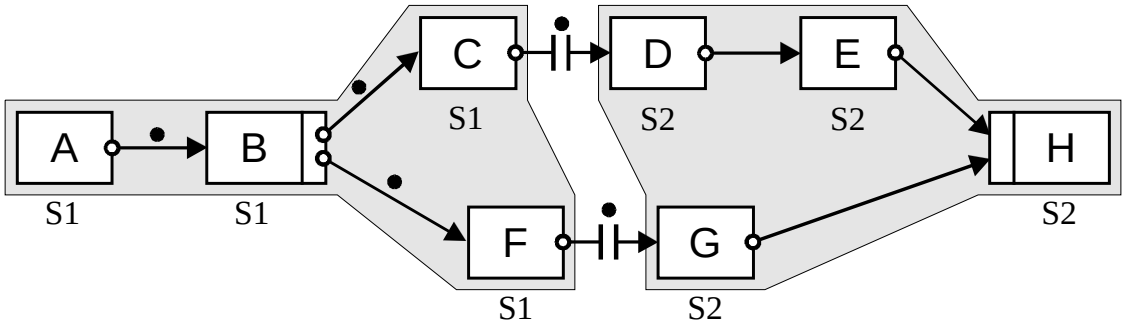


Abbildung 3-8: Eine Workflowinstanz vor Migrationen innerhalb einer parallelen Verzweigung.

Mit dem Algorithmus von oben würde die Historie bis zum Knoten B (einschließlich) bei beiden Migrationen übertragen werden (vgl. Abbildung 3-8). Die Migration $M_{C,D}$ findet unabhängig der Migration $M_{F,G}$ statt, zu verschiedenen Zeitpunkten und in unbekannter Reihenfolge. Man könnte jetzt Einwenden, daß sich S1 ja nur zu merken brauche, welche Information schon an S2 verschickt wurde, um diese dann nicht mehr zu berücksichtigen. Sieht man sich in der folgenden Abbildung 3-9 allerdings das gleiche Beispiel mit einer kleinen Veränderung an, wird klar, daß damit bei weitem nicht alle Fälle abgedeckt sind.

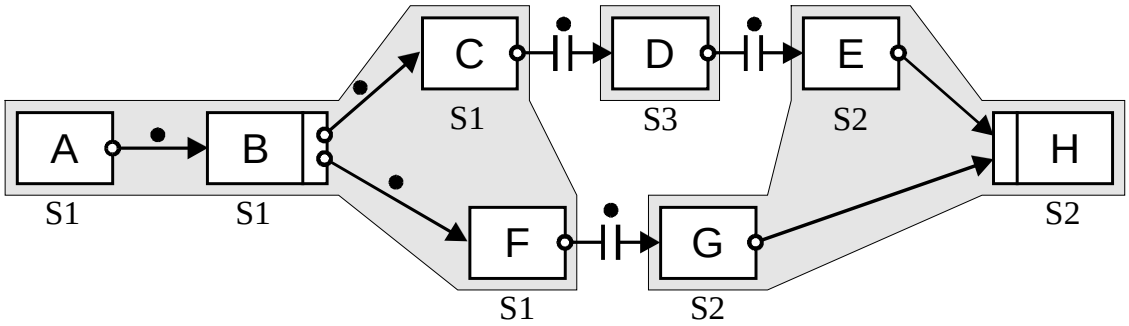


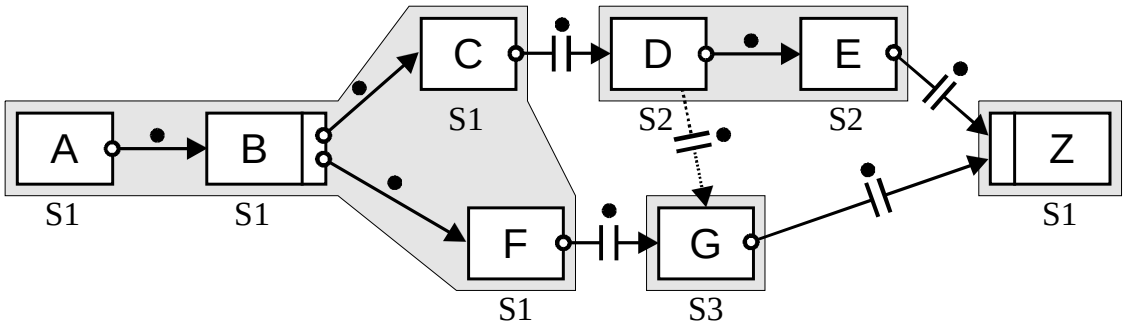
Abbildung 3-9: Redundantes Übertragen von Historieninformation im Schicken-Verfahren.

Durch den zwischen C und E liegenden Knoten D, der durch den Server S3 kontrolliert wird, besteht in dem beschriebenen Schicken-Verfahren keine Möglichkeit mehr, festzustellen, ob die Historie vor B schon an S2 verschickt wurde. Sowohl bei der Migration $M_{D,E}$ auch bei der Migration $M_{F,G}$ muß dieser Teil der Historie an S2 übertragen werden. Wegen der Unabhängigkeit der Aktivitäten in der parallelen Verzweigung sind beide Ablaufreihenfolgen denkbar. Die Ausgangsserver der Migrationen, S1 und S3, kennen diese Reihenfolge jedoch nicht, da sie in dem skizzierten Ansatz nicht miteinander kommunizieren. Bei mehr als zwei parallelen Zweigen kann es sogar notwendig werden, die gleichen Informationen noch häufiger zu verschicken.

3.4.2 Holen-Verfahren

Um die durchschnittlichen Übertragungskosten bei einer Migration weiter zu verringern, wird im folgenden ein verbessertes Verfahren entwickelt. Es schafft Abhilfe für die Fälle mit erhöhtem Kommunikationsaufkommen, die im Schicken-Verfahren bei Migrationen innerhalb paralleler Verzweigungen auftreten können.

Um die Übertragung redundanter Information zu vermeiden, genügt es nicht, über die Historien herauszufinden, wann der Server S_{Target} zum letzten Mal eine Aktivität der WF-Instanz unter Kontrolle hatte. Bei parallelen Verzweigungen ist es ja, wie in Abbildung 3-9 zu erkennen ist, möglich, daß bereits eine Migration eines parallelen Zweiges stattgefunden hat und somit schon ein großer Teil der benötigten Historie zu S_{Target} übertragen wurde. Die Ausgangsserver der Migration S_{Source} können dies im allgemeinen jedoch nicht wissen, da sie unabhängig voneinander arbeiten. Mit dem Schicken-Protokoll kann somit keine weitere Reduktion der zu übertragenden Daten erreicht werden. Um dies zu ermöglichen, wird im folgenden ein Holen-Protokoll verwendet. Es wird in jedem Fall eine zusätzliche Kommunikation mit S_{Target} notwendig, um herauszufinden, welche Information dieser Server schon besitzt. Die Behandlung von Knoten mit mehreren Ein- oder Ausgangskanten geschieht analog dem Schicken-Verfahren.



Ausführungshistorie bei S2 (exemplarisch)	**Ausführungshistorie bei S3 (exemplarisch)**
1 START(A, 1, S1), END(A, 1, S1)	1 START(A, 1, S1), END(A, 1, S1)
2 START(B, 1, S1), END(B, 1, S1)	2 START(B, 1, S1), END(B, 1, S1)
3 START(C, 1, S1), END(C, 1, S1)	3 START(C, 1, S1), END(C, 1, S1)
4 START(D, 1, S2), END(D, 1, S2)	4 START(D, 1, S2), END(D, 1, S2)
5 START(E, 1, S2), END(E, 1, S2)	5 START(F, 1, S1), END(F, 1, S1)
	6 START(G, 1, S3), END(G, 1, S3)

Abbildung 3-10: Vermeidung redundanter Übertragung mit dem Holen-Protokoll.

Anhand des in Abbildung 3-10 dargestellten Beispiels wird noch einmal die Notwendigkeit eines Holen-Protokolls gezeigt und ein Nachvollziehen der notwendigen Berechnungsschritte ermöglicht. Betrachtet werden die beiden Migrationen $M_{E,Z}$ und $M_{G,Z}$. Dabei soll o.B.d.A. zuerst $M_{G,Z}$ stattfinden. Hierbei spielt es noch keine Rolle, ob nach dem Schicken- oder Holen-Protokoll verfahren wird. Es müssen die Zeilen 4 und 6, mit den Einträgen zu den Knoten D und G, übertragen werden.

Bei der zweiten Migration $M_{E,Z}$ spielt das Verfahren jedoch sehr wohl eine Rolle. Nach der herkömmlichen Methode aus Abschnitt 3.4.1 würden die zu D und E korrespondierenden Einträge verschickt werden. Dabei wäre der Eintrag zu D unnötig, da er bereits auf dem Zielserver S1 vorhanden ist. Im Holen-Protokoll werden daher die Intervallgrenzen anders berechnet. S1 berechnet den Migrationspunkt G und teilt diesen S2 mit. Daraufhin kann S2 alle Einträge zu Vorgängern von G (einschließlich G) aus seiner Historie streichen und den verbliebenen Eintrag zu E an S1 mitteilen. So wird durch eine einfache Rückfrage unnötiges Kommunikationsvolumen vermieden.

Der grobe Ablauf des Algorithmus für die Historienübertragung bei einer Migration von S_{Source} nach S_{Target} sieht folgendermaßen aus:

- S_{Source} kündigt dem Zielserver S_{Target} die Migration an und teilt ihm die Prozeßinstanz, die Zielaktivität und deren zukünftige Iterationsnummer mit. Im Beispiel ist dies (Z, 1).
- S_{Target} berechnet die Menge M der früheren Migrationspunkte. Im Unterschied zum Schicken-Verfahren muß der Server zu einem Migrationspunkt aus M nicht mehr unbedingt S_{Target} sein. Die Aktivitäten der Menge M und deren Vorgänger entsprechen den Aktivitäten, zu deren Ausführung S_{Target} schon über Historieneinträge verfügt. Die Migrationspunkte M dürfen untereinander in keiner Reihenfolgebeziehung stehen, da die Information sonst redundant wäre. Sie stellen sozusagen die „rechten Intervallgrenzen für schon bekannte WF-Ausschnitte“ dar.

S_{Target} führt die Berechnung mit Hilfe einer Kopie der bei ihm vorhandenen Ausführungshistorie durch. Er nimmt den neuesten vorhandenen Eintrag seiner Historie, welcher zu einer Vorgängeraktivität der Migrationszielaktivität in der zukünftigen Iteration korrespondiert, und fügt diesen zu M hinzu. Bei der Migration $M_{F,G}$ im Beispiel ist dies zunächst F. Dann streicht er alle Vorgänger davon sowie den Eintrag selbst (berechnet über pred_i) aus der Historie. Sollten noch Einträge in der Historie verblieben sein, wird mit ihnen ebenso verfahren. Im Beispiel wird noch der Eintrag zu C und schließlich kein weiterer mehr gefunden. Aufgrund der Ordnung der Historieneinträge gemäß der Graphstruktur ergeben sich keine redundanten Werte für M.

S_{Target} teilt M an S_{Source} mit: {(F,1), (C,1)}.

- S_{Source} streicht aus einer Kopie der bei ihm vorhandenen Historie alle Einträge zu Vorgängeraktivitäten von M (einschließlich M). Sollte M=∅ sein, wird trivialerweise nichts gestrichen und die komplette Historie beibehalten. Findet zunächst die Migration $M_{E,Z}$ statt, sind die relevanten Historieneinträge gegeben durch {START(D, 1, S2), END(D, 1, S2), START(E, 1, S2), END(E, 1, S2)}. Findet zuerst die Migration $M_{G,Z}$ statt, erhält man folgende relevante Einträge: {START(D, 1, S2), END(D, 1, S2), START(G, 1, S3), END(G, 1, S3)}.

  Bevor S_{Source} die relevanten Historieninformationen an S_{Target} überträgt, werden noch die Einträge entfernt, die nicht zu einer Vorgängeraktivität der Migrationsquellaktivität in der aktuellen Iteration korrespondieren, also deren Information an dieser Stelle nicht benötigt wird. Da im Beispiel kein Zweig mit zu Z parallelen Aktivitäten existiert, werden keine Einträge gestrichen.

Nach der Übertragung der Teilhistorie von S_{Source} an S_{Target} kann der Zielserver seine Historie um die neuen Informationen ergänzen. Hierzu wird wieder der Algorithmus 3-1 (History-Merge) verwendet. Da jetzt von der Voraussetzung ausgegangen werden kann, daß niemals doppelte Historieneinträge verschickt werden, kann ein noch effizienteres Verfahren für das Zusammenführen der Historien benutzt werden. Dieses wird im Anschluß die beiden folgenden Algorithmen beschrieben.

Hier zunächst der Teil des Algorithmus, der vom Migrationszielserver durchgeführt werden muß:

Algorithmus CalculateMigrationPoints

input
$n_{Target} \in$ N: Migrationszielknoten
i ∈ Integer: Zukünftige Iterationsnummer des Knotens n_{Target} (= aktuelle Iterationsnummer + 1)
H: Kopie der auf dem Zielserver vorhandenen Ausführungshistorie

output
M: { n ∈ N, i ∈ Integer }: Menge von Migrationspunkten als Intervallgrenzen

begin
M = ∅; // die Rückgabemenge
// Nur die bei dieser Migration schon relevanten Einträge berücksichtigen
H = H ∩ pred_i(n_{Target}, i, H);

repeat
// Finde den aktuellsten vorhandenen Historieneintrag und übernehme den korrespondierenden
// Knoten als Migrationspunkt
(n_{del}, i_{del}) = correspondingNode(latestHistoryEntry (H));
M = M ∪ (n_{del}, i_{del});
if not n_{del} = NULL **then**
// Streiche alle korrespondierenden Historieneinträge zu Vorgängern des Migrationspunktes
H = H \ (pred_i(n_{del}, i_{del}, H) ∪ (n_{del}, i_{del}));
until n_{del} = NULL;

end

Algorithmus 3-3: Berechnung der Migrationspunkte als Intervallgrenzen.

Der Algorithmus terminiert in jedem Fall, da nach dem Streichen aller Historieneinträge zu den gefundenen Migrationspunkten und deren Vorgängern kein Eintrag mehr in H vorhanden ist. Der Versuch, einen weiteren solchen Historieneintrag zu finden, liefert dann NULL zurück und die Endebedingung der Repeat-Schleife ist erfüllt.

Im Anschluß berechnet der Ausgangsserver die zu übertragende Historie:

Algorithmus CalculateHistory

input

$n_{Source} \in N$: Ausgangsknoten der Migration
$i \in$ Integer: Aktuelle Iterationsnummer des Knotens n_{Source}
M: { $n \in N, i \in$ Integer }: Menge von Migrationspunkten als Intervallgrenzen
$H_{complete}$: Kopie der auf dem Ausgangsserver vorhandenen Ausführungshistorie

output

H: die zu übertragende Historie

begin

$H = H_{complete}$;
// Jeden übergebenen Migrationspunkt berücksichtigen

for each (n, i) $\in$ M **do**
// Streiche alle nicht benötigten, weil auf dem Zielserver bekannten, Einträge aus der Historie
$H = H \setminus (pred\_i(n, i, H) \cup (n, i))$;
// Nur die bei dieser Migration schon relevanten Einträge berücksichtigen
$H = H \cap (pred\_i(n_{Source}, i, H) \cup (n_{Source}, i))$;

end

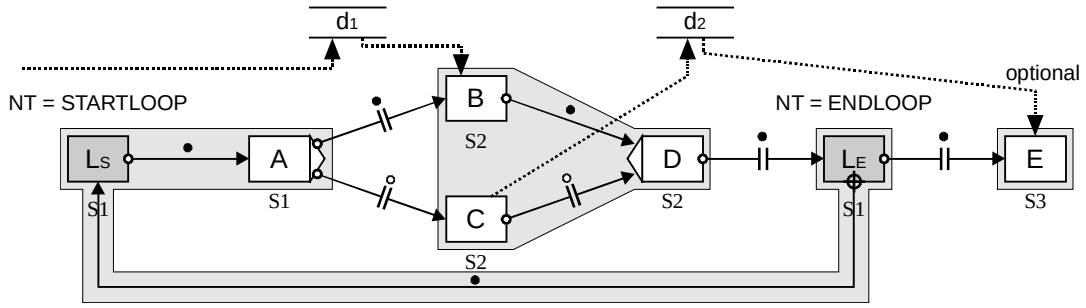
Algorithmus 3-4: Berechnung der zu übertragenden Historie im Holen-Verfahren.

Wenn das beschriebene Holen-Verfahren verwendet wird, kann eine optimierte Methode zum Zusammenführen der Historien benutzt werden.

Migrationen werden in der Reihenfolge ihres Eintreffens beim Zielserver bearbeitet und bei jeder Migration wird zunächst das Maximum an möglicher zugesicherter Information übertragen. Daher ist automatisch gewährleistet, daß die ältesten Historienabschnitte, die den dem Startknoten am nächsten liegenden Aktivitäten entsprechen, zuerst übertragen werden. Entsprechende Aktivitäten von Teilhistorien, die danach übertragen werden, liegen in der Graphstruktur nach oder parallel zu den Aktivitäten, deren Historie bereits übertragen wurde, niemals jedoch davor oder gar dazwischen. Damit bleibt die neue Historie beim Zielserver gültig, wenn in diesem Fall die komplette erhaltene Historie an die vorhandene Historie angehängt wird! So fällt die Behandlung von einzelnen Historieneinträgen und die Überprüfung auf Duplikate weg, was einen Performanzgewinn mit sich bringt.

3.4.3 Mögliche zusätzliche Optimierung für zyklische Graphen

Für WF-Graphen mit Schleifen sind weitere Optimierungen denkbar, die sich insbesondere bei großer Anzahl an Iterationen als vorteilhaft erweisen. Sie können bei Bedarf in einem WfMS eingesetzt werden. Zukünftigen Servern, deren Aktivitäten im Kontrollfluß einem Schleifenendknoten folgen, können bei der Übertragung der Ausführungshistorie nur die Historieneinträge über die jeweils letzte Iteration der in der Schleife enthaltenen Aktivitäten mitgeteilt werden (in Abbildung 3-11 fett gedruckt). Diese Informationen reichen aus, um die Markierungen des WF-Instanzgraphen korrekt anpassen zu können. Allerdings sind dann Zugriffe auf Informationen früherer Schleifeniterationen nicht ohne weiteres möglich (vgl. Abbildung 3-11).



Ausführungshistorie bei S3 (exemplarisch)

5	...	14	START(D, 2, S2), END(D, 2, S2)
6	**START(L_S, 1, S1), END(L_S, 1, S1)**	15	START(L_E, 2, S1), END(L_E, 2, S1)
7	START(A, 1, S1), END(A, 1, S1)	16	START(L_S, 3, S1), END(L_S, 2, S1)
8	START(B, 1, S2), END(B, 1, S2)	**17**	**START(A, 3, S1), END(A, 3, S1)**
9	START(D, 1, S2), END(D, 1, S2)	**18**	**START(B, 2, S2), END(B, 2, S2)**
10	START(L_E, 1, S1), END(L_E, 1, S1)	**19**	**START(D, 3, S2), END(D, 3, S2)**
11	START(L_S, 2, S1), END(L_S, 2, S1)	**20**	**START(L_E, 3, S1), END(L_E, 3, S1)**
12	START(A, 2, S1), END(A, 2, S1)	**21**	**START(E, 1, S3)**
13	**START(C, 1, S2), END(C, 1, S2)**	22	...

Abbildung 3-11. WF-Instanz mit Schleife und darin enthaltener bedingter Verzweigung.

Folgendes Vorgehen ist zur zusätzlichen Reduktion der zu übertragenden Historie notwendig:

1. Bestimmung abgeschlossener Schleifen, d.h. dem letzten Eintrag eines Schleifenendknotens folgt kein Eintrag des zugehörigen Schleifenanfangsknotens mehr[8].
2. Suchen des zu dem in 1. gefundenen Schleifenendknoten zugehörigen Schleifenstartknotens L_S in der Historie. Es ist nicht der nächste Eintrag zu L_S, sondern der mit der niedrigstmöglichen Iterationsnummer zu finden (nicht unbedingt die erste Iteration, da die zu reduzierende Schleife selbst verschachtelt in einer noch nicht abgeschlossenen Schleife enthalten sein kann).

   Das Verfahren entspricht also dem Auffinden der korrespondierenden Klammer in einem Term.
3. Während Schritt 2 ausgeführt wird, können von allen überlesenen Aktivitäten aus dem Schleifenkörper jeweils diejenigen mit der höchsten Iterationsnummer in einen Zwischenspeicher übernommen werden. Ist der gesuchte Historieneintrag des Schleifenanfangsknoten L_S gefunden (siehe 2.), kann der gesamte mit dieser Schleife geklammerte Bereich ausgetauscht werden gegen die zwischengespeicherten Einträge incl. der beiden Schleifenknoten am Anfang und am Ende. Die verbleibende Historie (die Zeilen 6, 13 und 17-21) ist im Beispiel fett hervorgehoben.

Eine solche nochmalige Datenreduktion wirkt sich vor allem auf die Kosten von Migrationen an späteren Stellen im Workflow aus, wenn zu einem Server migriert wird, der bisher nicht an der Ausführung beteiligt war. Besonders bei sehr vielen Schleifeniterationen wird der Einspareffekt spürbar. Allerdings muß bei jeder Migration ein durch O(Anzahl zu übertragender Historieneinträge) beschränkter Zusatzaufwand betrieben werden, um in den Historienausschnitten Schleifen zu finden.

[8] Genau genommen muß nach Schleifenendknoten gesucht werden, welche über die ausgehende Kontrollkante und nicht über die Schleifenkante verlassen wurden [Wei97]. Ansonsten dürfte im Fall einer Migration über eine Schleifenkante der letzte Schleifenendknoten für die Optimierung nicht berücksichtigt werden, sonst würde bereits eine noch nicht abgeschlossene Schleife optimiert.

3.5 Übertragung von Datenelementen

Noch nicht berücksichtigt wurden bisher die eigentlichen WF-Daten (vgl. Kapitel 2). Diese müssen beim Zielserver, ebenso wie WF-Kontrolldaten, bereitgestellt werden, um mit der WF-Kontrolle fortfahren zu können.

Für die Übertragung der WF-Daten gibt es mehrere Möglichkeiten. Diese unterscheiden sich hinsichtlich Analyse- und Implementierungsaufwand und weisen ein differenziertes Verhalten mit mehr oder weniger gutem Optimierungsertrag auf.

3.5.1 Übertragung der kompletten Information

Die naheliegendste Lösung sieht vor, daß die komplette auf dem Quellserver verfügbare Dateninformation übertragen wird. In diesem Verfahren werden bei jeder Migration, neben der kompletten Ausführungshistorie, auch alle Datenelementwerte übertragen. Ein Vorteil dieser Lösung ist, daß keine Berechnungen durchgeführt werden müssen, zur Feststellung welche Daten auf dem Zielserver wirklich benötigt werden. Leider werden dabei keine Möglichkeiten zur Reduktion der zu übertragenden Datenmenge berücksichtigt, so daß sehr viel unnötige Kommunikationslast anfällt.

Dadurch, daß in den Datenelementen Datenversionen gehalten werden, nimmt deren Größe mit dem Fortschreiten des Workflow zu. Dies bedeutet, daß Migrationen zunehmend teurer werden, wenn die vorhandenen Informationen jedesmal komplett weitergegeben werden.

3.5.2 Einsparmöglichkeiten durch Nutzen der Eigenschaften von ADEPT

Wenn man die Vorteile des ADEPT-Basismodells ausnutzt, müssen nicht immer die kompletten WF-Datenelemente übertragen werden (vgl. [RBD98]).

Da Datenelemente WF-Datenversionen speichern, um später ein evtl. Rollback zu ermöglichen, kann in Bezug auf Migrationen eine Optimierung mit sehr hohem Einspareffekt erreicht werden. Es genügt, bei Migrationen immer die für die Zielaktivität aktuell gültigen Datenelementwerte weiterzugeben, da die älteren Versionen nur im Falle eines Rollback benötigt werden. Bei einem Rollback werden in jedem Fall die Server beteiligt, auf denen die jeweiligen Datenelemente ursprünglich geschrieben wurden. Die Optimierung führt also zu keiner Unverträglichkeit mit dieser Erweiterung.

Migrationen werden bei diesem Ansatz zwar zunehmend teurer, da mit dem Fortschreiten des WF Einträge für Datenelemente hinzukommen, die bisher leer waren und deshalb nicht berücksichtigt werden mußten. Häufig gibt es aber einen „Punkt“ im Workflow, nach dessen Überschreiten die Anzahl der bei einer Migration zu übertragenden Datenelemente nicht mehr zunimmt.

3.5.3 Migration von Datenelementen „On Demand“

Eine andere Lösung ist es, bei Migrationen zunächst auf das Übertragen der Datenelementwerte zu verzichten. Erst wenn tatsächlich ein Lesezugriff auf ein Datenelement stattfindet, kann eine Art Cache-Manager überprüfen, ob die gewünschten Daten auf dem Server schon vorhanden sind oder erst noch geholt werden müssen. Mit den WF-Kontrolldaten läßt sich bestimmen, auf welchen Servern die benötigten Datenelementwerte geschrieben wurden und somit von woher sie bezogen werden können.

Es können auch noch Optimierungen in Bezug auf die Wahl einer möglichst kostengünstigen Bezugsquelle erfolgen. Eine solche Optimierung wird in dem im nächsten Abschnitt beschriebenen Verfahren vorgenommen.

Ist ein benötigter Datenelementeintrag bereits vorhanden sein, wird keine Übertragung erforderlich. Mehrfachübertragungen sind bei diesem Verfahren somit von vornherein ausgeschlossen.

Ein Vorteil dieses Verfahrens ist es, daß Datenelemente nur dann übertragen werden müssen, wenn sie auf dem Zielserver auch benötigt werden. Werden z.B. in bedingten Verzweigungen einzelne Teilzweige nicht ausgeführt, so müssen die von ihnen gelesenen Datenelemente auch nicht vorhanden sein.

Ein Nachteil dieses Verfahrens ist die potentielle Verzögerung beim Start der Aktivitäten. Jedesmal, wenn zuvor noch Daten in größerer Menge, dazu vielleicht noch bei einem oder mehreren schlecht angebundenen Servern besorgt werden müssen, muß erst auf das vollständige Eintreffen der Daten gewartet werden. Darüber hinaus wird so eine hohe Anzahl von Kommunikationen erzeugt, da für jede Aktivität die Daten evtl. separat von verschiedenen Servern geholt werden müssen.

Bei diesem Verfahren ist es auch nicht ohne weiteres möglich, einen Server zur Bearbeitung eines Teilgraphen für eine gewisse Zeit vom Netz abzutrennen. Es würden häufig nicht versorgte Datenelemente existieren, so daß mit der Ausführung nicht fortgefahren werden könnte. Derselbe Fall tritt ein, wenn ein Server aus einem anderen Grund zeitweise nicht verfügbar ist. Die Anforderungen an die Verfügbarkeit aller an der Ausführung einer Instanz beteiligten Server sind extrem hoch.

3.5.4 Optimierte Lösung

Aufgrund der vorangehend genannten Nachteile der skizzierten Extremlösungen wird in der Folge eine dritte Lösung entwickelt, die den Großteil der positiven Eigenschaften der „On Demand“-Lösung mit dem einzigen Pluspunkt der „naiven“ Lösung verbindet (vgl. Abschnitt 3.5.1). Die Verzögerungen bei der Informationsübertragung über das Netzwerk treten nur zum Zeitpunkt der Migrationen auf. Benutzer müssen nicht warten, wenn sie eine Folgeaktivität ausführen wollen, die vom gleichen Server wie die soeben bearbeitete kontrolliert wird. Kommunikation findet nur während den Migrationen statt.

Bei den Migrationen wird vorausberechnet, welche Datenelemente für die Arbeitsschritte des Zielservers benötigt werden. Nur diese werden übertragen, zusätzlich wird bei der Übertragung jeweils die günstigste Bezugsquelle ausgewählt. Auch hier sind die Anforderungen an die Verfügbarkeit der WF-Server sehr hoch, da sich eine Migration verzögert, sobald einer der benötigten Quellserver ausfällt[9].

Im folgenden Abschnitt 3.6 werden Varianten dieser Lösung samt zugehöriger Algorithmen und Beispielen ausführlich beschrieben.

3.6 Optimierungen für die Datenelementübertragung

Im folgenden wird vorausgesetzt, daß jeweils nur die *aktuelle Version eines Datenelementes* verschickt wird. Hier sind, wie zuvor erwähnt, wirkungsvolle Optimierungen möglich. Zuerst soll aber nur gezeigt werden, in welchem Umfang Kommunikationskosten gespart werden können.

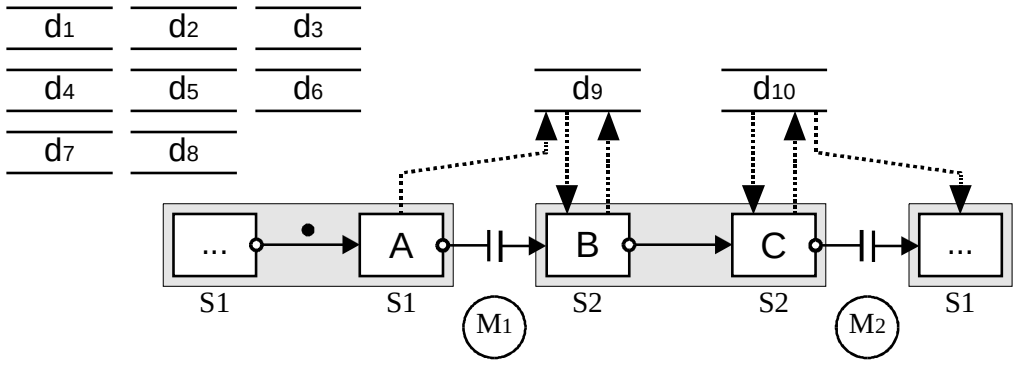


Abbildung 3-12: Sparpotential bei Übertragung nur der benötigten Datenelemente.

Hierzu ein einfaches Rechenbeispiel (vgl. Abbildung 3-12): Die Gesamtgröße einer WF-Instanz betrage 1 MB. An dieser Stelle soll die Übertragung der Arbeitslisteninformation vernachlässigt werden. Ein durchschnittlicher Arbeitsschritt bearbeite jeweils 100 kB Daten lesend und schreibend,

[9] In diesem Fall könnte versucht werden, daß Datenelement falls möglich von einer anderen Quelle zu beziehen. Dieser Aspekt wird allerdings ausgeklammert.

d.h. er verursacht Netzlast in Höhe von ca. 200 kB. Der Workflow bestehe aus einer einfachen Sequenz von Aktivitäten mit beispielsweise 40 Schritten, die auf zwei verschiedenen Servern S1 und S2 ausgeführt werden können. Für die Verringerung der Last im Gesamtsystem würde sich eine Migration von S1 auf S2 erst ab einer Anzahl von 10 Schritten auf S2 lohnen, in diesem Fall sind die Migrationskosten von 1 MB + 1 MB = 2 MB gleich hoch wie die bei der Ausführung der Arbeitsschritte anfallenden Übertragungen über die Netzgrenze hinweg (10 * 200 kB für die Kommunikation zwischen Client und Server sind ebenfalls 2 MB).

Wenn man davon ausgeht, daß in dieser gefundenen „Teilsequenz“ (Aktivität B und C der Abbildung) nur eine Teilmenge der gesamten Daten benötigt werden (d_9 und d_{10}, jedes Datenelement im Beispiel steht für 100 kB Daten), werden bei einem nicht optimierten Verfahren mit hoher Wahrscheinlichkeit sehr viele für die Teilsequenz nicht benötigte Daten bei der Migration übertragen. Im Beispiel müssen nur ein Fünftel der gesamten Daten der Instanz, nämlich 200 kB, übertragen werden. Die Grenze für die Rentabilität von Migrationen liegt im Beispielszenario nicht mehr bei 10 Aktivitäten, sondern bei 2 Aktivitäten. Bei diesem Rechenbeispiel wurde die Übertragung der WF-Kontrolldaten gegenüber den eigentlichen Daten vernachlässigt.

In [BD97] wird ein Verfahren vorgeschlagen, um eine bezüglich der Kommunikationskosten optimale Verteilung der Aktivitäten auf unterschiedliche WF-Server zu berechnen. Die Berechnung basiert auf konstanten Migrationskosten für alle Migrationen. Mit der Optimierung könnten daher sogar mehr rentable Migrationen stattfinden und die Kommunikationsmenge weiter reduziert werden.

Bevor in den Abschnitten 3.6.2 und 3.6.3 zwei optimierte Lösungen zur Übertragung von Datenelementen vorgestellt werden, soll hier noch eine Einschränkung getroffen werden. Diese Optimierungen sollen nur auf Datenelemente ab einer bestimmten Mindestgröße angewandt werden. Für kleine Datenelemente erweist sich eine andere Behandlung als vorteilhaft.

3.6.1 Abweichende Behandlung kleiner Datenelemente

Die besondere Situation bei der Übertragung kleiner Daten, bei denen die Nutzdaten sehr viel geringer als die Mindestpaketgröße des Netzwerks oder eine andere als Schwellwert zu benutzende Größe sind, sollte adäquat berücksichtigt werden. Beispielsweise könnten Daten geringer Größe quasi als „global“ in einem bestimmten Workflow zu deklariert und diese immer mit der Ausführungshistorie mitgeschickt werden. In vielen Fällen lassen sich dadurch aufwendige Berechnungen zur Laufzeit sparen, die ihre Vorteile nur bei großen Datenwerten entfalten. Vor allem jedoch wird die Anzahl der Kommunikationen erheblich reduziert.

Eine gute Kompromißlösung ist es, die in den folgenden Abschnitten 3.6.2 bzw. 3.6.3 beschriebene Optimierung nur auf „große Daten“ anzuwenden. Dann lohnt sich i.d.R. auch der Transfer von nur einem Wert, und es muß kein aufwendiges Verfahren zum Zusammenpacken von kleinen Paketen zu größeren Einheiten durchgeführt werden. Kleine Daten, deren Volumen unterhalb eines bestimmten Grenzwertes liegt, können immer mit abgeholt werden, ohne Rücksicht darauf, ob sie auf dem Zielserver auch benötigt werden.

Um zu garantieren, daß dennoch nicht unnötig viel zusätzliche Kommunikation erzeugt wird, muß nur noch gesichert werden, daß auch für die kleinen Datenelemente lediglich die aktuell gültigen Werte übertragen werden und daß im Falle paralleler Verzweigungen kein mehrfaches Übertragen zum selben Server stattfindet.

Ein intelligentes Verfahren überträgt die Daten zusammen mit der Ausführungshistorie, und zwar genau mit den Abschnitten, die zu ihren Schreiberaktivitäten korrespondieren. Wenn ein Server die WF-Kontrolldaten so mitgeteilt bekommt, erhält er damit auch automatisch die aktuellen Werte kleiner Datenelemente. Mit dem in 3.4.2 beschriebenen Holen-Verfahren zur Übermittlung der Ausführungshistorie wird mehrfache Übermittlung identischer Information ausgeschlossen. Dies gilt natürlich entsprechend für die dazu gepackten Datenelemente.

Allerdings kommt es noch vor, daß mehrere Versionen der gleichen Variablen übertragen werden[10]. Wie kann jetzt noch auf die aktuelle Version reduziert werden?

Bei jeder Übermittlung von Ausführungshistorie muß der Migrationsquellserver, also der, der die Historie bereitstellt, folgendes tun:

1. Bestimme zu allen Variablen die aktuellste Version. Die gesuchte Version ist genau die, die zurückgeliefert wird, wenn der Zielknoten der Migration eine solche Variable liest. Diese explizite Berechnung ist nötig, da von allen Variablen mehrere Versionen vorhanden sein können. Bei jedem Schreibzugriff auf ein Datenelement wird bekanntlich ein neuer Datenelementeintrag mit der neuen Version der Variablen erzeugt.
2. Prüfe für jede der in 1. berechneten Variablen, ob für ihre Schreiberaktivität samt zugehöriger Iterationsnummer ein korrespondierender Eintrag in dem zu übertragenden Teil der Historie vorhanden ist. Variablen, zu denen kein passender Eintrag gefunden wird, werden nicht übermittelt, da dies bereits ein anderer Server getan hat bzw. tun wird. Damit ist gewährleistet, daß niemals Variablen redundant übertragen werden.

Somit fällt überhaupt keine zusätzliche Kommunikation für kleine Variablen an, da sie immer zusammen mit der Ausführungshistorie übertragen werden. Bei einer geringen Anzahl solcher Variablen passen diese vielleicht sogar vollständig in die Restkapazität eines Paketes, mit dem die WF-Kontrolldaten übertragen werden. So kann mit geringem Aufwand noch einmal eine erhebliche Verbesserung des Kommunikationsverhaltens erreicht werden.

3.6.2 Schicken-Verfahren

Um zu bestimmen, welche großen Datenelemente auf den Zielserver übertragen werden müssen, bedient man sich eines der im folgenden beschriebenen Verfahren. Ziel ist es, genau die Datenelemente zu übertragen, die von Folgeaktivitäten gelesen werden. Damit wird die geforderte Datenversorgung der Aktivitäten gewährleistet und gleichzeitig die Kommunikation minimiert.

Für ein Schicken-Verfahren gibt es zwei verschiedene Ansätze. Zunächst muß man feststellen, welche Datenelementwerte auf dem Ausgangsserver einer Migration geschrieben wurden und diese Information dann überall dorthin verteilen, wo sie zukünftig benötigt wird.

Folgende Vorgehensweise bietet sich an: Man bestimmt zunächst die Menge D aller Datenelemente, die von einer Vorgängeraktivität des Migrationspunktes auf dem Migrationsquellserver geschrieben wurden. Dabei werden nur die Aktivitäten der aktuellen, zusammenhängenden Partition berücksichtigt. Interessant ist nur die letzte Version des Datenelements, d.h. die Version, die die Zielaktivität erhalten würde, wenn sie das Datenelement liest. Jetzt werden zu jedem $d \in D$ alle Folgeaktivitäten A, die dieses Datenelement lesen, bestimmt. Das heißt, in A werden nur Aktivitäten aufgenommen, die vor der nächsten Schreiberaktivität (einschließlich derselben) des Datenelements d liegen[11]. Jeder Server, der eine Aktivität $a \in A$ kontrolliert, muß die aktuelle Version von d geschickt bekommen. Dies ist also nicht nur der Server der direkten Nachfolgepartition. Dafür gibt es zwei Möglichkeiten:

a) Man schickt die Daten mit dem Kontrollfluß, d.h. über einen Umweg. Ein Teil der Daten muß so noch über weitere Server wandern, die diese Daten eigentlich nicht benötigen. Er wird bei zukünftigen Migrationen weitergegeben, bis er am Ziel angelangt ist.

   Es ist leicht einzusehen, daß diese Option bezüglich der Kommunikationskosten ungünstig ist. Zusätzliche Schwierigkeiten, die ebenfalls gelöst werden müssen, sind z.B. die Auswahl des zu

[10] Wenn beispielsweise die korrespondierenden Historieneinträge zu einer angenommenen Sequenz von Aktivitäten D,E,F und G übertragen werden, und das Datenelement d_1 von Aktivität E und G beschrieben wurde, werden ohne weitere Optimierungen diese beiden Versionen von d_1 zusammen mit dem Historienausschnitt übertragen. Häufig ist dies auch so gewünscht, wenn z.B. auf eine alte Version zurückgegriffen werden können muß.

[11] Daß diese Bestimmung nicht trivial ist, wird im Folgenden in Zusammenhang mit einer bedingten Verzweigung gezeigt (siehe Abbildung 3-14).

verwendenden Zweiges, falls Daten über eine parallele Verzweigung hinweg geschickt werden müßten.

b) Man schickt jedem Server, der die Daten benötigt, diese direkt zu.

Mit dem Verfahren kann vermieden werden, daß ein Server dasselbe Datenelement mehr als einmal zugeschickt bekommt. Dies liegt daran, daß jeweils nur ein bestimmter Server dieses Element verschicken darf und dieser sich daher merken kann, wem er das Datenelement schon geschickt hat. In beiden Fällen gibt es aber keine Möglichkeit zur Anforderung der Daten von einer günstigeren Quelle, da die Daten immer vom Server der sie schreibenden Aktivität aus verteilt werden. Im folgenden soll nur noch die im Kommunikationsverhalten weitaus günstigere Variante b) betrachtet werden.

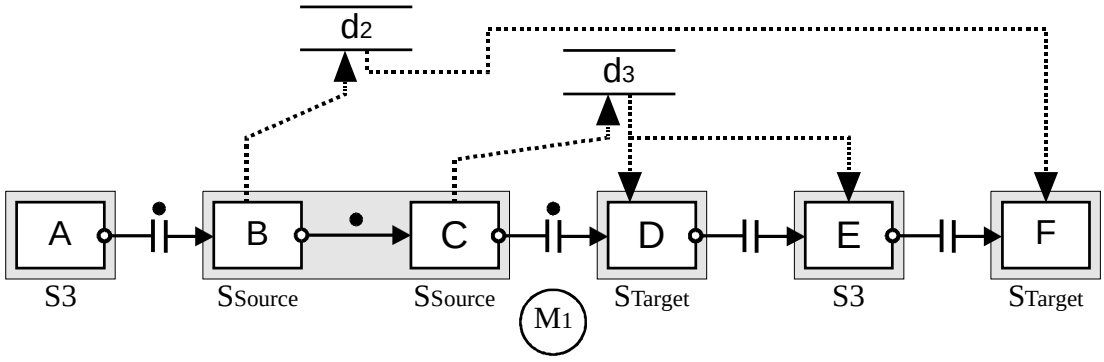


Abbildung 3-13: Übertragen von Datenelementeinträgen beim Schicken-Verfahren.

In Abbildung 3-13 werden bei der Migration M1 alle erforderlichen Datenelementeinträge an die zugehörigen Empfänger mitgeteilt. Dies sind hier die Server S_{Target} und S3, mit drei betroffenen Aktivitäten D, E und F und zwei Datenelemente d_2 und d_3. Die aktuellen Werte von d_2 und von d_3 müssen an S_{Target} geschickt werden. Obwohl die Aktivität E noch in einer von der Struktur her entfernten Partition liegt, muß der aktuelle Wert von d_3 jetzt schon an S3 geschickt werden, da er ja innerhalb der Partition von S_{Source} geschrieben wurde. Bei dieser Schicken-Lösung ist keine Möglichkeit zum späteren Abholen bzw. Schicken (beispielsweise von S_{Target} nach S3) vorgesehen.

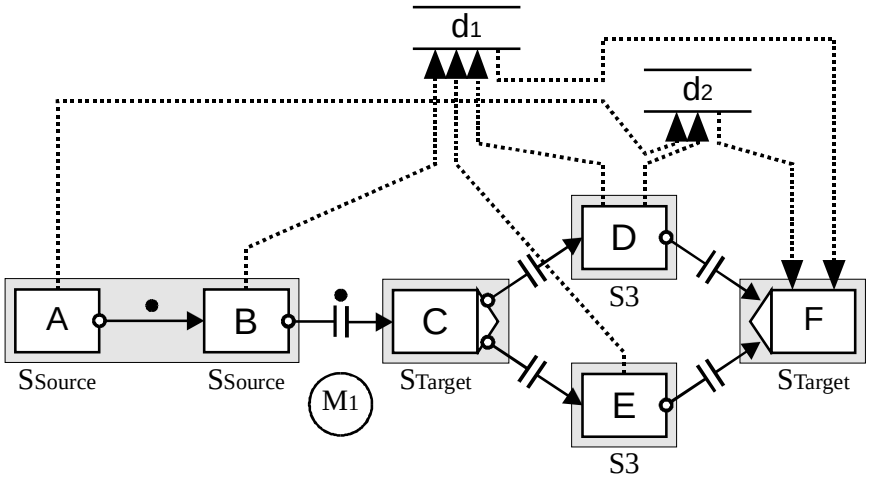


Abbildung 3-14: Verschicken von Daten bei bedingten Verzweigungen.

Liegt im Kontrollfluß nach dem Migrationspunkt eine bedingte Verzweigung, müssen die Daten unabhängig davon, welcher Weg gewählt wird, verschickt werden. Wenn nur eine Aktivität in einem Teilzweig ein Datenelement lesen würde, muß das benötigte Datenelement an den zugehörigen Server geschickt werden, auch wenn die Aktivität später überhaupt nicht ausgeführt wurde, so daß die Daten

gar nicht benötigt wurden. In Abbildung 3-14 liegt ein anderes Szenario vor, welches aber ebenfalls zur Konsequenz hat, daß Daten möglicherweise unnötig übertragen werden.

Das Datenelement d_2 muß bei der Migration M_1 an S_{Target} geschickt werden, obwohl es dort vielleicht gar nicht gebraucht wird. Wenn nämlich in der bedingten Verzweigung der obere Teilzweig mit der Aktivität D ausgeführt wird, wird das Datenelement dort erneut beschrieben und bei der Migration $M_{D,F}$ bekommt S_{Target} dann diese von nun an gültige Version. Wird allerdings der untere Teilzweig gewählt, ist die Kommunikation notwendig, da F dann tatsächlich die von A geschriebene Version von d_2 liest.

Für das Datenelement d_1 ist die Situation anders. Es muß bei Migration M_1 nicht verschickt werden, da zwischen dem Schreiber B und dem Leser F in jedem Fall ein Schreiber (entweder D oder E) liegt.

Die Berechnungen zur Bestimmung der nachfolgenden Leser sind wegen der bedingten Verzweigungen sehr aufwendig. Sie können mit Hilfe der in [Hen97] definierten Algorithmen zur Datenfluß-analyse durchgeführt werden. Für jedes auf der Partition des Migrationsquellservers geschriebene Datenelement $d \in D$ müssen die das Datenelement d lesenden und der Migrationsquellaktivität nachfolgenden Aktivitäten A untersucht werden, ob sie als potentielle Leser in Betracht kommen.

Auf einer Kopie der Prozeßinstanz werden für jede Folgeaktivität $a \in A$ und jedes Datenelement d alle Schreibkanten zu d entfernt, die von Vorgängern der Migrationsquellaktivität ausgehen. Jetzt wird mit writerExists(a, d) geprüft, ob die Datenversorgung von a weiterhin gewährleistet ist. Ist dies der Fall, existiert ein anderer Schreiber und somit muß d nicht an den Server von a geschickt werden. Bei zyklischen Graphen funktioniert dieses Vorgehen nicht mehr.

Ein neu zu entwickelnder Algorithmus zur Bestimmung der potentiellen Leser eines Datenelement-eintrags, der auch in Verbindung mit Schleifen funktioniert, wird wegen der zu writerExists ähnlichen Problemstellung (Schreiben in bedingten Verzweigungen) bezüglich der Komplexität dem writerExists-Algorithmus entsprechen.

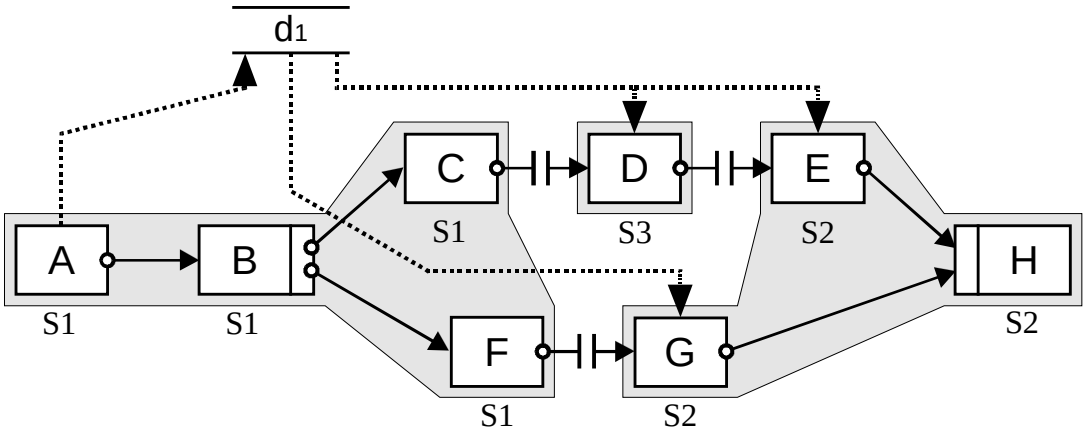
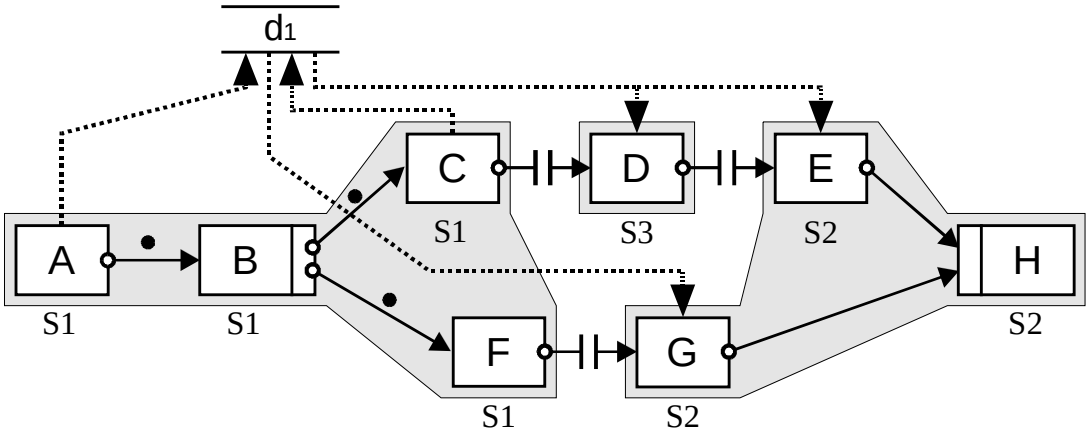


Abbildung 3-15: Verschicken von Daten in eine parallele Verzweigung hinein.

Wenn bei Migrationen in parallele Verzweigungen hinein keine redundanten Übertragungen stattfinden sollen, muß zu jedem Datenelementeintrag gespeichert werden, an welche Server er schon verschickt wurde. Findet zuerst die Migration $M_{F,G}$ statt, so muß bei der Migration $M_{C,D}$ das Datenelement d_1 nur noch an den Server S3 verschickt werden, da S2 dieses Element bereits erhalten hat.

Hier kann auch noch einmal nachvollzogen werden, daß das bereits verworfene Verfahren a), bei dem die Daten nicht direkt zugeschickt werden, prinzipiell ein schlechteres Kommunikationsverhalten mit sich bringt. Der Server S3 müßte nämlich bei der Migration $M_{D,E}$ das Datenelement d_1 in jedem Fall mitschicken. Er kann nicht feststellen, ob die Migration $M_{F,G}$ schon stattgefunden hat und d_1 schon auf S2 vorhanden ist. Es gäbe also noch zusätzliche Fälle unnötiger Kommunikation zu dem ohnehin

schon vorhandenen Nachteil, daß Datenelemente zu Servern geschickt werden, die ein Datenelement gar nicht selbst lesen, sondern es nur zum Weitergeben brauchen.



Ausführungshistorie bei S1
1 START(A, 1, S1), END(A, 1, S1)
2 START(B, 1, S1), END(B, 1, S1)
3 START(C, 1, S1)
4 START(F, 1, S1)
5 END(C, 1, S1)

Historie zum Datenelement d_1
1 A, 1, COMPLETED, „Hallo“
2 C, 1, COMPLETED, „Salut“

Abbildung 3-16: Verschicken mehrerer Versionen des gleichen Datenelements.

Die folgenden Ausführungen beziehen sich wieder auf das Verfahren b), bei dem die Daten direkt zugeschickt werden.

Es ist durchaus möglich, daß ein Datenelement in mehreren Versionen zum Zielserver geschickt werden muß. Die Aktivität G auf S2 benötigt die von A geschriebene Version von d_1, Aktivität E auf S2 benötigt die von C geschriebene Version. Beim Schicken wird dies berücksichtigt, so daß bei jeder Migration nur die passenden Datenelementeinträge übertragen werden.

Weil die Schreiberaktivitäten eines Datenelements in einer Reihenfolgebeziehung stehen müssen, existiert für jeden Datenelementeintrag eine instanzweit eindeutige Versionsnummer. Somit kann der Datenelementeintrag auf dem Migrationszielserver entsprechend seiner Versionsnummer an der richtigen Stelle eingefügt werden. Dabei sind Lücken zulässig. Bei dem optimierten Verfahren dürfen identische Versionen nur einmal zu einem bestimmten Server geschickt werden. Mit einer einfacheren Variante gäbe es aber diesbezüglich keine Probleme, da sich Duplikate feststellen lassen.

Schleifen wurden im Zusammenhang mit dem Schicken-Verfahren bisher noch nicht angesprochen. Sie bereiten keine Probleme, wenn bei der Bestimmung der lesenden Aktivitäten A auch Nachfolger über Schleifenkanten berücksichtigt werden. Für diesen Fall kann der Datenelementeintrag in der für die folgende Schleifeniteration korrekten Version verschickt werden, allerdings ohne Rücksicht darauf, ob auch wirklich eine weitere Schleifeniteration stattfinden wird. Wie bei bedingten Verzweigungen müssen somit Daten verschickt werden, die möglicherweise gar nicht benötigt werden.

Abschließend noch der Algorithmus für das Schicken-Verfahren:

Algorithmus BestimmeZuVerschickendeDatenelemente

input

$n_{Source} \in N$: Ausgangsknoten der Migration
$n_{Target} \in N$: Migrationszielknoten

output

Result: $\{d \in D, n \in N, i \in Integer, s \in S\}$: Menge von Tupeln, die angibt, welcher Datenelementeintrag (eindeutig bestimmt durch n, i) des Datenelements d zu welchem Server s geschickt werden muß

begin

```
Result = ∅;
// Zusammenhängende Vorgängeraktivitäten P von n mit selbem Server wie n ermitteln
P = pred_part(n_Source) ∪ n_Source;
// D seien die Datenelemente, die von einer Aktivität p ∈ P geschrieben werden
D={ d | ∃ p ∈ P ∧ p ∈ writers(d) };
for each d∈D do
    // Die für die Nachfolger gültige aktuelle Version von d bestimmen
    (n_Write, i_Write) = lastWriter(n_Target, d, H);
    // Nachfolgeaktivitäten A von n_Target ermitteln, die d in der gleichen Version lesen
    A = succ_readers(n_Target, d);
    for each a ∈ A do
        if not alreadySent(d, n_Write, i_Write, Server(a)) and Server(a) ≠ Server(n_Source) then
            Result = Result ∪ (d, n_Write, i_Write, Server(a));
```

end

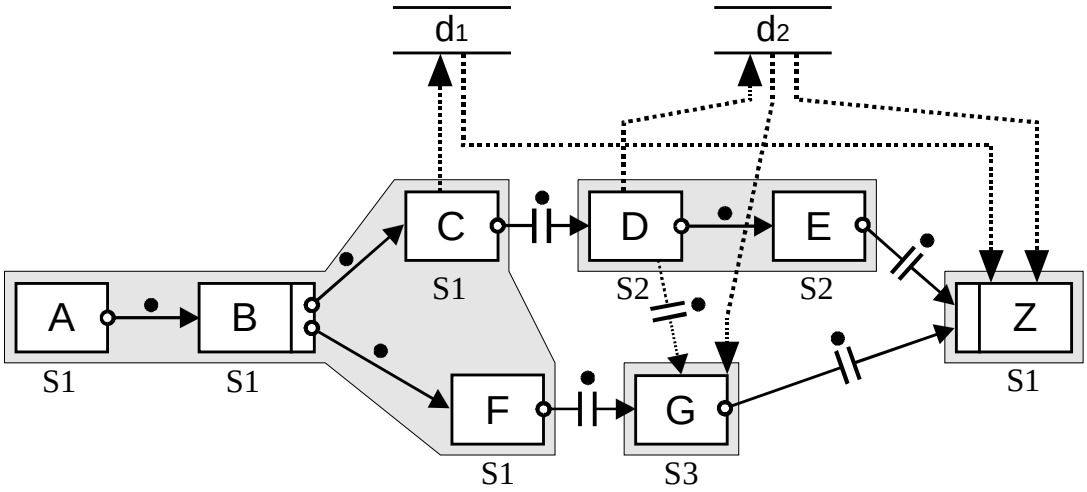
Algorithmus 3-5: Bestimmung der zu verschickenden Datenelemente.

3.6.3 Holen-Verfahren

Um die Übertragungskosten weiter zu senken, wird wie beim Transfer von WF-Kontrolldaten nun zu einem Holen-Protokoll übergegangen. Dabei weiß jeder Server, über welche Daten er bereits verfügt, so daß unnötige Kommunikation vermieden werden kann. Beim Schicken-Verfahren müssen die Daten immer vom Server, auf dem sie geschrieben wurden, bereitgestellt werden. Jetzt gibt es ggf. mehrere Bezugsquellen.

Damit die Möglichkeit, Daten vom günstigsten Server zu beziehen, optimal genutzt werden kann, muß an Knoten mit mehreren Eingangskanten gewartet werden, bis alle signalisiert worden sind. Erst dann kann mit dem Holen der Datenwerte begonnen werden.

An der folgenden Abbildung 3-17 soll dies illustriert werden. Nach den Migrationen $M_{E,Z}$ und $M_{G,Z}$ kann der Wert zum Datenelement d_2 sowohl von dem ursprünglichen Schreiber, d.h. von S2, sowohl als auch von S3, auf dem er ebenfalls vorhanden ist, bezogen werden. Die kostengünstigere Alternative ist zu wählen. Für das Datenelement d_1 fallen keine Kommunikationskosten an, da es ja auf dem Zielserver S1 geschrieben wurde und somit bereits vorhanden ist.



Ausführungshistorie bei S3 (exemplarisch)	
1 START(A, 1, S1), END(A, 1, S1) 2 START(B, 1, S1), END(B, 1, S1) 3 START(C, 1, S1), END(C, 1, S1) 4 START(D, 1, S2), END(D, 1, S2) 5 START(F, 1, S1), END(F, 1, S1) 6 START(G, 1, S3), END(G, 1, S3)	
Historie zum Datenelement d_1 (nur bei S1) 1 C, 1, COMPLETED, „Salut“	**Historie zum Datenelement d_2 (bei S2 und S3)** 1 D, 1, COMPLETED, „Hallo“

Abbildung 3-17: Warten am Joinknoten zum Optimieren der Bezugskosten für Datenelemente.

Vor der Aktivierung jeder Aktivität sollen nur die Datenelemente übertragen werden, die von ihr benötigt werden. Dazu muß folgende Erweiterung gegenüber einem zentralen WfMS vorgenommen werden: Wenn alle eingehenden Kanten der Aktivität so markiert sind, daß die Aktivität nach den Schaltregeln (vgl. [Hen97]) aktiviert werden muß, sind zuvor noch die benötigten Datenelemente zu holen. Erst dann darf die Aktivierung stattfinden, was wiederum zu Folge hat, daß die Aktivität in den Arbeitslisten erscheint und damit ausgewählt und ausgeführt werden kann.

Das Protokoll zum Holen der Datenelemente hat folgenden Ablauf: Zuerst müssen alle Datenelemente bestimmt werden, die von der betreffenden Aktivität A gelesen werden. Zu jedem Datenelement sind die zugehörigen Schreiberaktivitäten bekannt. Interessant ist jedoch nur der letzte Schreiber, welcher im folgenden Algorithmus mit n_{Write} bezeichnet wird. Im einfachsten Fall ist das Datenelement vom Server des letzten Schreibers zu beziehen. Darüber hinaus sind nur solche Schreiber zu berücksichtigen, die vor dem Migrationspunkt liegen. Alle weiteren schon vorhandenen Informationen sind an dieser Stelle noch nicht relevant für den Zielserver. Damit wird hier übrigens analog zur Übertragung der Ausführungshistorie verfahren.

Hier wird jetzt eine zusätzliche Optimierung eingesetzt. Der zuvor festgestellte Datenelementeintrag könnte in identischer Version auf weiteren WF-Servern $S(n_1)..S(n_n)$ vorhanden sein. Dies ist genau dann der Fall, wenn ein Server zuvor eine Aktivität $\in \{n_1..n_n\}$ kontrolliert hat, die diesen Eintrag gelesen hat. Möglicherweise kann der Wert also billiger bezogen werden als von dem Server, auf dem er geschrieben wurde. Sollten sich mehrere Alternativen als gleich teuer erweisen, so muß noch ein weiteres Auswahlkriterium festgelegt werden. Um von vornherein eine ungünstige, weil ungleichmäßige, Verteilung zu vermeiden, kann der Server per Zufall bestimmt werden.

In einigen Fällen kann es vorkommen, daß der Wert auf dem Zielserver bereits vorhanden ist, die Kommunikationskosten betragen dann 0. Dies stellt jedoch lediglich einen Spezialfall für die beschriebene Optimierung dar.

Jetzt können die so berechneten Datenelementeinträge nach Bezugsservern gruppiert werden, damit bei der Anforderung möglichst wenige Kommunikationen anfallen. Nachdem auf diese Art die Daten-

versorgung sichergestellt wurde, kann die Aktivität aktiviert werden und die reguläre Ausführung des Workflows wieder aufsetzen.

Hier nun der zugehörige Algorithmus:

Algorithmus BestimmeBenoetigteDatenelemente

input
n ∈ N: Die Aktivität, die mit Daten versorgt werden soll

output
Result: {d ∈ D, n ∈ N, i ∈ Integer, s ∈ S}: Menge von Tupeln, die angibt, welcher Datenelementeintrag (eindeutig bestimmt durch n, i) des Datenelements d am besten von welchem Server s bezogen werden kann

begin
Result = ∅; // der Rückgabewert
S_n = Server(n);
// Berechnung der Datenelemente D, die von der Aktivität n gelesen werden
D = { d | n ∈ readers(d) };
for each d ∈ D **do**
// Bestimmung des passenden Schreibers für d
(n_{Write}, i_{Write}) = lastWriter(n, d, H);
if not (<d, n_{Write}, i_{Write}> ist schon auf S_n vorhanden) **then**
// Berechnung weiterer Bezugsquellen für benötigten Datenelementeintrag <d, n_{Write}, i_{Write}>
(n_1, i_1)..(n_n, i_n) ={a | a ∈ succ_i(n_{Write}, i_{Write}, H) ∧ a∈ readers(d) ∧ (state(a)=COMPLETED)};
S = guenstigsterServer(S_n, Server(n_{Write}), Server(n_1)..Server(n_n));
Result = Result ∪ (d, n_{Write}, i_{Write}, S);

end

Algorithmus 3-6: Bestimmung der zu übertragenden Datenelemente.

Wären auch optional schreibende Parameter zugelassen, müßte das Protokoll erweitert werden. Dann besteht die Möglichkeit, daß ein Server für einen Datenelementeintrag keine Werte liefern kann, so daß für diese Datenelemente erneut Anfragen stattfinden müssen.

Beim Aufruf von lastWriter[12] in Algorithmus 3-6 würde auch ein Schreiber, der nur optional schreibt, zurückgeliefert werden. Falls dieser nicht geschrieben hat, enthält der Server des optionalen Schreibers nicht in jedem Fall die dann benötigte, zuvor geschriebene Version des Datenelements. Es muß also wiederum lastWriter, jetzt für diese Aktivität, aufgerufen werden. Gegebenenfalls gelangt man so bis zum Startknoten, falls dieser auch über keinen Wert für das Datenelement verfügt, wird NULL zurückgeliefert.

3.6.4 Weitergehende Optimierungsmöglichkeiten zum Holen-Verfahren

An dieser Stelle erfolgt eine Auflistung weiterer Optimierungsmöglichkeiten, die im Laufe der Entwicklung der Verfahren entstanden sind. Sei es, weil diese Methoden unerwünschte Nebenwirkungen mit sich bringen, deren Einführung bei nur geringem Verbesserungseffekt tiefgreifende Änderungen an den bisherigen Algorithmen nötig machen würden oder daß sie nur für in der Praxis kaum auftretende Spezialfälle relevant sind, sie werden nur angesprochen und in dieser Arbeit nicht weiter verfolgt.

[12] Siehe Algorithmus 2-4 in Kapitel 2.

Günstigste Bezugsquelle auch für kleine Datenelemente

Wenn auch für kleine Datenelemente Optimierungen vorgenommen werden sollen, kann ein ähnliches Verfahren wie für große Datenelemente verwendet werden. Allerdings muß hierbei noch zusätzlicher Optimierungsaufwand betrieben werden, um eine große Anzahl von gering ausgelasteten Datenpaketen in der Mindestpaketgröße des Netzwerks zu vermeiden.

Man könnte z.B. versuchen, all diese kleinen Pakete daraufhin zu untersuchen, ob ihre enthaltenen Datenelementeinträge auch von anderen Servern übermittelt werden können. Pakete, in denen schon mindestens ein Datum exklusiv bezogen wird, können nicht weiter optimiert werden. Wenn dies aber nicht der Fall ist, kann ein kleines Datum von einem teureren Server geholt werden, von dem auch noch andere Daten in größerer Menge bezogen werden. Das Datum würde dann lediglich in eine Art Restkapazität dazu gepackt. Es muß ein sinnvoller Trade-Off zwischen Optimierungsaufwand, Übertragungskosten und der Anzahl „ineffizient kleiner Nachrichten“ gefunden werden. Ein Algorithmus für die Ideallösung dieses Problems dürfte NP-vollständig sein, aber möglicherweise lassen sich ja auch mit einem Greedy-Verfahren brauchbare Ergebnisse erzielen.

Holen der Datenelemente für alle Aktivitäten der Zielpartition

Um nicht vor der Ausführung jeder Aktivität Daten übertragen zu müssen - dies würde zu einer hohen Anzahl an Kommunikationen führen - können bei jeder Migration die Datenelemente für alle Aktivitäten der Partition des Zielservers geholt werden. Die Möglichkeiten, die Daten von der günstigsten Quelle zu beziehen, bleiben bei dieser Variante erhalten. An den Algorithmen werden allerdings Änderungen nötig.

Das Protokoll hat folgenden Ablauf: Zunächst wird auf dem Zielserver die mit der Zielaktivität zusammenhängende Partition P berechnet. Die in dieser Partition enthaltenen Aktivitäten $p \in P$ müssen der Zielaktivität über Kontroll-, Sync- oder Schleifenkanten folgen und vom gleichen Server kontrolliert werden. Aktivitäten des Zielservers, welche der Zielaktivität in der Graphstruktur vorangehen, dürfen bei der Übertragung der Datenelemente nicht berücksichtigt werden. Ihre Datenversorgung wird durch eine andere im Kontrollfluß vor diesen Aktivitäten liegende Migration gesichert. Die Daten können nämlich immer nur in Richtung des Kontrollflusses fließen.

Nun muß die Menge aller Datenelemente bestimmt werden, die von einer Aktivität $p \in P$ gelesen werden. Bei dem in Abschnitt 3.6.3 beschriebenen Verfahren ist dies nur eine einzige Aktivität, deren Eingabeparameter versorgt werden sollen. Die Bestimmung der optimalen Bezugsquelle für Datenelemente verläuft hier identisch. Auch ist es möglich, daß ein Datenelementeintrag schon vorhanden ist. Allerdings kann für Leser, deren einziger zugehöriger Schreiber erst auf der Partition P liegt, kein gültiger Datenelementeintrag gefunden und somit auch nicht angefordert werden[13].

Jetzt können die so berechneten Datenelementeinträge nach Bezugsservern gruppiert werden, damit bei der Anforderung möglichst wenige Kommunikationen anfallen. Nachdem auf diese Art die Datenversorgung sichergestellt wurde, kann die betroffene Migrationskante auf „SIGNALED“ gesetzt und mit der Markierung des WF-Graphen fortgefahren werden. Sobald die Startbedingungen der Aktivität auf dem Zielserver erfüllt sind, kann die reguläre Ausführung des Workflows dort wieder aufsetzen.

[13] Auch wenn auf Partition P garantiert ein „blindes Überschreiben“ des Datenelements stattfindet, bevor eine Aktivität auf P das Datenelement liest, muß es nicht übertragen werden. Soll diese Optimierung berücksichtigt werden, muß der Algorithmus lastWriter ergänzt werden.

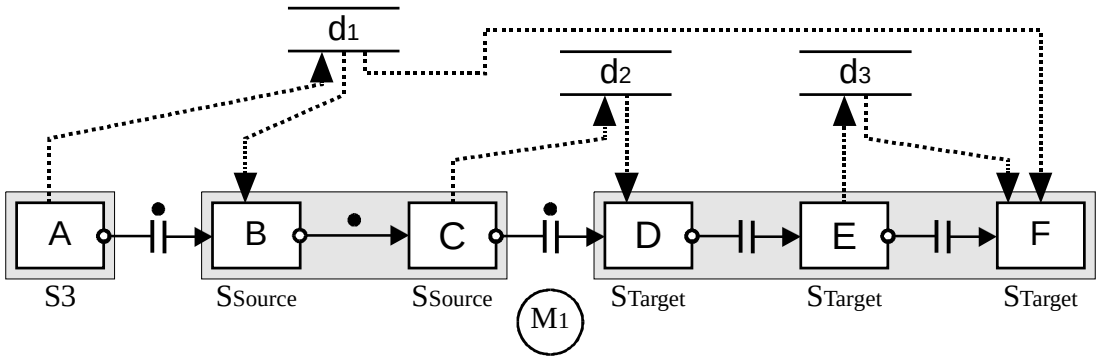


Abbildung 3-18: Übertragung von Datenelementen für eine gesamte Partition im Holen-Verfahren.

Bei der Migration M_1 in Abbildung 3-18 wird die Datenversorgung der Aktivitäten des Zielservers S_{Target} garantiert. Es werden nur die Aktivitäten berücksichtigt, die in derselben Partition wie D liegen, also D, E und F. Zunächst wird festgestellt, daß Werte für alle drei Datenelemente benötigt werden. Da jedoch mit E der letzte Schreiber für d_3 in der Partition enthalten ist, können noch keine Daten geliefert werden. Der Wert für d_2 muß von S_{Source} bezogen werden, bei d_1 besteht eine Auswahlmöglichkeit, da die Aktivität B den Wert ebenfalls gelesen hat.

Mit der beschriebenen Optimierung wird allerdings wieder eine Verschlechterung in Kauf genommen. Bei bedingten Verzweigungen werden zum Beispiel auch Daten für Aktivitäten aus Teilzweigen, die später nicht ausgewählt werden, übermittelt. Dies ist nicht optimal bezüglich der Reduzierung des Kommunikationsvolumens.

Holen der Datenelemente für Teilbereiche der Zielpartition

Um für bedingte Verzweigungen eine Optimierung vorzunehmen, müssen zusätzliche „künstliche Migrationspunkte“ nur für Datenelemente eingebaut werden. Im einfachsten Fall geschieht dies nach jedem bedingten Verzweigungsknoten. Bei allen zuvor stattfindenden Migration dürfen nur die Datenelemente übertragen werden, die nicht ausschließlich innerhalb der bedingten Verzweigung benötigt werden. Die Daten für den gewählten Zweig dürfen erst an dem entsprechenden neuen Migrationspunkt übertragen werden. Somit wird die nutzlose Übertragung von Daten für Zweige, die gar nicht ausgewählt werden, verhindert. Auch diese Optimierung wird durch eine zusätzliche Verzögerung vor der Ausführung eines Schrittes und eine höhere Anzahl an Kommunikationen erkauft. Zusätzlich ist eine Abänderung der Algorithmen zur Migration von Datenelementen notwendig.

Die folgende Abbildung 3-19 a) zeigt einen Fall, in dem eine solche Optimierung sinnvoll ist. Im Teil b) wurden die „künstlichen Migrationspunkte“ eingebaut, an denen nur Datenelemente übertragen werden dürfen. Je unwahrscheinlicher die Auswahl des Zweiges mit der Aktivität Z in dem Beispiel ist, desto höher ist der durchschnittliche Ertrag der Verbesserung.

Normalerweise sind solche Migrationskanten zwischen Aktivitäten des gleichen Servers unzulässig. Die Auswirkungen durch diese Kanten sollten sich daher auf die Migration von Datenelementen beschränken, die Übertragung von Ausführungshistorie und die Durchführung weiterer bei „normalen“ Migrationen anfallenden Aufgaben sind nicht notwendig. Eine im Algorithmus zur Bestimmung der Datenelemente berechnete Partition darf jetzt nicht mehr erst dann enden, wenn eine Aktivität mit anderer Serverzuordnung folgt, sondern muß auch bei erzwungenen Migrationspunkten aufhören.

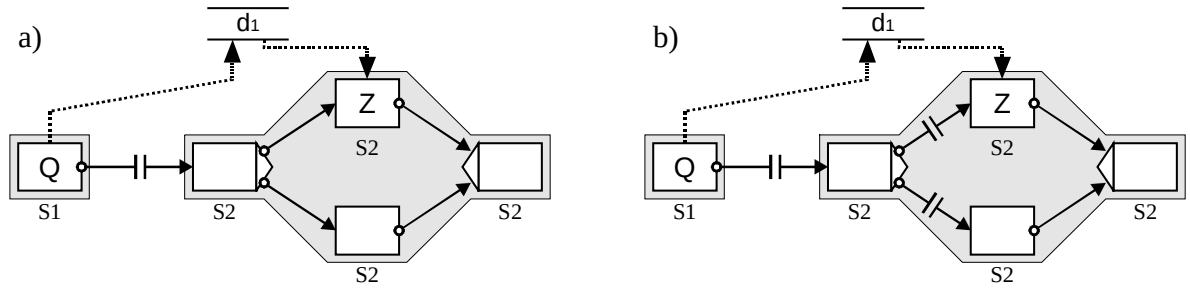


Abbildung 3-19: Zusätzliche Optimierungsmöglichkeit bei bedingten Verzweigungen.

Bei Schleifen könnte man nach dem gleichen Prinzip verfahren. Die „künstlichen Migrationspunkte“ müßten in diesem Fall nach dem Schleifenendknoten eingebaut werden.

Zügigere Abwicklung von Migrationen

Im Falle eines Knotens mit mehr als einer Eingangskante kann die Abwicklung einer Migration in manchen Fällen noch beschleunigt werden. Hier muß abgewägt werden, ob, wie im beschriebenen Verfahren, die Verzögerung durch das Übertragen aller Datenelemente auf einmal wirklich in Kauf genommen werden kann. Diese tritt auf, da auf das Eintreffen aller Zweige gewartet wird, um möglichst viele Informationen vom Server mit den geringsten Kommunikationskosten zu beziehen.

Falls die Kommunikationskosten zu allen Servern auf ähnlichem Niveau sind, können schon nach dem Eintreffen jedes einzelnen Zweiges Datenübertragungen gemacht werden, so daß die letzte Übertragung zum Preis einer größeren Anzahl an Kommunikationen früher fertig ist. Da aber mit der WF-Bearbeitung sowieso auf alle Zweige gewartet werden muß, bringt diese Änderung nur eine sehr geringe Verbesserung. In den meisten Fällen ist die Dauer für die Durchführung der Datenübertragung verschwindend gering gegenüber der Laufzeit des Workflow. Viel problematischer als die Verzögerung aufgrund von Migrationen ist die Verzögerung, die ein Benutzer vom Zeitpunkt der Selektion eines Arbeitsschrittes bis zu dessen Bearbeitung erfährt.

Indirekte Migrationen am Beginn einer parallelen Verzweigung

In eine völlig andere Richtung geht die im folgenden beschriebene, ebenfalls nicht verwendete Optimierung. Bei parallelen Verzweigungen (und nur bei diesen) kann die folgende Situation auftreten: Der Verzweigungsknoten liege auf dem Server S_{Source}, die beiden Zielserver $S_{Target1}$ und $S_{Target2}$ seien nur über eine sehr teuere Verbindung angebunden, können allerdings untereinander günstig kommunizieren. Besser wäre es hier, wenn anstelle von zwei sehr teuren Migrationen zunächst eine Migration beispielsweise nach $S_{Target1}$ stattfindet, und $S_{Target2}$ von $S_{Target1}$ mit Daten versorgt wird. Dies stellt allerdings einen sehr aufwendigen Spezialfall dar, der die Algorithmen deutlich verkomplizieren würde. Eine bessere Lösung wäre es, solche Probleme bereits bei der Modellierung des Workflows zu berücksichtigen. Würde für den Verzweigungsknoten ein leerer Knoten benutzt, könnte dieser problemlos dem Server $S_{Target1}$ zugeordnet werden. Es entstünde dadurch keine zusätzliche Migration, die Informationsübertragung wird aber ohne Abänderung der Algorithmen bedeutend günstiger. Automatisch darf vom System keine zusätzliche leere Aktivität eingefügt werden.

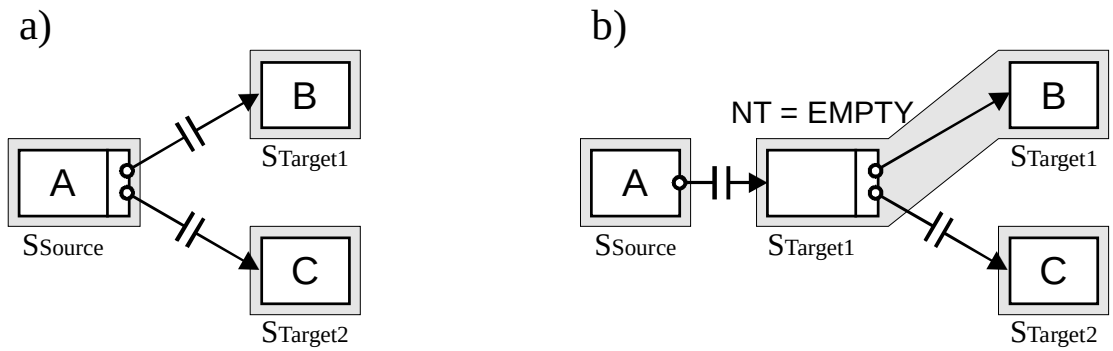


Abbildung 3-20: Mögliche Verringerung der Migrationskosten bei geänderter Graphstruktur.

Abbildung 3-20 a) zeigt den ursprünglichen Graphen und Abbildung 3-20 b) die abgeänderte Version mit leerem Verzweigungsknoten. Die Serverzuordnung des leeren Knotens muß vom Verteilungsalgorithmus sinnvoll berechnet werden. Sie ist nicht an zusätzliche Bedingungen geknüpft. Je nach der Struktur des Netzes könnte er einem der Server der drei angrenzenden Knoten zugeteilt werden. Durch den zusätzlichen leeren Knoten kann daher immer nur eine Verbesserung der Gesamtkosten bewirkt werden.

3.7 Fazit

3.7.1 Vergleichende Betrachtung unterschiedlicher Lösungen

In Tabelle 3-1 werden die drei vorgestellten Grundverfahren in ihren wichtigsten Eigenschaften verglichen. Die Tabelle bezieht sich vor allem auf die Lösungen zum Übertragen der Datenelemente.

Beim kompletten Übertragen wird alles, was an Informationen über eine WF-Instanz vorhanden ist, verschickt. Die On Demand-Lösung steht hingegen genau für das andere Extrem, daß nämlich Daten nur dann migriert werden, wenn sie auch wirklich gebraucht werden, und zwar genau zu diesem Zeitpunkt. Die Varianten der optimierten Lösung verbinden die Vorteile beider Lösungen miteinander und versuchen, die in Kauf genommenen Nachteile gering zu halten. Wenn alle Optimierungsmöglichkeiten eingesetzt werden, würde man auch hier eine maximale Reduzierung des Kommunikationsvolumens erreichen.

	Komplettes Übertragen	On Demand-Lösung	Optimierte Lösung: Daten für jede Aktivität einzeln	Optimierte Lösung: Daten für Teile der Partition
Reduzierung des Kommunikationsvolumens	mangelhaft	sehr gut	sehr gut	sehr gut
Reduzierung der Anzahl der Kommunikationen	ja	gering	gering	ja
Optimierung bei obligaten und optionalen Parametern	keine	ja	ja	ja
Optimierung bei bedingten Verzweigungen und Schleifen	keine	ja	ja	ja
Berechnungsaufwand	gering	mittel	mittel	hoch
Verzögerung vor einzelnen Aktivitäten	nur bei Migration	ja	für Benutzer nicht sichtbar	i.d.R. nur bei Migration

	Komplettes Übertragen	On Demand-Lösung	Optimierte Lösung: Daten für jede Aktivität einzeln	Optimierte Lösung: Daten für Teile der Partition
Anforderungen an die Verfügbarkeit aller Server	durchschnittlich	sehr hoch	sehr hoch	hoch

Tabelle 3-1: Vergleich unterschiedlicher Verfahren zur Datenübertragung bei Migrationen.

In Tabelle 3-2 werden noch einmal die grundlegenden Unterschiede der Schicken- und der Holen-Lösung für die Übertragung von Datenelementen herausgehoben.

	Schicken-Lösung	Holen-Lösung
Redundanz	keine	keine
Bezug von günstigster Quelle	nein	ja
Berechnung der zu übertragenden Daten zur Definitionszeit	weitgehend möglich	nicht sinnvoll
Frühzeitiges Übertragen	ja	nein
Aktiver Server	Migrationsquellserver	Migrationszielserver
Berechnungsaufwand	hoch	mittel

Tabelle 3-2: Vergleich von Schicken- und Holen-Lösung bei der Datenübertragung.

Zusammenfassend läßt sich sagen, daß die Kombination von zueinander passenden Optimierungen für das Übertragen von WF-Kontrolldaten und WF-Daten eine erhebliche Reduktion der Kommunikationslast mit sich bringt. Als bestes Verfahren für beides erweist sich die Holen-Lösung. Ein nicht unerheblicher Vorteil ist hier auch die Offenheit für Erweiterungen. Im Ausblick auf das folgende Kapitel „Migrationen bei variablen Serverzuordnungen„ wird sich diese Lösung sogar als die einzig sinnvolle herausstellen.

Auch im Zusammenhang mit dynamischen WF-Änderungen, auf die in Kapitel 5 ausführlich eingegangen wird, sind die Holen-Verfahren die naheliegenden. Beispielsweise werden beim Einfügen neuer Aktivitäten ohnehin Lösungen nötig, die denen des Holens entsprechen, so daß die bestehenden Algorithmen dafür genutzt werden können.

3.7.2 Orthogonale Konzepte

Für die Realisierung eines guten Gesamtsystems von entscheidender Bedeutung sind auch einige bisher noch nicht angesprochene Konzepte zur Verringerung der Kommunikationslast.

So könnten generell alle Daten zusätzlich noch vor der Übertragung *komprimiert* und danach wieder *entpackt* werden. Manche Systeme bieten dies von sich aus an, z.B. mit einem Zip-Stream zur Informationsübertragung. Dies geschieht dann orthogonal zu den die Anwendungssemantik nutzenden Konzepten. Je nach Beschaffenheit der Daten ist eine beachtliche zusätzliche Reduktion erreichbar. Der Preis dafür ist ein erhöhter Berechnungsaufwand. Die Rechenleistung muß für diesen Zweck auch verfügbar sein.

Falls dem WfMS nicht nur die Daten selbst, sondern auch die *Änderungsoperationen*, welche den Unterschied zwischen zwei Versionen eines Datenelements ausmachen, bekannt wären, könnten möglicherweise enorme Einsparungen bei der Übertragung großer Datenelemente erzielt werden. Wenn z.B. in mehreren Arbeitsschritten lediglich die Helligkeit eines Photos verändert wurde und dieses anschließend wieder zu seinem Ursprungsserver gelangt, bräuchte nicht das gesamte Bild

übertragen zu werden. Theoretisch würde es genügen, alle Operationen, die auf diesem Objekt ausgeführt wurden, erneut gesammelt auf dem Zielserver auszuführen. Dies stellt jedoch enorme Anforderungen an das WfMS und erfordert eine enge Integration mit allen Anwendungsprogrammen. Außerdem muß hierbei der zusätzliche Berechnungsaufwand gegenüber der Einsparung an Kommunikation abgewägt werden. Es macht keinen Sinn, ein System ohne viel Kommunikation zu betreiben, bei dem die Rechenkapazität zum kritischen Engpaß wird. Aus Praktikabilitätsgründen scheidet diese Lösung deshalb für die Mehrzahl von WF-Anwendungen wohl aus.

Eine wichtige Grundvoraussetzung ist natürlich eine die Realität gut abbildende Modellierung des Workflow sowie eine effiziente Organisation des Ablaufs. Die Arbeitslast sollte sich im Mittel adäquat auf die eingesetzten Ressourcen verteilen und diese gut ausnutzen. Jedoch ergeben sich im Laufe der Zeit immer Veränderungen. Sollten alle Optimierungsverfahren nicht helfen, z.B. wenn eine Organisationseinheit in der Zwischenzeit zu groß geworden ist, muß entweder ein Teilnetz aufgerüstet werden oder sogar auf herkömmliche Art eine Neupartitionierung des Netzes vorgenommen werden. Gegebenenfalls werden sogar ein oder mehrere zusätzliche WF-Server notwendig. Nur so kann die Arbeitsfähigkeit des Gesamtsystems wiederhergestellt werden. Dabei darf nicht vergessen werden, die Verteilungen der WF-Vorlagen neu zu berechnen.

4 Migrationen bei variablen Serverzuordnungen

Im folgenden Kapitel wird die verteilte Ausführung von Workflows untersucht, die abschnittsweise von verschiedenen Servern kontrolliert werden. Im Gegensatz zu Kapitel 3 sind zur Modellierungszeit noch nicht alle Server bekannt. Einige Server werden erst zur Laufzeit berechnet, indem ein der Aktivität zur Modellierungszeit zugewiesener *variabler Serverzuordnungsausdruck* ausgewertet wird. Es wird untersucht, inwieweit die entwickelten Verfahren angewendet werden können und welche neuen Probleme gelöst werden müssen, um eine Migration im bisherigen Sinn korrekt durchzuführen.

4.1 Variable Serverzuordnungen

Bisher wurde den Aktivitäten einer WF-Vorlage bei der Berechnung einer optimalen Verteilung statisch ein Server zugewiesen. Für den Fall, daß die Bearbeiter jeder Aktivität unabhängig von den Bearbeitern aller anderen Aktivitäten bestimmt werden können, wird mit diesem Ansatz auch ein gutes Ergebnis erreicht. In der Praxis können jedoch die Bearbeiter eines WF-Schrittes abhängig von den Bearbeitern vorausgegangener WF-Aktivitäten sein. So kann es erforderlich sein, daß derselbe Arzt, der eine bestimmte medizinische Untersuchung durchführt (Aktivität X), später auch den zugehörigen Befund schreibt.

In WfMS wird dieser Tatsache Rechnung getragen, indem sogenannte *abhängige Bearbeiterzuordnungen* bei der WF-Modellierung festgelegt werden können. Eine Bearbeiterzuordnung für eine Aktivität ist dann nicht mehr nur ein Term der Form „Rolle = Arzt ∧ Abteilung = Chirurgie“, sondern kann aus Ausdrücken bestehen wie „Bearb(X)“ oder „Rolle = Arzt ∧ OE(Bearb(X))“ und somit vorangegangene Aktivitäten referenzieren (vgl. [BD00]). „Bearb(X)“ bedeutet, daß die Aktivität vom gleichen Bearbeiter ausgeführt werden muß wie Aktivität X. Wenn für den Bearbeiter von Aktivität K „OE(Bearb(X))“ festgelegt ist, muß dieser der gleichen Organisationseinheit angehören wie der Bearbeiter von Aktivität X.

Bei der Bestimmung der optimalen WF-Partitionierung werden die Abhängigkeiten aufgrund variabler Bearbeiterzuordnungen erkannt. Als mögliche Serverzuordnung für eine Aktivität muß jetzt nicht nur jeder Server in Betracht gezogen werden, sondern auch *variable Serverzuordnungsausdrücke* (s.u.). Die kostengünstigste Alternative wird gewählt.

Werden abhängige Bearbeiterzuordnungen verwendet, ist mit einer rein statischen Serverzuordnung das Optimum bezüglich der Reduzierung der Kommunikationslast nicht mehr erreichbar. Ist zum Beispiel für die Aktivität X der Server S5 festgelegt, da sich die meisten potentiellen Bearbeiter im Teilnetz (Domain) des Servers S5 befinden und wird X dann aber von einem Akteur eines anderen Domain bearbeitet, so entsteht für diesen Schritt zusätzliche Kommunikation, da der Bearbeiter nicht im Domain des Servers S5 angesiedelt ist. Bei statischer Serverzuordnung fällt diese zusätzliche Kommunikation auch für alle mit „Bearb(X)“ abhängigen Folgeaktivitäten an. Es wäre daher besser, nur für den Schritt X den Server statisch festzulegen, aber für die folgenden Aktivitäten den Serverzuordnungsausdruck „Domain(Bearb(X))“ zu verwenden. Dann könnte für die Folgeschritte der ideale Server, nämlich der im Domain des tatsächlichen Bearbeiters, benutzt werden.

Für die Festlegung von Serverzuordnungsausdrücken gibt es folgende Möglichkeiten:

1. ServZuordn(K) = S_i

   Der Aktivität K wird der Server S_i statisch zugeordnet.

2. ServZuordn(K) = Server(X)

   Die Aktivität K soll vom selben Server kontrolliert werden wie die Vorgängeraktivität X.

3. ServZuordn(K) = Domain(Bearb(X))

   Der Aktivität K wird der Server zugewiesen, der sich im Domain des Bearbeiters befindet, der die Aktivität X ausgeführt hat. Ist Aktivität K derselbe Bearbeiter zugeordnet wie Aktivität X, so ist der Server von Aktivität K durch diese Zuordnung stets optimal.

Darüber hinaus ist noch eine Variante von 2. und 3. denkbar, bei der zuvor noch eine Funktion auf die Zuordnung angewandt wurde oder sogar beliebige Ausdrücke.

Die Serverzuordnungsausdrücke können vom Modellierer vorgegeben werden. In den meisten Fällen lassen sie sich aber automatisch ableiten. Die Berechnung einer optimalen Verteilung ist sehr aufwendig, da sehr viele Kombinationen berücksichtigt werden müssen. Im weiteren nehmen wir die Serverzuordnungen einer WF-Vorlage als gegeben an.

Zusätzlich werden zur Modellierungszeit noch sogenannte *Serverklassen* berechnet. Sind zwei Aktivitäten in der gleichen Serverklasse enthalten, werden sie bei der Ausführung immer vom selben Server kontrolliert. Dieser muß aber zur Modellierungszeit noch nicht bekannt sein. Des weiteren ist es möglich, daß bei der WF-Ausführung verschiedene Serverklassen „zusammenfallen".

Gegenüber einem herkömmlichen verteilten WfMS ohne variable Serverzuordnungsausdrücke ergibt sich für die Ausführung der Aktivitäten eine Änderung. Bisher wurde für jede Aktivität bereits zur Modellierungszeit statisch festgelegt, auf welchem Server sie ausgeführt wird, jetzt ist zur Ausführungszeit zunächst eine Auswertung der Ausdrücke notwendig. Folglich ist der tatsächliche Server für eine Aktivität nicht immer im voraus bekannt und kann in einigen Fällen erst unmittelbar vor deren Aktivierung bestimmt werden. Dies trifft für die Aktivität B in Abbildung 4-1 zu. Erst nachdem die Aktivität A zu Bearbeitung ausgewählt worden ist, ist der Server für die Aktivität B bestimmt. Aktivität C ist dem gleichen Server zugeordnet wie B und damit in derselben Serverklasse.

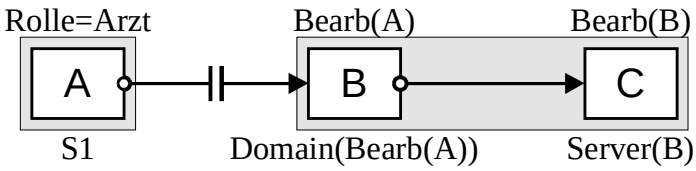


Abbildung 4-1: Sequenz mit variablen Serverzuordnungsausdrücken.

An den Übergängen zwischen den Serverklassen sind zwar mögliche Migrationskanten bekannt. Es ist aber auch möglich, daß gar keine Migration notwendig wird, da die Auswertung des Serverzuordnungsausdrucks den selben Server wie den Server der gerade aktuellen Aktivität ergibt. Falls der Bearbeiter von A im Domain von S1 angesiedelt ist, gibt es zwischen A und B gar keine Migration und die Serverklassen fallen zusammen. In verschiedenen Instanzen der Vorlage sind selbstverständlich unterschiedliche Server als Ergebnis der Auswertung desselben Zuordnungsausdrucks möglich. Das gesamte System verhält sich dynamischer als bei den statischen Serverzuordnungen, und der aktuelle Zustand der WF-Instanz wird besser berücksichtigt.

4.2 Fragestellungen

In diesem Kapitel soll untersucht werden, inwieweit die in Kapitel 3 für statische Serverzuordnungen entwickelten Verfahren noch angewendet werden können. Dabei wird wieder zwischen der Übertragung von WF-Kontrolldaten und von WF-Daten unterschieden. Es wird versucht, so weit als möglich die schon erarbeiteten Lösungen wiederzuverwenden. Dabei ergeben sich aber auch Unvereinbarkeiten und Besonderheiten, für die im Detail Lösungen gefunden werden müssen. Des weiteren zeigt sich, daß die in Kapitel 3 diskutierten Verfahrensvarianten z.T. unterschiedlich gut bei variablen Serverzuordnungen anwendbar sind.

Schließlich wird noch Bezug genommen auf die Auswirkungen, die sich aus den optimierten Migrationen und damit aus der geänderten Berechnung der Ausführungskosten auf den Verteilungsalgorithmus ergeben.

4.3 Transfer von WF-Kontrolldaten

Die in Kapitel 3 beschriebenen naiven Verfahren für den Transfer von WF-Kontrolldaten sollen hier nicht mehr betrachtet werden. Sie könnten zwar auch für den variablen Fall verwendet werden, erwiesen sich allerdings schon bei statischen Serverzuordnungen den Verfahren als unterlegen, die Information zur Ausführungshistorie zu übertragen. Dafür bieten sich auch für den variablen Fall wieder ein Schicken- und ein Holen-Verfahren an. Es ist zu untersuchen, inwieweit die schon beschriebenen Algorithmen benutzt werden können. Dazu formulieren wir zunächst die folgende These:

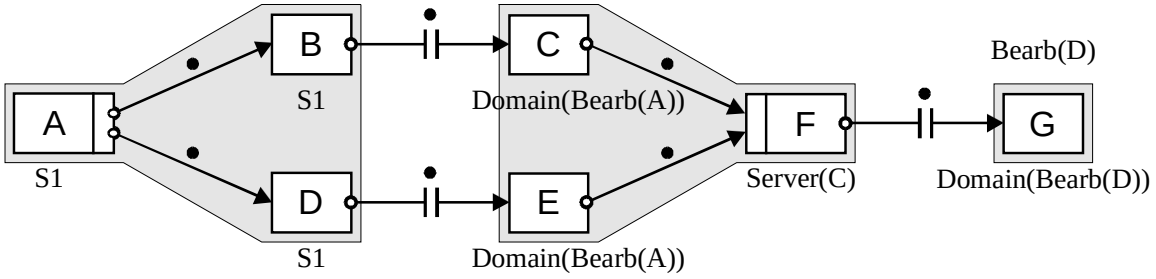
These: Im Fall von variablen Serverzuordnungen können für den Transfer von WF-Kontrolldaten exakt die gleichen Algorithmen wie im statischen Fall verwendet werden.

Die Begründung für diese These ist einfach: Mit der Ausführungshistorie werden Informationen über vergangene und damit bekannte Zustände der WF-Instanz übertragen. Das Problem, daß der Ort von in der Zukunft ausgeführten Aktivitäten möglicherweise noch unbekannt ist, spielt deshalb keine Rolle. Der Migrationszielserver muß zum Zeitpunkt der Historienübertragung bekannt sein, und damit kann wie im statischen Fall verfahren werden.

Das *Schicken-Protokoll* führt allerdings zu Redundanz in der Übertragung. Bei parallelen Verzweigungen ist es möglich, daß die komplette Historieninformation zu Aktivitäten, die vor dem AND-Verzweigungsknoten liegen, mehrfach zum Migrationszielserver geschickt wird. Daher wird diese Lösung hier nicht weiter verwendet.

Beim *Holen-Verfahren* wird dagegen ein Abglich mit der auf dem Zielserver schon vorhanden Information durchgeführt. Der Zielserver teilt daraufhin dem Migrationsquellserver mit, bis zu welchen Aktivitäten im Graphen er schon über die korrespondierende Ausführungshistorie verfügt. Der Quellserver kann anhand dessen seine Historieneinträge filtern, bevor er sie überträgt. Damit wird mehrfaches Übertragen identischer Einträge ausgeschlossen. Der in Abschnitt 3.4.2 definierte Algorithmus 3-3 für den Zielserver und der Algorithmus 3-4, welcher vom Quellserver durchzuführen ist, können unverändert übernommen werden.

Bei variablen Serverzuordnungen bekommt die Historie noch eine zusätzliche Bedeutung. Ausschließlich sie enthält jetzt die Information, auf welchem Server eine Aktivität tatsächlich ausgeführt wurde. Im statischen Fall konnte diese Information aus der WF-Vorlage entnommen werden. Die Struktur der Ausführungshistorie unterscheidet sich jedoch nicht von der bei statischen Serverzuordnungen. Somit können die gleichen Algorithmen für das Mischen der Historien benutzt werden (vgl. Abschnitt 3.3.2).



Ausführungshistorie bei S5 (exemplarisch)

1	START(A, 1, S1), END(A, 1, S1)	5	START(C, 1, S3)
2	START(D, 1, S1), END(D, 1, S1)	6	END(C, 1, S3), END(E, 1, S3)
3	START(E, 1, S3)	7	START(F, 1, S3), END(F, 1, S3)
4	START(B, 1, S1), END(B, 1, S1)	8	START (G, 1, S5)

Abbildung 4-2: Übertragung von Ausführungshistorie bei variablen Serverzuordnungen.

Anhand des in Abbildung 4-2 dargestellten Workflows soll noch einmal die besondere Bedeutung der Historie hervorgehoben werden. Während in der Prozeßvorlage nur Serverzuordnungen enthalten sind, kann der Ausführungshistorie der jeweilige Server entnommen werden. Da sich der Bearbeiter der Aktivität A in der Domain von S3 befindet, wurden die Aktivität C und E dort ausgeführt. Aktivität F wurde ebenfalls von diesem Server kontrolliert. Bei der Migration zwischen F und G mußte die Historieninformation gemäß Zeile 1 bis 7 an S5 übertragen werden, da S5 (in dessen Domain sich der Bearbeiter von D befindet) bisher noch nicht an der Ausführung beteiligt war.

Genauso gut hätte aber auch ein Bearbeiter aus dem Teilnetz von S1 (bzw. S3) die Aktivität D bearbeiten können, dann wäre lediglich die dem Server S1 noch fehlende Ausführungsinformation übertragen worden (bzw. überhaupt keine Migration nötig geworden).

Im Zusammenhang mit bedingten Verzweigungen ist es jetzt von besonderer Bedeutung, das in Abschnitt 3.4 vorgeschlagene Verfahren zu verwenden, um die Durchführbarkeit von Migrationen zu gewährleisten.

Wenn in der bedingten Verzweigung in Abbildung 4-3 der Teilzweig mit den Aktivitäten E und Q nicht gewählt wird, werden der Zustand der Knoten E und Q auf SKIPPED gesetzt und die Kanten des Zweiges einschließlich der von Q ausgehenden Sync-Kante mit FALSE_SIGNALED markiert.

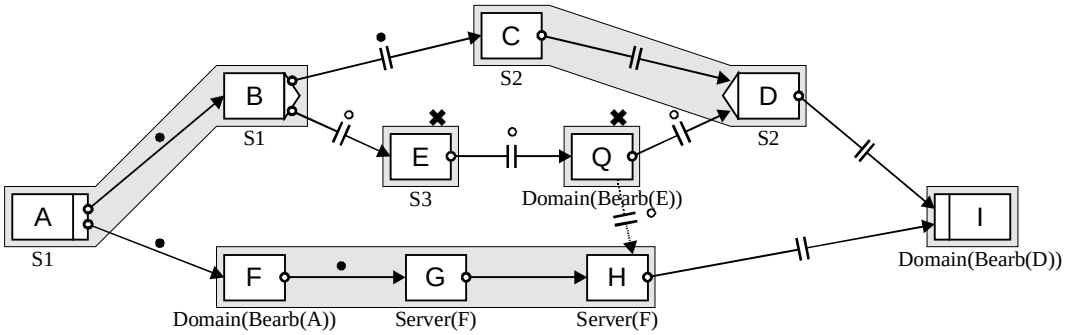


Abbildung 4-3: Übersprungene Aktivitäten in einer bedingten Verzweigung.

In Abschnitt 3.4 wird festgestellt, daß die Migrationen $M_{B,E}$, $M_{E,Q}$ und $M_{Q,D}$ nicht notwendig sind, die Migration $M_{Q,H}$ im Prinzip jedoch schon, da der Server von Aktivität H die Informationen über die Vorgängeraktivität B benötigt. Als Lösung wird vorgeschlagen, anstelle der vier genannten Migrationen die Migration $M_{B,H}$ durchzuführen.

Aus folgendem Grund muß diese optimierte Variante zwingend gewählt werden: Die Migration $M_{E,Q}$ kann nicht ausgeführt werden, da der Zuordnungsausdruck der Zielaktivität nicht ausgewertet werden kann, weil die referenzierte Aktivität E übersprungen wurde.

Für die Verwendung der Zuordnungsausdrücke muß folgende Einschränkung vorgenommen werden: Ein Serverzuordnungsausdruck $ServZuordn_K$ darf nur eine Aktivität referenzieren, die sicher vor K ausgeführt wird, d.h. sie ist Vorgänger von K und nicht in einer bedingten Verzweigung, in der nicht auch K liegt[14]. Beispielsweise wären die Zuordnungsausdrücke „Server(Q)“ oder „Domain(Bearb(Q))“ keine zulässigen Ausdrücke für die Aktivität H.

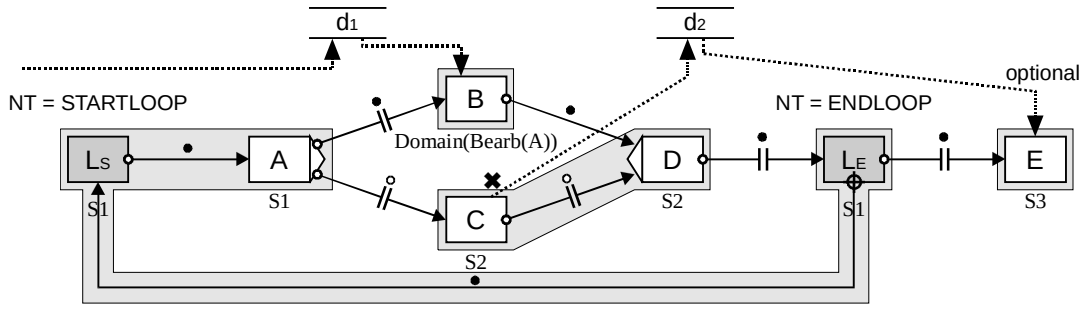
[14] Um die Bezugnahme auf Server innerhalb einer bedingten Verzweigung nicht gänzlich auszuschließen, wurde in [BD00] das Konzept der komplexen Serverzuordnungen entwickelt. Hierbei hängt das Ergebnis der Auswertung ggf. davon ab, welcher vorangehende Teilzweig gewählt wurde.

4.3.1 Mögliche zusätzliche Optimierung für zyklische Graphen

Im Anschluß an die Beschreibung der Verfahren wurde in Abschnitt 3.4.3 noch eine Methode zur Reduktion der Ausführungshistorie vorgeschlagen. Der gleiche Algorithmus kann prinzipiell auch hier wieder benutzt werden. Jetzt ergeben sich allerdings weitergehende Auswirkungen bzw. Probleme, da nun wesentliche Informationen fehlen können. Es ist möglich, daß eine Aktivität innerhalb einer Schleife in verschiedenen Iterationen von jeweils anderen Servern kontrolliert wird. Die Server sind nur in der Historie vermerkt. Aufgrund dieses Unterschiedes wird im folgenden noch einmal gesondert auf die mögliche Optimierung für den variablen Fall eingegangen.

Für WF-Graphen mit Schleifenstrukturen ist eine Optimierung denkbar. Das Verfahren basiert auf der Idee, nur so viel Historieninformation zu übertragen, daß die zukünftigen Arbeitsschritte wissen, woher sie ihre Datenelemente zu beziehen haben.

Insbesondere bei einer großen Anzahl von Iterationen mit wechselnden Servern in den einzelnen Schleifendurchläufen kann bei der Übertragung der Historie durch Verzicht auf viele Historieneinträge Kommunikationsvolumen gespart werden. Diese Vorteile kommen besonders zum Tragen, wenn ein bestimmter Server zunächst mehrere Iterationen durchführt, dann aber eine Migration notwendig wird. Noch größer ist der Einspareffekt für die Weitergabe der Historieninformation nach Abschluß einer Schleife.



Ausführungshistorie bei S3 (exemplarisch)	
5 ...	14 START(D, 2, S2), END(D, 2, S2)
6 START(L_S, 1, S1), END(L_S, 1, S1)	15 START(L_E, 2, S1), END(L_E, 2, S1)
7 START(A, 1, S1), END(A, 1, S1)	16 START(L_S, 3, S1), END(L_S, 2, S1)
8 START(B, 1, S1), END(B, 1, S1)	17 START(A, 3, S1), END(A, 3, S1)
9 START(D, 1, S2), END(D, 1, S2)	**18 START(B, 2, S2), END(B, 2, S2)**
10 START(L_E, 1, S1), END(L_E, 1, S1)	19 START(D, 3, S2), END(D, 3, S2)
11 START(L_S, 2, S1), END(L_S, 2, S1)	20 START(L_E, 3, S1), END(L_E, 3, S1)
12 START(A, 2, S1), END(A, 2, S1)	21 START(E, 1, S3)
13 START(C, 1, S2), END(C, 1, S2)	22 ...

Abbildung 4-4. WF-Instanz mit Schleife und darin enthaltener bedingter Verzweigung.

Bis jetzt ergeben sich noch keine wesentlichen Unterschiede zum Fall mit statischen Serverzuordnungen. Am Beispiel aus Abbildung 4-4 wird noch einmal erkennbar, daß es nicht genügt, die Informationen der letzten Schleifeniteration weiterzugeben. Der Historieneintrag zur Ausführung von Aktivität C in Zeile 13 muß zusätzlich zu den der letzten Iteration entsprechenden Zeilen 16 bis 20 weitergegeben werden, da ansonsten der optionale Lesezugriff von Aktivität E nicht versorgt werden kann. Zum anderen ist erkennbar, daß selbst für den Fall, daß alle zwingend benötigten Informationen weitergegeben werden, möglicherweise interessante Informationen verloren gehen (nämlich die Zeile 8). Die Aktivität B wurde einmal auf dem Server S1, ein anderes mal (dritte Schleifeniteration, zweite Ausführung von B) auf Server S2 ausgeführt.

Gäbe es im gesamten Workflow lediglich obligate Parameter und kein Schreiben in bedingten Verzweigungen, müßte immer nur maximal Historieninformation aus der vorigen Schleifeniteration übertragen werden. Die notwendige Information über die Datenversorgung der Folgeaktivitäten des

Schleifenendknotens ist gewährleistet, da ein im Schleifenkörper enthaltener Schreiber dann in jeder Iteration ausgeführt werden muß.

Bei Verwendung bedingter Verzweigungen können jedoch auch optional gelesene Parameter auftreten. Innerhalb einer Schleife ist es durchaus denkbar, daß in unterschiedlichen Iterationen verschiedene Teilzweige ausgeführt werden. Deshalb kann es auch Datenelemente geben, die in einer Iteration nur dann beschrieben werden, wenn ein entsprechender Zweig durchlaufen wurde. Würde bei der Historienübertragung nur die letzte Schleifeniteration berücksichtigt, ginge die Information über die Ausführung dieser Schreiberaktivität verloren. Dadurch könnte bei der Migration der Datenelementeintrag nicht übertragen werden, weil er als nicht existent gelten würde. Ein an späterer Stelle optional lesender Parameter wäre dann nicht versorgt, obwohl ein passender Wert vorhanden ist!

Um Abhilfe zu schaffen, müssen für alle Aktivitäten, die irgendwann innerhalb der Schleife durchgeführt wurden, die jeweils aktuellsten Einträge weitergegeben werden. Der Server des Schleifenendknoten könnte jeweils beim Abschluß einer Schleifeniteration die dafür notwendigen Berechnungen durchführen.

Leider wird die beschriebene Optimierung unter Umständen teuer erkauft. Durch fehlende WF-Kontrolldaten früherer Schleifeniterationen wird in einigen Fällen die Möglichkeit zum kostenoptimierten Bezug von Datenelementen verschenkt. Man nehme hierzu den einfachen Fall, daß ein sehr großes Datenelement d_1 von einer Aktivität der Verzweigung innerhalb der Schleife gelesen wird (hier von Knoten B). Von einer E folgenden Aktivität R auf S3 soll ein Lesezugriff auf d_1 stattfinden. Zuerst fand die Ausführung von B auf dem für S3 sehr günstig erreichbaren Server S1 statt, in einer späteren Iteration auf dem in Bezug auf die Kommunikation teureren Server S2. Soll nun eine Migration erfolgen, würde S3 damit die Information fehlen, daß es für das Datenelement d_1 eine wesentlich günstigere Bezugsquelle als in diesem Beispiel den Server S2 gibt.

Es bleibt also zweifelhaft, ob sich der Aufwand für diese Optimierung wirklich lohnt. Da nicht bekannt ist, auf welchen Servern zukünftige Arbeitsschritte ausgeführt werden, macht ein noch aufwendigeres Reduktionsverfahren für die Historie aus Schleifen, bei dem auch noch potentielle Kommunikationskosten berücksichtigt werden, ebenfalls keinen Sinn. Schließlich könnte ein Server für bestimmte andere Server sehr günstig zu erreichen sein, für den Rest der Server hingegen unangemessen teuer. Somit könnten in beiden Fällen genau die falschen Historieneinträge aufgrund der „Optimierung“ weggelassen worden sein[15].

Darüber hinaus wäre mit der „Schleifen-Optimierung“ das ursprünglich definierte Kriterium der Vollständigkeit der Ausführungshistorie verletzt.

Denkbar wäre es, die beschriebene Schleifen-Optimierung bei sehr großen Workflows, die sich über viele Server hinziehen und die Schleifen mit voraussichtlich vielen Iterationen enthalten, einzusetzen. Um sicherzustellen, daß mit der Optimierung keine Nachteile erkauft werden, muß zusätzlich noch ein bezüglich der Kommunikationskosten relativ homogenes Netzwerk vorausgesetzt werden. Man könnte dem Modellierer die Entscheidung überlassen, ob die Optimierung bei einer bestimmten Vorlage genutzt werden soll oder nicht. Jedoch müßten ihm damit zunächst die internen Abläufe des WfMS klar gemacht werden, da es sich doch um ein sehr spezielles Feintuning handelt. Eigentlich sollte sich der Modellierer nicht mit solchen technischen Detailproblemen auseinandersetzen müssen, da er ja vor allem für eine adäquate Umsetzung der realen Abläufe in eine Prozeßvorlage zuständig ist. Eine andere Möglichkeit wäre daher, die Erfahrungswerte erst im Laufe der Ausführung von Instanzen dieser Vorlage zu sammeln und daraufhin automatisch im Nachhinein eine Abschätzung zu machen, ob sich die Schleifen-Optimierung amortisiert.

[15] Gemeint ist Entscheidung zwischen Zeile 8 und Zeile 18. Ggf. müßten doch alle Historieneinträge des Schleifenkörpers, zumindest aber diejenigen von Aktivitäten aus bedingten Verzweigungen, weitergegeben werden.

4.4 Übertragen von Datenelementen

Für die Übertragung der Datenelemente muß, wenn keine der einfachen Methoden verwendet wird, anders verfahren werden als bei ausschließlich statischen Serverzuordnungen. Es ist nicht mehr notwendigerweise bekannt, welche Server nachfolgenden Aktivitäten zugeordnet sein werden. Einige Varianten zur optimierten Lösung im statischen Fall bauen jedoch auf dieser Voraussetzung auf.

4.4.1 Bewertung der Verfahren im Fall variabler Serverzuordnungen

Wenn bei jeder Migration die kompletten Datenelemente verschickt werden, kann wie bei statischen Serverzuordnungen vorgegangen werden. Für die Variante, bei der nur die aktuellsten Werte aller Datenelemente verschickt werden, gilt dies ebenfalls. Beide Verfahren funktionieren unabhängig von den tatsächlichen Servern der Folgeaktivitäten. Auch die „On Demand"-Lösung kann unverändert übernommen werden, da hier zwar optimiert wird, jedoch nur Daten für die gerade aktivierte Aktivität geholt werden. Zu diesem Zeitpunkt ist deren Server immer schon bekannt.

Die abweichende Behandlung für kleine Datenelemente kann ebenfalls unverändert übernommen werden. Sie baut auf dem Verfahren zur Holen-Lösung für Historieneinträge auf. Daß dieses Verfahren auch bei variablen Serverzuordnungen anwendbar ist, wurde in Abschnitt 4.3 gezeigt. Zusätzlich wird noch der Algorithmus zur Bestimmung der aus der Sicht des Migrationszielknotens aktuellsten Version eines Datenelements benötigt. Auch er läßt sich mit den zu diesem Zeitpunkt bereits vorhandenen Informationen durchführen.

Allerdings kann die Schicken-Lösung (siehe Abschnitt 3.6.2) nicht mehr verwendet werden. Bei diesem Verfahren müssen die von der Migrationsquellaktivität und ihren Vorgängern auf dem gleichen Server geschriebenen Daten unmittelbar an alle nachfolgenden Leser des jeweiligen Datenelementeintrags geschickt werden. Nur dann ist die Datenversorgung auch sichergestellt. Da nicht von allen zukünftigen Aktivitäten, und damit auch Lesern, der Server bekannt ist, können diese nicht versorgt werden. Die folgende Abbildung 4-5 verdeutlicht die Problematik.

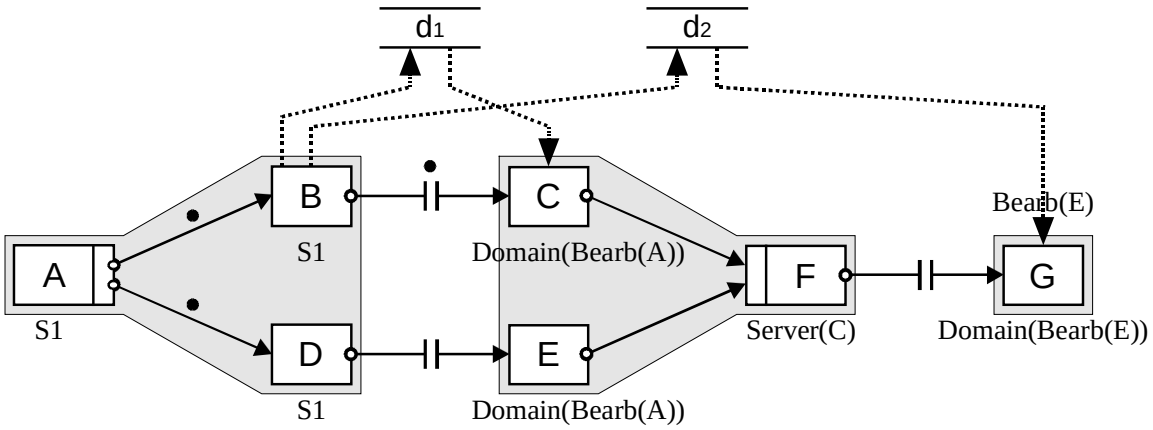


Abbildung 4-5: Übertragen der von der folgenden Aktivität benötigten Datenelemente.

Gemäß dem Schicken-Verfahren müssen bei der Migration $M_{B,C}$ alle der Quellaktivität B vorangehenden Schreiber der gleichen Partition (einschließlich B) betrachtet werden. Alle nachfolgenden Leser, d.h. im Beispiel C und G müssen mit den aktuellen Werten für d_1 bzw. d_2 versorgt werden. Da zum Zeitpunkt der Migration der Bearbeiter der Aktivität E noch nicht bekannt ist, kann der Serverzuordnungsausdruck von G noch nicht berechnet werden. Der Server ist also noch unbekannt, und dementsprechend kann nichts verschickt werden.

4.4.2 Holen-Verfahren

Die Holen-Lösung aus Abschnitt 3.6.3 kann im Prinzip ohne Veränderungen übernommen werden. Hierbei werden die benötigten Daten für jede Aktivität migriert, bevor diese aktiviert wird. Alle benötigte Zustandsinformation und natürlich auch der Server dieser Aktivität sind damit bekannt. Im Beispiel aus Abbildung 4-5 wird bei diesem Verfahren, im Anschluß an die Migration M_1 und vor der Ausführung von C, das Datenelement d_1 an den zu diesem Zeitpunkt bekannten Server (mit ServZuordn = Domain(Bearb(A))) übertragen. Das Datenelement d_2 wird vor der Ausführung von G migriert.

Der für statische Serverzuordnungen angegebene Algorithmus 3-6 muß nur an einer Stelle modifiziert werden. Aufgrund der variablen Serverzuordnung muß bei der Bestimmung der den benötigten Datenelementeintrag lesenden Aktivitäten jetzt auch immer deren Iterationsnummer berücksichtigt werden. Bei der Bestimmung des ausführenden Servers muß diese angegeben werden, da die Aktivität in verschiedenen Iterationen möglicherweise auf unterschiedlichen Servern ausgeführt wurden. Dies wirkt sich auf die Berechnung des günstigsten Servers aus. Falls ein Datenelement d von einer Aktivität in einer Schleife gelesen wird und d zwischenzeitlich nicht mehr beschrieben wurde, kommen in manchen Fällen noch mehr Server als bei statischen Zuordnungen[16] als Bezugsquelle für d in Frage (vgl. Abbildung 4-4).

Hier nun der zugehörige Algorithmus für variable Serverzuordnungen:

Algorithmus BestimmeBenoetigteDatenelemente

input
n ∈ N: Die Aktivität, die mit Daten versorgt werden soll

output
Result: {d ∈ D, n ∈ N, i ∈ Integer, s ∈ S}: Menge von Tupeln, die angibt, welcher Datenelementeintrag (eindeutig bestimmt durch n, i) des Datenelements d am besten von welchem Server s bezogen werden kann

begin
Result = ∅; // der Rückgabewert
S_n = Server(n);
// Berechnung der Datenelemente D, die von der Aktivität n gelesen werden
D = { d | n ∈ readers(d) };
for each d ∈ D **do**
// Bestimmung des passenden Schreibers für d
(n_{Write}, W_{rite}) = lastWriter(n, d, H);
if not (<d, n_{Write}, i_{Write}> ist schon auf S_n vorhanden) **then**
// Berechnung weiterer Bezugsquellen für benötigten Datenelementeintrag <d, n_{Write}, i_{Write}>
(n_1, i_1)..(n_n, i_n) ={a | a ∈ succ_i(n_{Write}, i_{Write}, H) ∧ a∈ readers(d) ∧ (state(a)=COMPLETED)};
S = guenstigsterServer(S_{Target}, Server (n_{Write}, i_{Write}), Server (n_1, i_1).. Server (n_n, i_n));
Result = Result ∪ (d, n_{Write}, i_{Write}, S);

end

Algorithmus 4-1: Bestimmung der zu übertragenden Datenelemente.

4.4.3 Holen der Datenelemente für alle Aktivitäten der Zielpartition

Wie in Kapitel 3 wird an dieser Stelle auf Optimierungspotentiale eingegangen, ohne diese allerdings weiter zu verfolgen oder spezielle Algorithmen anzugeben.

[16] Bei statischen Serverzuordnungen wird eine Aktivität immer vom gleichen Server kontrolliert.

Zu den in Kapitel 3 angegebenen optimierten Varianten der Holen-Lösung ergeben sich Unterschiede. Im Rahmen der Holen-Lösung wird dabei versucht, für alle noch folgenden Aktivitäten des Zielknotens, die ebenfalls auf dem Migrationszielserver ausgeführt werden, die Daten bereitzustellen, soweit sie schon existieren. Dieser Anspruch ist hier nicht mehr aufrecht zu erhalten, da bei variablen Serverzuordnungen im allgemeinen gar nicht alle Aktivitäten bekannt sind, die vom Migrationszielserver kontrolliert werden. Die in Kapitel 3 skizzierte Lösung kann von der Idee her übernommen werden, allerdings ist jetzt unter Umständen erst ein Teil der Aktivitäten bekannt, die der Zielaktivität folgen und vom gleichen Server kontrolliert werden. Nur diese können bereits mit Daten versorgt werden.

Im wesentlichen sollte daher jetzt so verfahren werden: Es werden nur noch für die der Zielaktivität folgenden Aktivitäten der gleichen Partition Daten bereitgestellt. Sollte nach der Partition eine Migration stattfinden, kann diese regulär abgewickelt werden. Ergibt sich durch den Serverzuordnungsausdruck aber, daß gar keine Migration notwendig wird, muß jetzt bezogen auf die Datenelemente dennoch wieder wie bei einer Migration verfahren werden. Aktivitäten dieser neuen Partition könnten ja noch Datenelemente fehlen, da bisher nicht bekannt war, daß sie ebenfalls von diesem Server kontrolliert werden. Im Vergleich zu statischen Serverzuordnungen entstehen so meist mehr Kommunikationen, das Datenvolumen hingegen bleibt gleich.

Abschließend wird ein Beispielszenario für eine Migration angegeben: Zunächst wird auf dem Zielserver die Partition P der Zielaktivität berechnet. Die in dieser Partition enthaltenen Aktivitäten $p \in P$ müssen über Kontroll-, Sync- oder Schleifenkanten folgen und vom gleichen Server kontrolliert werden. Ob zwei Aktivitäten vom gleichen Server kontrolliert werden, wurde zur Modellierungszeit vorberechnet (Serverklassen) und in der WF-Vorlage gespeichert. Für den Beispielgraph der Abbildung 4-5 bedeutet dies, daß die Aktivitäten C und F zurückgeliefert werden. Aktivitäten des Zielservers, welche der Zielaktivität in der Graphstruktur vorangehen, dürfen bei der Übertragung der Datenelemente nicht berücksichtigt werden. (hier: Aktivität E). Ihre Datenversorgung wird durch eine andere im Kontrollfluß vor diesen Aktivitäten liegende Migration gesichert. Die Daten können nämlich immer nur in Richtung des Kontrollflusses fließen. Auch wenn die Aktivität G eventuell vom gleichen Server wie F kontrolliert werden wird, kann sie bei der aktuellen Migration M_1 nicht berücksichtigt werden.

Die Bestimmung der optimalen Bezugsquellen für die Datenelemente wird jetzt wie in Algorithmus 4-1 durchgeführt. Gegenüber statischen Serverzuordnungen stehen oftmals mehr mögliche Bezugsquellen zu Auswahl.

4.5 Einfluß der Optimierung auf die Kostenberechnung zur Modellierungszeit

In [BD00] wird ein Verfahren vorgestellt, bei dem eine bezüglich der zu erwartenden Kommunikationskosten optimale Verteilung von Aktivitäten auf ausführende WF-Server berechnet wird. In einer ersten Phase werden dabei noch keine Migrationskosten berücksichtigt, sondern lediglich die Vorgaben in Bezug auf die Verteilung der Benutzer und im Rahmen der Ausführung von Aktivitäten anfallende Kosten. Um unrentable Migrationen zu eliminieren, werden in einer zweiten Phase die Migrationskosten beachtet und nach einem bestimmten Verfahren Aktivitäten zusammengefaßt, falls die Gesamtkosten dadurch reduziert werden. Allerdings wird von konstanten Migrationskosten ausgegangen, die aufgrund der hier vorgestellten Optimierungen zu hoch sind. Wenn eine optimierte Lösung zum Übertragen von Datenelementen eingesetzt wird, kann lediglich ein Teil der Migrationskosten als konstant angenommen werden. Das Übertragen eines großen Datenelements findet nur statt, falls dies notwendig ist, und verursacht somit nicht konstante Kosten. In der Berechnung der Kommunikationskosten ist dies zu berücksichtigen.

Für die Berechnung der Migrationskosten wird eine Matrix aus Werten benötigt, die angibt, mit welcher Wahrscheinlichkeit bei einer Migration $M_{Q,Z}$ eine Migration des Datenelements e von Server S_1 nach Server S_2 stattfindet: $MigrProb^{(e)}_{Q,Z}(S_1, S_2)$. Der Zeitpunkt, wann diese Migration des Datenelements stattfindet, spielt für die Berechnung keine Rolle.

4.5.1 Allgemeine Betrachtung

Eine Allgemeine Betrachtung der Ausführungswahrscheinlickeiten für schreibende und lesende Aktivitäten eines Datenelements ist wegen bedingten Verzweigungen und Schleifen sehr kompliziert (vgl. Abbildung 4-6).

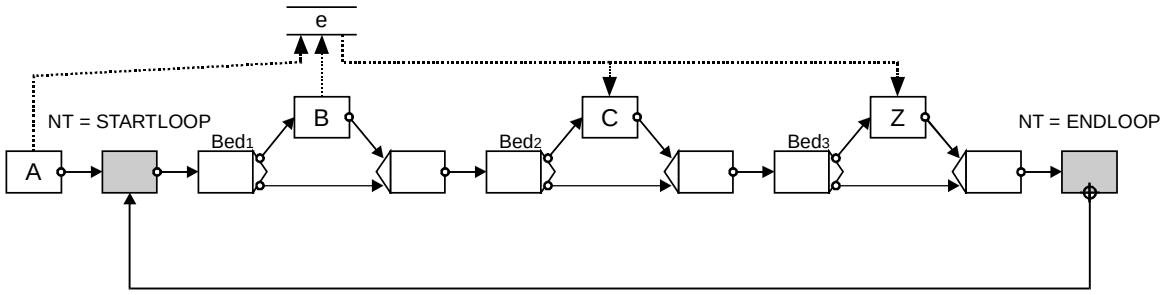


Abbildung 4-6: Migrationen von Datenelementen und bedingte Verzweigungen.

Viele Abhängigkeiten können nicht abgeschätzt werden. Im Beispiel soll eine Migration zu Z angenommen werden. Woher das Datenelement e bezogen werden kann, hängt von den Bedingungen Bed_1 bis Bed_3 und der Schleifeniteration ab, welche ihrerseits von Datenelementen abhängen.

Von den möglichen Bezugsquellen sollen zuerst die Schreiber betrachtet werden:

- Falls $Bed_1 = Bed_3$, so wird immer das von B geschriebene Element bezogen.
- Falls $Bed_1 = \neg Bed_3$, so wird das von A geschriebene Element bezogen.
- Falls Bed_1 unabhängig Bed_3 oder weitere Faktoren wie z.B. die Iterationsnummer die Bedingung beeinflussen, kommt als Schreiber die Aktivität B der gleichen Iteration oder einer Vorgängeriteration oder A in Frage.

Zusätzlich ist noch ein Bezug vom Server von C möglich, falls die Aktivität C ausgeführt wird und das Datenelement e damit gelesen hat. So spielt auch die Bedingung Bed_2 noch eine Rolle.

Die Angabe der (unabhängigen) Wahrscheinlichkeiten für die Auswahl der Teilzweige führt nicht weiter, da damit keine Abhängigkeiten berücksichtigt sind.

Diese komplexen Betrachtungen führen zu einer sehr aufwendigen Berechnung, hinter der sehr viele Annahmen stehen. Sind nicht alle diese Abhängigkeiten bekannt, wird das Berechnungsergebnis sehr ungenau.

4.5.2 Auswertung vorhandener Ausführungsinformation

Eine Alternative zur theoretischen Berechnung der tatsächlichen Migrationskosten ist die Auswertung von Informationen aus dem Logfile einer WF-Instanz. Dabei können auch die Ausführungsdaten von einem zentralen WfMS verwendet werden. Entscheidend ist die Gewinnung von Information über Leser- und Schreiberaktivitäten eines Datenelements.

Zur Bestimmung von $MigrProb^{(e)}_{Q,Z}(S_1, S_2)$ muß jetzt für alle Migrationen von Aktivität Q zu Z wie im folgenden beschrieben vorgegangen werden. Falls die Migration von Q nach Z in einer Schleife stattfindet, muß die Berechnung für jede Schleifeniteration durchgeführt werden. Die Vorgehensweise entspricht derjenigen bei der Bestimmung der günstigsten Bezugsquelle eines Datenelements vor der Aktivierung einer Aktivität (vgl. 4.4.2).

- Aus der Ausführungshistorie die letzte Schreiberaktivität Q_0 ermitteln.
- Die Leseraktivitäten $Q_1..Q_n$ zwischen Q_0 und Z ermitteln.
- Die Server $S(Q_i)$ für $Q_0..Q_n$ und $S(Z)$ ermitteln. Dabei werden die Serverzuordnungen der Aktivitäten verwendet, variable Serverzuordnungen können ebenfalls ausgewertet werden, indem die Bearbeiterinformation der Ausführungshistorie eingesetzt wird.

- Anhand der Kostenmatrix $K(S_1, S_2)$ das $Q' \in \{Q_0..Q_n\}$ ermitteln, von dessen Server Daten geholt werden, d.h. $\forall Q_i$: $K(S(Q'), S(Z)) \leq K(S(Q_i), S(Z))$.
- Die beiden Zähler erhöhen:

  $MigrProb^{(e)}_{Q,Z}(S(Q'), S(Z)) = MigrProb^{(e)}_{Q,Z}(S(Q'), S(Z)) + 1;$

  $AnzMigr = AnzMigr + 1;$

Abschließend muß $MigrProb^{(e)}_{Q,Z}(S_1, S_2)$ noch normiert werden:

$\forall S_1,S_2$: $MigrProb^{(e)}_{Q,Z}(S_1, S_2) = MigrProb^{(e)}_{Q,Z}(S_1, S_2) / AnzMigr;$

Alle weiteren Datenelemente werden genauso behandelt. Um aussagekräftige Werte zu erhalten, müssen unbedingt viele Instanzen berücksichtigt werden.

4.6 Zusammenführung paralleler Ausführungszweige

Bevor eine Migration durchgeführt wird, muß zunächst ihr Zielserver bestimmt werden. Nur wenn dieser unterschiedlich vom Ausgangsserver ist muß eine Migration durchgeführt werden.

Bei ausschließlich statischen Serverzuordnungen ist der Server für jede Aktivität schon festgelegt und kann direkt als Migrationszielserver verwendet werden.

Bei variablen Serverzuordnungen muß zuvor der Serverzuordnungsausdruck der Folgeaktivität ausgewertet werden. Läßt sich der Ausdruck vom Migrationsquellserver auswerten, ergibt sich daraus der Zielserver. Dies muß jedoch nicht immer der Fall sein, wie der folgenden Abbildung 4-7 zu entnehmen ist (vgl. [BD00]).

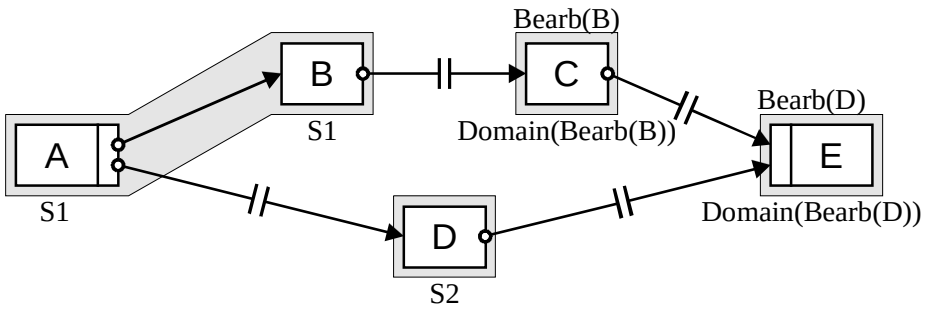


Abbildung 4-7: Bestimmung des Migrationszielservers für den Joinknoten.

An Synchronisationspunkten, d.h. Knoten mit mehr als einer Eingangskante, kann sich der Serverzuordnungsausdruck auf beliebige Vorgängeraktivitäten beziehen. Darin enthalten sind natürlich auch Aktivitäten aus einem anderen, noch nicht eingetroffenen parallelen Zweig, in diesem Beispiel der untere Zweig (mit Aktivität D). Es muß also in manchen Fällen mit der Durchführung der Migration gewartet werden, bis die benötigte Information verfügbar ist. Wird die Aktivität C vor D beendet, kann die Migration nach E erst stattfinden, wenn auch die Aktivität D beendet ist. Außerdem muß ein Ort festgelegt werden, an dem diese Information bereitgestellt wird.

Grundsätzlich besteht natürlich auch die Option, den Zielserver per Broadcast allen anderen Servern im System mitzuteilen, sobald der Zuordnungsausdruck ausgewertet werden kann. Es ist offensichtlich, daß dies sicherlich nicht die optimale Lösung in Bezug auf die zu übertragende Kommunikationsmenge ist. Ein spezieller Server nur für diese Aufgabe wäre möglich, er stellt dann jedoch einen „Single point of failure“ für alle Workflows im gesamten System dar. Die folgenden weiteren Möglichkeiten bieten sich an: Übermittlung der Information an eine zentrale Stelle, wie z.B. den Startserver dieser WF-Instanz. Allerdings könnte der Workflow inzwischen schon so weit fortgeschritten sein, daß dieser Server in einem weit entfernten Teilnetz liegt. Besser wäre es daher, für die

Wahl des Koordinators die Struktur des Workflows auszunutzen und genau den Server des passenden, d.h. naheliegendsten Splitknotens zu verwenden. Dies muß bei verschachtelten parallelen Verzweigungen nicht unbedingt der dem Zielknoten unmittelbar vorangehende Splitknoten sein. Im Beispiel aus Abbildung 4-7 übernimmt der Server des „kleinsten gemeinsamen Splitknotens“ der Aktivitäten D und C, der Server S1 von Knoten A, diese Aufgabe.

Zur Bestimmung des Migrationszielservers bei einer Migration von der Quellaktivität SourceAct zur Zielaktivität TargetAct bietet sich die im folgenden beschriebene Vorgehensweise an.

Zunächst muß bestimmt werden, ob es sich bei TargetAct um einen Knoten mit mehr als einer eingehenden Kontroll- oder Synchronisationskante handelt. Seien Preds die direkten Vorgänger von TargetAct. Es muß geprüft werden, ob der Zuordnungsausdruck von TargetAct eine Aktivität x referenziert, die nicht in den Vorgängeraktivitäten von jedem $p \in$ Preds (incl. p) enthalten ist. Dann wäre der Ausdruck nämlich nicht von allen Quellservern auswertbar. Eine äquivalente Formulierung hierfür lautet: Es existiert ein direkter Vorgänger p von TargetAct, bei dem x nicht in der Vorgängermenge von p (incl. p) enthalten ist. Die Aktivitäten, die den Ausdruck auswerten können, sind daher verantwortlich, den anderen Aktivitäten die Information bekanntzugeben. Als Mittler wird die am nächsten bei TargetAct liegende gemeinsame Splitaktivität von allen unmittelbaren Vorgängern von TargetAct herangezogen. Da die so bestimmte Splitaktivität ein gemeinsamer Vorgänger ist, der somit schon ausgeführt wurde, ist auch in jedem Fall dessen zugehöriger Server bekannt.

Es ist durchaus möglich, daß mehrere Aktivitäten den Ausdruck auswerten können. Dies stellt kein Problem dar, da das Ergebnis identisch sein muß. Die nach Information fragenden Aktivitäten können darüber hinaus immer bereits versorgt werden, wenn die Information zum ersten mal eingetroffen ist.

Wichtig ist, den Iterationszähler für TargetAct immer mit zu übertragen und beim Koordinator abzuspeichern. Nur so ist gewährleistet, daß im Falle von Schleifen bei Anfragen von Knoten, die den Serverzuordnungsausdruck von TargetAct nicht auswerten können, keine Werte aus einer vorigen Iteration zurückgeliefert werden.

Vor dem Algorithmus soll noch ein (konstruierter) Fall gezeigt werden, bei dem auch Synchronisationskanten berücksichtigt werden müssen. In Abbildung 4-8 muß die Aktivität B mit der Migration zu C auf die benötigte Information zum Migrationsziel warten. Die Aktivität D kann erst aktiviert werden, nachdem die von E eingehende Sync-Kante signalisiert wurde. Bei der Migration $M_{D,C}$ stellt der Server S2 fest, daß der Server von B den Zuordnungsausdruck seiner Zielaktivität nicht auswerten kann. S2 muß diese Information für den Server von B verfügbar machen. Er teilt die Information nach Beenden der Aktivität D an S1, den Server des gemeinsamen Splitknotens A, mit. Damit können die Server, die mit ihren Migrationen zu C warten, benachrichtigt werden (hier: S1). Die Migrationen $M_{D,F}$ und $M_{E,F}$ verbleiben im Wartezustand, bis Aktivität C beendet und der tatsächliche Server von F bekanntgegeben wurde. Für die Ausführung des Workflow an sich stellt dies kein Problem dar, da an einem Synchronisationskanten ohnehin gewartet wird, bis alle Eingangskanten signalisiert wurden.

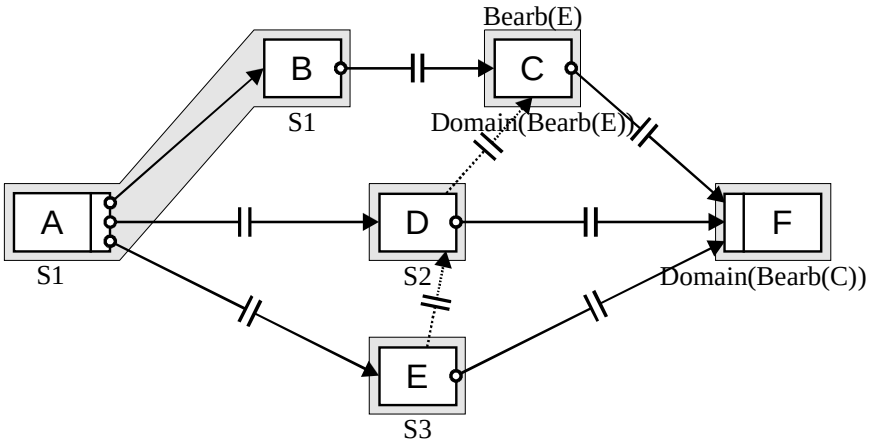


Abbildung 4-8: Verzögerte Migrationen aufgrund von Abhängigkeiten.

Das Warten auf die Benachrichtigung muß asynchron geschehen. Bei einem Knoten mit mehreren Ausgangskanten kann es vorkommen, daß eine Migration warten muß, der andere Zweig jedoch schon fortgeführt werden kann (vgl. Abbildung 4-9). Die Migration $M_{B,E}$ muß auf das Beenden des Knotens D und die anschließende Nachricht von S1 warten, währenddessen konnte schon die Migration $M_{B,C}$ durchgeführt und die Aktivität C gestartet werden.

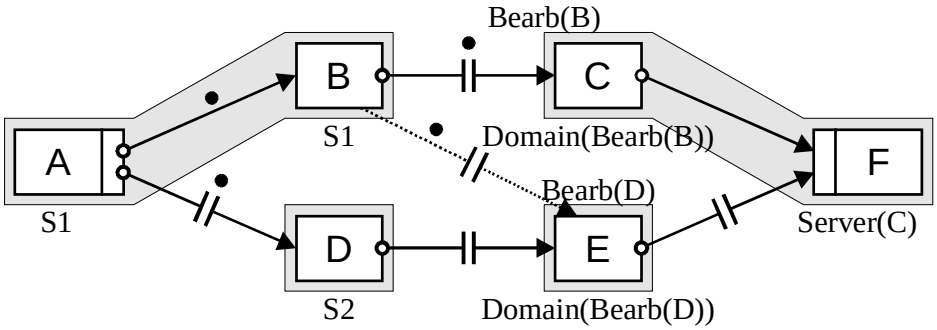


Abbildung 4-9: Asynchrones Warten auf die Information über das Migrationsziel.

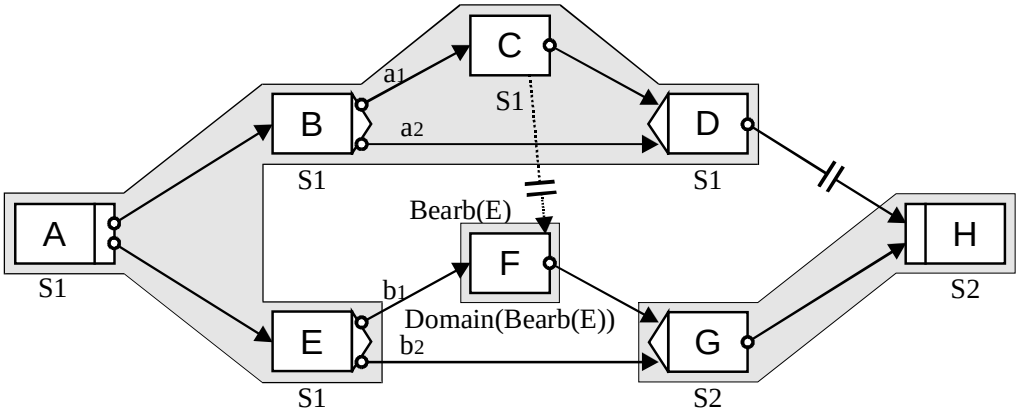


Abbildung 4-10: Unbekannter Migrationszielserver in bedingter Verzweigung.

In Verbindung mit bedingten Verzweigungen kann noch ein spezielles Problem auftreten. Wenn im Workflow aus Abbildung 4-10 die Alternative a1 oder a2 gewählt wird, muß die Sync-Kante C→F signalisiert werden und im Anschluß die Migration $M_{C,F}$ (bzw. $M_{B,F}$ bei Wahl von a2) stattfinden. Der Migrationszielserver ist noch unbekannt und muß daher bei S1, dem Server von A, erfragt werden. Bei der Migration $M_{E,F}$ (Alternative b1) stellt der Server von E die benötigte Information bereit.

Wird jedoch die Alternative b2 gewählt, so wird diese Information nie bereitgestellt, und damit verbleibt $M_{C,F}$ im Wartezustand. Für die Beibehaltung der korrekten Ausführungssemantik des Workflow ist dies nicht problematisch, da die Sync-Kante C→F im folgenden ohnehin als nicht existent gilt. Nach den Ausführungsregeln kann der Knoten G sofort aktiviert werden[17], da bei einem OR-Joinknoten dafür nur eine Kontrollkante im Zustand TRUE_SIGNALED notwendig ist. Die Migration $M_{C,F}$ kann vernachlässigt werden, besser wäre es aber man meldet dem Server von A, daß diese Migration den Server „NULL“ als Ziel hat. Damit können die Nachrichten wie bisher weitergegeben und umsonst wartende Migrationen abgebrochen werden. Bei der „Dead-Path-

[17] Der Knoten G steht somit beispielsweise auch in keiner garantierten Nachfolgerbeziehung zum Knoten B.

Elimination“[18] muß daher genauso wie bei einer regulären Migration überprüft werden, ob andere Server ein Migrationsziel nicht bestimmen können. Als Ziel wird dann jeweils „NULL“ festgelegt.

Algorithmus zur Bestimmung des Zielservers bei Migration von SourceAct zu TargetAct

```
input
   TargetAct ∈ N: Die Aktivität, zu der migriert werden soll
begin
   // Den Serverzuordnungsausdruck auswerten
   S_Target = ServZuordn(TargetAct);
   if S_Target ≠ NULL then  // S_Target konnte berechnet werden
      Preds = pred(TargetAct); // unmittelbare Vorgänger von TargetAct
      x = von ServZuordn(TargetAct) referenzierte Aktivität;
      // Prüfen, ob ein anderer Quellserver existiert, der den Ausdruck nicht auswerten kann
      if ( ∃ p ∈ Preds | x ∉ pred*(p) ∪ p ) then
         SplitAct = correspondingSplitAct( Preds );
         Iteration = letzte bekannte Iterationsnummer von TargetAct + 1;
         Notify ( SplitAct, TargetAct, Iteration, S_Target); // den Wert bei SplitAct hinterlegen
   else
      // S_Target muß erfragt werden
      SplitAct = correspondingSplitAct( pred(TargetAct) );
      Iteration = letzte bekannte Iterationsnummer von TargetAct + 1;
      registerForNotify( SplitAct, TargetAct, Iteration ); // SplitAct mitteilen, auf was gewartet wird
      // Auf Wert von SplitAct warten. Dieser kann die Antwort erst dann schicken, wenn er die
      // Information besitzt. Es wartet nur der thread, der die Migration von SourceAct nach
      // TargetAct durchführt, andere Zweige können weiter laufen.
      wait until notified;
      S_Target = incomingNotifyMessage;
   if S_Target ≠ NULL then  // S_Target konnte berechnet werden oder ein gültiger Wert wurde erhalten
      Migrate(Instance) → S_Target; // Die eigentliche Migration durchführen
end
```

Algorithmus 4-2: Bestimmung des Zielservers bei der Migration.

4.7 Zusammenfassung

Die im Zusammenhang mit statischen Serverzuordnungen in Kapitel 3 entwickelten Verfahren lassen sich größtenteils auch im variablen Fall einsetzen. Es ergeben sich aber an einigen Stellen Detailprobleme aus der Tatsache, daß die Server von in der Zukunft liegenden Aktivitäten nicht immer bekannt sind.

Zur Übertragung von WF-Kontrolldaten kann das Holen-Protokoll aus Kapitel 3 unverändert übernommen werden. Allerdings ergeben sich bei der Bestimmung des Migrationszielservers Besonderheiten. Bei übersprungenen Aktivitäten einer bedingten Verzweigung können die Zuordnungsausdrücke möglicherweise nicht ausgewertet werden. Für Migrationen, die von einem OR-Splitknoten ausgehen, muß daher eine besondere Behandlung vorgenommen werden. Auch bei der Migration zu Knoten mit mehreren Eingangskanten kann der Zuordnungsausdruck des Zielknotens manchmal nicht sofort ausgewertet werden. In diesem Fall ist das aber zumindest einem der Migrationsquellserver möglich, der die Information dann mit einem geeigneten Verfahren weitergibt.

[18] So wird das Behandeln der nicht ausgewählten Zweige, d.h. das Markieren der Aktivitäten mit SKIPPED und der Kanten mit FALSE_SIGNALED, auch bezeichnet.

Für den Transfer der eigentlichen WF-Daten scheidet das Schicken-Protokoll aus, da hierbei vorausgesetzt wird, daß die Server aller Aktivitäten bekannt sind. Das Holen-Verfahren kann übernommen werden, lediglich bei einer möglichen Optimierung wird eine Änderung notwendig: Sollen gleich die Datenelemente für eine gesamte Partition geholt werden, müssen Datenelemente nachgefordert werden, wenn die Partition größer als ursprünglich angenommen sein sollte, weil zwei Serverklassen zusammenfallen.

Die Bewertung der Verfahren, die bei variablen Serverzuordnungen anwendbar sind, entspricht der aus Abschnitt 3.7.

5 Verteilte WF-Ausführung und dynamische WF-Änderungen

Neben der Skalierbarkeit ist die dynamische Adaptierbarkeit laufender Workflows eine unabdingbare Voraussetzung, um ein WfMS wirklich sinnvoll in der Praxis einsetzen zu können. Nur ein System, das Workflows effizient und flexibel unterstützt, kann in größerem Stil und für ein breites Spektrum an Abläufen eingesetzt werden. In der WF-Literatur wurden diese beiden Themenbereiche bisher lediglich isoliert voneinander betrachtet. Für die Realisierung eines WfMS, das beide Anforderungen erfüllt, muß eine gute Lösung für das Zusammenspiel aller Aspekte gefunden werden.

In einem verteilten WfMS, das dynamische Änderungen des Ausführungsgraphen einer WF-Instanz zuläßt, werden zusätzliche Maßnahmen nötig, um die Instanzgraphen aller an der Ausführung beteiligten Server auf dem aktuellen Stand zu halten. Zwar sind dynamische Änderungen im Verhältnis zur regulären WF-Ausführung nur relativ selten nötig, dennoch sollte durch eine dynamische Änderung die globale Ausführung der Instanz möglichst wenig beeinträchtigt werden, und es soll möglichst wenig zusätzliches Kommunikationsvolumen erzeugt werden. Im folgenden Kapitel werden Verfahren entwickelt, die diesen Anforderungen genügen.

5.1 Dynamische WF-Änderungen in ADEPT$_{flex}$

In diesem Abschnitt fassen wir informell die wichtigsten Eigenschaften des ADEPT$_{flex}$-Modells zusammen (für ausführliche Darstellung siehe [RD98], [Rei00]). Das ADEPT$_{flex}$-Modell ist ein adaptives WF-Metamodell, das Aspekte wie robustes und zuverlässiges Ausführungsverhalten mit der Möglichkeit, *Ad-Hoc-Abweichungen* vom vorgeplanten Ablauf vorzunehmen, verbindet. Hierzu wird eine vollständige und minimale Menge an Änderungsoperationen angeboten, mit der sich der Ausführungsgraph einer WF-Instanz zur Laufzeit modifizieren läßt. Die Unterstützung solcher Ad-Hoc-Abweichungen ist für den praktischen Einsatz von WfMS von großer Bedeutung, wenn ein hohes Maß an Flexibilität erzielt werden soll. Beispielsweise kann es notwendig werden, an einer bestimmten Stelle im Ablauf einen zusätzlichen Arbeitsschritt einzufügen oder die Reihenfolge von Schritten zu vertauschen. Insbesondere bei umfangreichen und komplexen Prozessen ist es - allein aus kombinatorischen Gründen - nicht möglich bzw. sinnvoll, alle Ausführungsvarianten und Ausnahmen eines Prozesses in seiner Vorlage abzubilden [RHD98].

Das Hauptaugenmerk bei der Konzeption der ADEPT$_{flex}$-Änderungsoperationen betrifft Korrektheits- und Konsistenzaspekte. Dies bedeutet, daß der WF-Graph auch nach der Änderung syntaktisch korrekt sein muß und einen konsistenten Zustand aufweist. Er darf keine zyklischen Abhängigkeiten enthalten und die Datenversorgung aller Eingabeparameter muß weiterhin gewährleistet sein, so daß es zu keinen Verklemmungen oder anderen Laufzeitfehlern (z.B. Aufruf von Aktivitätenprogrammen mit fehlenden Eingabedaten) kommt. Vor der persistenten Durchführung einer Änderung müssen diese Kriterien überprüft und die Änderung gegebenenfalls zurückgewiesen werden, wenn sie mit dem aktuellen Zustand der Prozeßinstanz nicht vereinbar ist. Im Fall unversorgter Eingabeparameter gibt es verschiedene Möglichkeiten, fehlende Daten auf anderen Wegen bereitzustellen (z.B. mittels sogenannter Nachforderungsdienste, vgl. [Rei00]).

Von ADEPT$_{flex}$ werden u.a. die folgenden Änderungsoperationen angeboten:

- Dynamisches Einfügen einer Aktivität oder eines Aktivitätenblocks zwischen 2 aufeinanderfolgenden Knoten, als neuer Zweig innerhalb einer bedingten Verzweigung oder parallel zu einem logischen Block (z.B. eine Sequenz oder eine Verzweigung).
- Dynamisches Löschen einer Aktivität.
- Überspringen von Aktivitäten mit oder ohne Nachholen der übersprungenen Schritte.
- Einfügen und Entfernen von Synchronisationskanten zwischen Aktivitäten in unterschiedlichen Zweigen einer parallelen Verzweigung.
- Dynamisches Zurücksetzen eines Bereichs der WF-Instanz (incl. dem Zurücknehmen von temporären Änderungen).

Eine formale Darstellung dieser und anderer Änderungsoperationen findet man in [Rei00]. Auf diesen Basisänderungsoperationen aufbauend können komplexe Änderungsoperationen auf einer semantisch höheren Ebene festgelegt werden, wie z.B. das dynamische Einfügen einer Aktivität oder eines Aktivitätenblocks zwischen zwei Mengen von Aktivitäten.

$ADEPT_{flex}$ definiert für jede Änderungsoperation formale Voraussetzungen für ihre Anwendung, sowohl bezogen auf die aktuelle Struktur als auch an den aktuellen Zustand des Ausführungsgraphen, die zugehörigen Graphersetzungsregeln zur Verwirklichung der Änderung, Mechanismen zur Erkennung auftretender Probleme sowie Strategien zur Behandlung dieser Probleme. Die Operationen selbst werden auf eine Menge *elementarer Änderungsprimitive* wie dem Einfügen oder Löschen von Knoten und Kanten abgebildet. Dadurch ist auch ihre Semantik formal festgelegt. Zu einer Änderungsoperation gehörende Änderungsprimitive und evtl. assoziierte Folgeoperationen werden in einer *Änderungstransaktion* geklammert. Eine solche Änderungstransaktion kann relativ lange dauern, da im Fall von auftretenden Problemen Rückfragen beim Benutzer notwendig werden können.

Bei der Durchführung einer dynamischen Änderung wird in der *Ausführungshistorie* der betreffenden WF-Instanz ein Eintrag vorgenommen, der auf einen korrespondierenden Eintrag der *Instanz-Änderungshistorie* verweist. In diese Änderungshistorie erfolgt ein Eintrag mit folgenden Informationen: Zeitstempel, Startbereich der Änderung, Art der Änderungsoperation, Änderungsprimitive und ggf. Folgeoperationen (vgl. [Wei97] und Tabelle 5-1 im folgenden Beispiel).

Eine WF-Modifikation kann im ADEPT-WfMS von einem an der Ausführung der Instanz beteiligten Akteur vorgenommen werden. Gegebenenfalls ist vorher eine Rücksprache mit anderen Teilnehmern des Systems erforderlich, dieser Aspekt wird hier allerdings nicht weiter berücksichtigt. Denkbar ist auch eine automatische Modifikation der Instanz durch einen Agenten nach dem Eintreten zuvor festgelegter Ereignisse.

Der Ablauf einer Änderung für den logisch zentralen Fall soll zunächst am Beispiel einer dynamischen Einfügeoperation veranschaulicht werden, danach folgt ein Beispiel für das dynamische Löschen einer Aktivität.

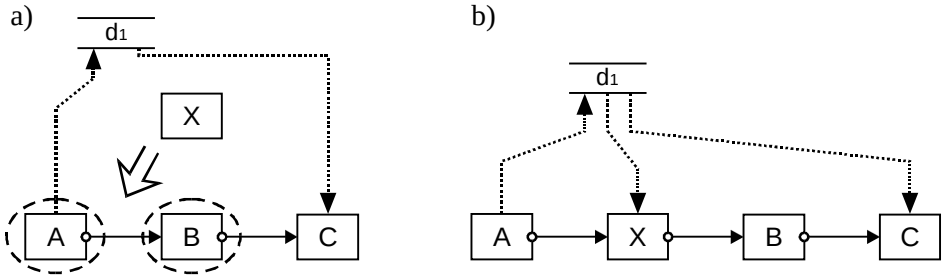


Abbildung 5-1: Einfache dynamische Einfügeoperation (*serialInsert*).

In einem noch nicht ausgeführten Teilbereich des Instanzgraphen soll zwischen den Aktivitäten A und B eine neue Aktivität X seriell eingefügt werden (vgl. Abbildung 5-1). Im vorliegenden Fall ergeben sich keine besonderen Probleme. Nach der Durchführung der Änderungsoperation ergibt sich der in b) dargestellte Graph. X liest vom Datenelement d_1, welches zuvor vom Knoten A beschrieben wird. Zu diesem Zweck mußte eine Datenflußkante hinzugefügt werden. Würde es keinen (vorangehenden) Schreiber von d_1 geben, hätte entweder ein Nachforderungsdienst zur Versorgung des Eingabeparameters vorgeschaltet werden müssen, oder aber X hätte an dieser Stelle im Graphen nicht eingefügt werden können (vgl. [Rei00]).

Im allgemeinen können Einfügeoperationen auch von komplexerer Natur sein, was durch das Beispiel aus Abbildung 5-2 deutlich wird. Hier wird eine Menge von Aktivitäten angegeben, zwischen denen X ausgeführt werden soll.

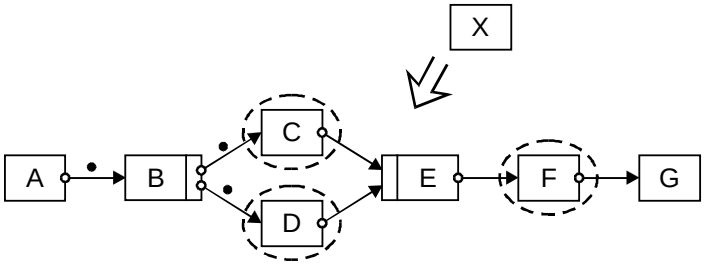
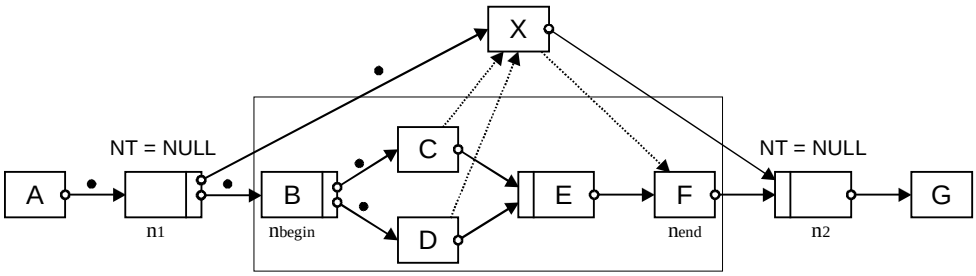


Abbildung 5-2: Dynamisches Einfügen zwischen zwei Mengen von Aktivitäten.

Die Aktivität X soll in dem dargestellten WF-Graphen vor der Aktivität F und nach den Aktivitäten C und D eingefügt werden. Dies wird dadurch realisiert, daß der Knoten X als paralleler Zweig zum minimalen Block, der die Aktivitäten F, C und D umschließt, eingefügt wird. Der berechnete minimale Block besteht aus den Knoten B, C, D, E und F. Zu beachten ist, daß in ihm auch Knoten enthalten sind, deren Zustand bereits COMPLETED ist (vgl. Abbildung 5-3). Allerdings darf ein neuer Knoten niemals vor einer solchen Aktivität eingefügt werden, sondern nur vor Knoten im Zustand NOT_ACTIVATED oder ACTIVATED.



Ausführungshistorie
1 START(A, 1), END(A, 1)
2 START(B, 1), END(B, 1)
3 START(C, 1)
4 DDM(1234)[19]

Abbildung 5-3: Instanzausführungsgraph nach dem Einfügen von X.

Tr-Time-stamp	Tr-ID	Gültig-keitsdauer	Start-bereich	Liste der Änderungsoperationen		
				Nr.	Art der Änderungsoperation	Änderungsprimitive
1	1234	permanent	{B}	1	insert(X, {C,D}, {F})	_DeleteEdge(A,B), _InsertNode(n1,AND-split), _InsertEdge(A,n1,CONTROL_E), ...

Tabelle 5-1: Zugehörige Änderungshistorie zu Abbildung 5-3.

Die Einhaltung der gewünschten Reihenfolge (Aktivieren von X nach C und D, Beenden vor F) wird durch das Einfügen der Sync-Kanten C→X, D→X und X→F erzielt.

Um wieder einen gültigen Status der Instanz zu erreichen, müssen die neu eingefügten Knoten und Kanten noch passend markiert werden. Der Verzweigungsknoten n_1 muß auf COMPLETED gesetzt werden, da seine Eingangskante markiert ist, seine ausgehenden Kontrollkanten müssen auf

[19] DDM bedeutet „Do Dynamic Modification“. Der Eintrag verweist auf den zugehörigen Eintrag der Änderungshistorie.

TRUE_SIGNALED gesetzt werden[20]. Die Synchronisationskanten zu X können noch nicht markiert werden, da C und D noch nicht beendet wurden. Somit kann auch X noch nicht aktiviert werden.

Der ADEPT$_{flex}$-Formalismus bietet noch verschiedene Möglichkeiten, die Struktur des Graphen nach der Anwendung der allgemeingültigen Änderungsoperationen und vor dem Anpassen der Markierungen zu vereinfachen (z.B. Verschmelzen von n_1 und B). Einige Regeln hierzu finden sich in [RD98], an dieser Stelle soll nicht weiter auf solche Optimierungen eingegangen werden.

Für die Darstellung der Änderungshistorie wird im folgenden nicht die ausführliche Variante wie in Tabelle 5-1 verwendet, sondern eine verkürzte Version:

Definition 5-1: Verkürzte Variante eines Änderungshistorieneintrags H_E.

Zur Bildung der verkürzten Variante werden aus dem vollständigen Eintrag H_E (vgl. Tabelle 5-1) lediglich die folgenden Attribute übernommen:
Tr-ID, zugehörige Änderungsoperation(en).

Abbildung 5-4 zeigt ein kurzes Beispiel zum dynamischen Löschen von WF-Aktivitäten. Das Problem beim Löschen besteht darin, daß es möglicherweise Schritte gibt, die von der zu löschenden Aktivität abhängig sind. ADEPT$_{flex}$ bietet wieder verschiedene Strategien zur Behandlung dieser Fälle an.

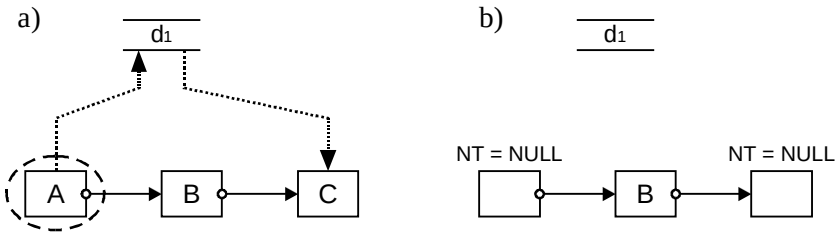


Abbildung 5-4: Dynamisches Löschen von Aktivitäten.

Wenn der in a) abgebildete Knoten A gelöscht werden soll, ist der Datenzugriff des Nachfolgers C auf das Datenelement d_1 nicht mehr versorgt. Eine Möglichkeit besteht darin, einen Nachforderungsdienst einzusetzen, eine andere ist, den abhängigen Knoten C ebenfalls mit zu entfernen. Das Löschen geschieht, indem der ursprüngliche Knoten durch einen leeren Knoten ersetzt wird und die zugehörigen Datenkanten entfernt werden. Damit kann die Änderung später ggf. wieder zurückgenommen werden [RD98].

Bisher sind wir davon ausgegangen, daß verschiedene dynamische Änderungen nie gleichzeitig erfolgen dürfen und währenddessen auch keine Aktivitäten gestartet werden dürfen. Eine Lösung zur Synchronisation von nebenläufigen Änderungen an unterschiedlichen Bereichen des Workflow wurde in [Wei97] erarbeitet. Dabei wird ein optimistisches Synchronisationsverfahren gewählt, bei dem jede dynamische Änderung in zwei Phasen, einer Lese- und einer Schreibphase durchgeführt wird. Zunächst wird auf einer Kopie des aktuellen Graphen gearbeitet, um das Fortschreiten der Instanz nicht durch Sperren zu behindern. Nachdem auch alle notwendigen Begleitänderungen durchgeführt worden sind, d.h. der Graph sich wieder in einem konsistenten Zustand befindet, kann in die Schreibphase übergegangen werden. Sollten sich jetzt Konflikte zum inzwischen aktuellen Zustand, z.B. aufgrund des Fortschreitens in der Ausführung oder zwischenzeitlich durchgeführten Änderungsoperationen, ergeben, so muß die Änderungstransaktion abgebrochen werden. Anderenfalls können die Änderungen zusammen eingebracht werden.

[20] Dies bereitet im vorliegenden Fall keinerlei Schwierigkeiten, da es sich bei n_1 um eine Nullaktivität ohne assoziierte Aktion handelt, also um so etwas wie einen „Dummy“-Knoten.

In [Wei97] wird auch auf spezielle Anforderungen adaptiver WfMS, die sich aus der Unterscheidung von temporären und permanenten Änderungen und der Rücknahme von temporären Änderungen ergeben, eingegangen. Dabei müssen - logisch gesehen - zwei Graphen für eine WF-Instanz gehalten werden, einer mit allen Änderungen und Ausführungsinformation und einer nur mit den permanenten Änderungen [RD98]. Auch im verteilten Fall können auf jedem Server zwei Graphen konstruiert und aktuell gehalten werden. Diese Erweiterungen werden im folgenden außer acht gelassen, um eine Konzentration auf die wesentlichen Aspekte von dynamischen Änderungen im Kontext einer verteilten WF-Steuerung zu ermöglichen. Entsprechende Erweiterungen können allerdings später noch erfolgen, da sie sich in Bezug auf die Verteilung genauso behandeln lassen wie die „gewöhnlichen" dynamischen Änderungen.

Weitere Details zu ADEPT$_{flex}$ findet man in [Hen97] oder [Rei00].

5.2 Herausforderungen dynamischer Änderungen bei verteilter Ausführung

Eine wichtige Voraussetzung für die Durchführung dynamischer Änderungen ist die Verfügbarkeit aktueller Information zum Zustand der Prozeßinstanz. Für jede Änderungsoperation gibt es, bezogen auf den Status einer Instanz, einschränkende Bedingungen, so darf z.B. eine neue Aktivität nicht vor einer schon beendeten Aktivität eingefügt werden[21]. Bei der verteilten Ausführung einer Prozeßinstanz ist ihr Zustand häufig dem Server, von dem die Änderung ausgeht, nicht vollständig bekannt. Befindet sich die Ausführung gerade innerhalb einer parallelen Verzweigung und kontrollieren mehrere Server die Instanz, gibt es demzufolge Aktivitäten, deren Ausführungszustand nicht bekannt ist. Der aktuelle Zustand der Prozeßinstanz muß daher in vielen Fällen zunächst aus den Einzelzuständen der beteiligten Server rekonstruiert werden. Dabei kann die Aktualität nur dann wirklich garantiert werden, wenn mit dem Einholen der Zustandsinformation das Fortschreiten der Ausführung auf den anderen Servern unterbunden wird.

Außerdem muß die Graphstruktur (incl. DF-Kanten) auf dem aktuellen Stand sein, da die Änderung sonst möglicherweise im Konflikt mit schon durchgeführten Änderungen steht und nachträglich zurückgewiesen werden muß. Wenn eine Änderung erfolgreich durchgeführt werden konnte, muß sie mit einem geeigneten Verfahren zum erforderlichen Zeitpunkt zu den anderen Servern propagiert werden. Dabei soll möglichst wenig zusätzliches Kommunikationsvolumen anfallen.

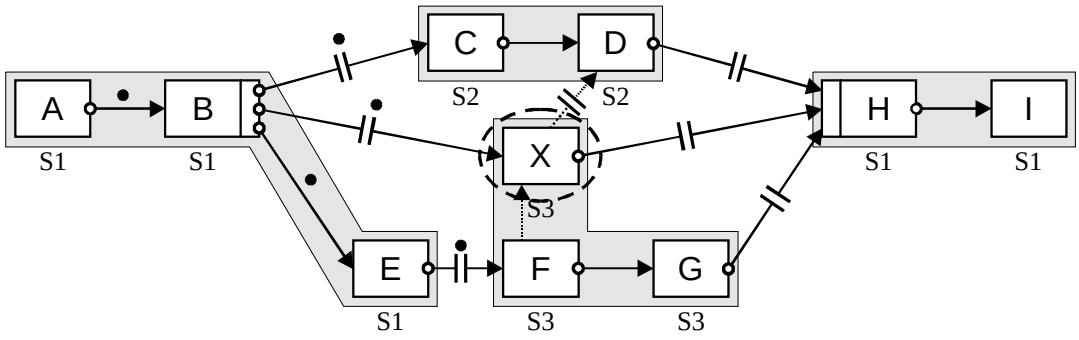


Abbildung 5-5: Dynamische Änderung an verteiltem WF-Graphen.

Im Beispiel aus Abbildung 5-5 wurde vom Bearbeiter von F die Aktivität X neu eingefügt, mit der Bedingung, daß sie nach Beenden von F und vor der Aktivierung von D ausgeführt werden soll. Das Einfügen von X erfolgt als neuer Zweig der parallelen Verzweigung, die gewünschte Ausführungsreihenfolge wird über die neu eingefügten Synchronisationskanten F→X und X→D erreicht.

[21] Die neue Aktivität könnte sonst nicht mehr aktiviert werden. Um Änderungen für künftige Schleifeniterationen, in denen die Bedingungen an den Zustand wieder erfüllt sein werden, durchzuführen, existiert in ADEPT$_{flex}$ das Konzept der vorgemerkten Änderungen. Darauf wird in dieser Arbeit nicht eingegangen.

Vor der Durchführung der Änderung muß der Server S3, von dem die Änderung ausgeht, seine Zustandsinformation soweit auf den neusten Stand bringen, daß er weiß, ob die Aktivität D schon gestartet wurde oder nicht. Nur wenn D noch nicht gestartet wurde, sind die Statusvoraussetzungen für die Änderung erfüllt, und die Einfügeoperation kann durchgeführt werden. Nach der Modifikation darf der Knoten D erst dann aktiviert werden, wenn zusätzlich zur Kante C→D die eingehende Sync-Kante X→D signalisiert worden ist. Bislang ist dem Server S2 die Existenz dieser Kante allerdings noch nicht bekannt. Aus diesem Grund muß der Server S2 nach Durchführung der Änderung über die neue Struktur des Graphen benachrichtigt werden.

Sehr wichtig ist in diesem Zusammenhang, daß die strukturelle Änderung entweder auf allen betroffenen Servern oder überhaupt nicht durchgeführt wird. Es muß abgewägt werden, ob schon während der Vorbereitung der Änderung auf diesen Servern das Fortschreiten im Workflow zeitweise vollständig oder bis zu einer bestimmten Aktivität im Graphen verhindert werden soll oder aber die Änderung zurückgewiesen wird, falls auf einem der Server die Voraussetzungen für die Änderung nicht mehr erfüllt sind.

Die Änderung muß auch dem Server S1 und weiteren im Kontrollfluß folgenden Servern bekannt gegeben werden. Für aktuell nicht beteiligte, aber zukünftig involvierte Server genügt es, die Information bei den folgenden Migrationen weiterzugeben, damit sie auf dem aktuellen Stand, sowohl was Struktur als auch Status des Ausführungsgraphen betrifft, sind, wenn sie die Kontrolle des Workflow übernehmen. Bei Migrationen müssen dann im Vergleich zu einem System ohne dynamische Änderungen zusätzliche Informationen übertragen werden.

In diesem Zusammenhang muß auch entschieden werden, wie strikt die Synchronisation der Struktur der Ausführungsgraphen erfolgen soll. Unter Umständen muß nicht jede Änderung sofort mit allen aktuell an der Ausführung beteiligten Servern synchronisiert werden. Ein großer Vorteil aus der Verteilung, die Möglichkeit zur unabhängigen Ausführung von Aktivitäten paralleler Verzweigungen auf unterschiedlichen Servern, wird bei zu enger Synchronisation unnötig eingeschränkt. Beispielsweise könnte der Bearbeiter von C im Graphen aus Abbildung 5-5 die Aktivität D löschen, ohne daß der Server S2 zusätzliche Zustandsinformation benötigt und ohne daß ein anderer Server sofort über die Änderung in Kenntnis gesetzt wird. Diese Problematik soll im Zusammenhang mit Optimierungen angeschnitten werden.

Des weiteren ergeben sich im Zusammenspiel von dynamischen Änderungen mit verteilter Ausführung folgende Detailprobleme (vgl. [RBD99]):

- Wie stellt ein WF-Server, der eine Änderung durchführen möchte und dessen Ausführung sich gerade innerhalb einer parallelen Verzweigung befindet, überhaupt fest, welche anderen WF-Server gerade an der Ausführung beteiligt sind?
- Inwieweit müssen nach der Durchführung von Strukturänderungen die Serverzuordnungen von Aktivitäten angepaßt bzw. neu festgelegt werden? Da die komplette Neuberechnung einer günstigen Verteilung wohl unangemessen aufwendig ist, muß zumindest für eine neu eingefügte oder verschobene Aktivität eine günstige Zuordnung gefunden werden.
- Wie wird mit Aktivitäten umgegangen, deren Serverzuordnungsausdrücke (z.B. „Server(A)“) nicht mehr ausgewertet werden können, da die referenzierte Aktivität gelöscht wurde?

Die Verfahren zum Übertragen von WF-Kontrolldaten und das Holen-Verfahren zur Beschaffung der WF-Daten für die Folgeaktivität können ohne Probleme übernommen werden. Darüber hinaus kommen nur die Verfahren aus Kapitel 4 in Frage, die als geeignet für die Verwendung bei variablen Serverzuordnungen befunden werden. Auch hier ist noch nicht alle Information über in der Zukunft auszuführende Aktivitäten bekannt. Beispielsweise kann das Schicken-Verfahren zur Übertragung von Datenelementen (vgl. Kapitel 3) nicht mehr angewendet werden, weil es von einer unveränderten Graphstruktur ausgeht. Bei den optimierten Holen-Verfahren muß nach einer dynamischen Änderung der aktuellen Partition eine Neuberechnung der benötigten Datenelemente mit evtl. anschließendem Übertragen fehlender Datenelemente stattfinden.

5.3 Möglichkeiten zur Synchronisation dynamischer Änderungen einer Multi-Server-Umgebung

Eine Änderung an einer Prozeßinstanz muß garantiert bei allen Servern, die die Ausführung der Instanz aktuell kontrollieren und die von der Änderung betroffen sind, durchgeführt werden. Hierfür gibt es unterschiedlich strikte Synchronisationsverfahren, die sich hinsichtlich der beiden folgenden Kriterien unterscheiden:

- Anzahl der Server, mit denen synchronisiert wird.
- Zeitpunkt, zu dem mit einem bestimmten Server synchronisiert wird.

Auch wenn dynamische Änderungen relativ selten stattfinden, ist es für Eigenschaften wie Performanz und Kommunikationsaufkommen im verteilten WfMS von entscheidender Bedeutung, welche Variante gewählt wird.

Das eigentlich auch unter den Begriff „Synchronisation“ fallende Einholen von Informationen zum Ausführungszustand aller aktuell an der Ausführung beteiligten Server wird im Vergleich der Verfahren nicht berücksichtigt. Es ist bei Änderungen, die von Aktivitäten innerhalb einer parallelen Verzweigung ausgehen, je nach Änderungsoperation notwendig.

5.3.1 Strikte Synchronisation aller Server des WfMS

Eine Synchronisationsvariante wäre es, eine Änderung allen WF-Servern des WfMS bekannt zu machen. Bei einer solch strikten Synchronisation könnte auf allen Servern immer die aktuelle Struktur eines Instanzgraphen vorausgesetzt werden. Es würden sich somit für die Ausführung einer Prozeßinstanz und für die Migrationen keinerlei Unterschiede zum bisherigen Verfahren ergeben.

Eine solche Vorgehensweise ist jedoch sehr teuer, bei einem großen System mit häufigen WF-Änderungen führt sie leicht zum Kommunikationsoverkill. Insbesondere wenn die Anzahl der im System vorhandenen Server größer ist als die Zahl der Server, die die geänderte Prozeßinstanz zu irgendeinem Zeitpunkt kontrollieren, werden größtenteils unnötige Informationen verschickt. Hinzu kommt, daß die Prozeßinstanz zunächst gar nicht auf allen Servern vorhanden ist. Sie existiert nur auf denjenigen Servern, die bereits an der Ausführung der Instanz beteiligt waren oder dies im Moment aktuell sind.

Bei der beschriebenen Vorgehensweise werden auf zahlreichen Servern Kopien von Prozeßinstanzen angelegt und verändert, die dort im Fall von statischen Serverzuordnungen nie und im Fall von variablen Serverzuordnungen nur mit geringer Wahrscheinlichkeit benötigt werden.

Da Änderungen entweder auf allen oder auf keinem Server durchzuführen sind, werden sehr hohe Ansprüche an die gleichzeitige Verfügbarkeit aller Server gestellt, zumindest wenn im System keine Komponente vorhanden ist, die die Änderungen auf einem zur Zeit inaktiven Server nachträglich einbringen kann.

Die Vorteile der verteilten Ausführung von Workflows, daß nur ein kleiner Teil der Server für eine bestimmte Prozeßinstanz zuständig ist, gehen damit verloren. Die strikte Synchronisation ist für den praktischen Einsatz somit in den meisten Fällen ungeeignet.

5.3.2 Synchronisation aller aktuellen und zukünftigen Server der WF-Instanz

Eine bessere Variante als in Abschnitt 5.3.1 ist es, nur eine Teilmenge der Server des verteilten WfMS zu synchronisieren. Dafür muß der Server, von dem die Änderung ausgeht, zunächst die aktuell an der Ausführung beteiligten Server ermitteln. Wie auch die Server von Aktivitäten im Zustand NOT_ACTIVATED, d.h. die zukünftigen Server, müssen sie über die Änderung informiert werden. Bei statischen Serverzuordnungen stellt dieses Verfahren eine echte Alternative zu dem in Abschnitt 5.3.1 beschriebenen Verfahren dar.

Für variable Serverzuordnungen scheidet das Verfahren aus, da evtl. noch nicht alle in der Zukunft an der WF-Ausführung beteiligte Server bekannt sind.

Ein Vorteil der Synchronisation von Änderungen mit allen aktuell und zukünftig beteiligten Servern der WF-Instanz ist, daß in den meisten Fällen[22] kein unnötiges Kommunikationsvolumen anfällt. Alle diese Server müssen die Änderung spätestens vor der Übernahme der WF-Kontrolle mitgeteilt bekommen, damit sie über die aktuelle Struktur des Graphen verfügen. Die Ausführung der Instanz und die Migrationen können wie bisher durchgeführt werden.

Wenn im System keine Komponente vorhanden ist, die die Änderungen auf einem zur Zeit inaktiven Server nachträglich einbringen kann, müssen alle aktuellen und zukünftigen Servern der WF-Instanz gleichzeitig verfügbar sein. Möglicherweise sind auch die für die Ausbreitung der Änderungen anfallenden Kommunikationskosten noch zu hoch. Zumindest wenn der Server, von dem die Änderung ausgeht, schlecht angebunden ist, wäre es günstiger, die Änderungsinformation mit den regulären Migrationen zu übertragen.

5.3.3 Synchronisation aller aktuell an der Ausführung beteiligten Server

Die Synchronisation mit den aktuell an der Ausführung beteiligten Servern läßt sich bei beiden Varianten der Serverzuordnungen durchführen. Mit dem Verfahren ist gewährleistet, daß zum Zeitpunkt der Änderung alle Server, die gerade Aktivitäten der WF-Instanz kontrollieren, den aktuellen Zustand des Graphen kennen. In der Zukunft an der WF-Ausführung beteiligten Servern müssen die Informationen über den geänderten Graphen bei späteren Migrationen mitgeteilt werden. Sie müssen daher im Moment noch nicht verfügbar sein.

Da die Änderungen zwischen allen an der Ausführung beteiligten Servern immer vollständig synchron erfolgen, entstehen an Knoten mit mehreren Eingangskanten, die von unterschiedlichen Servern kontrollierte Vorgängerknoten haben, keine Probleme. Während die Ausführungshistorien explizit zusammengeführt werden müssen (vgl. Abschnitt 3.3.2), da jeder der Server unterschiedliche Zustandsinformationen besitzen kann, muß dies mit der Änderungshistorie nicht gemacht werden. Jeder Migrationsquellserver verfügt bereits über die gleiche Änderungshistorie.

Um ein möglichst niedriges Kommunikationsvolumen zu erreichen, muß vermieden werden, daß dieselben Informationen mehrfach zu einem Server übertragen werden.

Aufgrund von Vorteilen wie der Vermeidung unnötiger Kommunikation und der Tatsache, daß das Verfahren auch bei variablen Serverzuordnungen einsetzbar ist, soll es in den folgenden Abschnitten für die Algorithmen zur Unterstützung von verteilten dynamischen Änderungen vorausgesetzt werden.

5.3.4 Synchronisation einer Teilmenge der aktuellen Server

Um die unabhängige Bearbeitung paralleler Zweige besser zu berücksichtigen, ist es in manchen Fällen wünschenswert, nur eine Teilmenge der aktuellen Server zu synchronisieren. Möglicherweise ist aktuell auch gar keine Synchronisation erforderlich. Wird beispielsweise in einem Zweig eine Aktivität gelöscht, ergeben sich keine Auswirkungen auf parallele Aktivitäten (vorausgesetzt diese liegen nicht im Nachfolgebereich des gelöschten Knoten über Sync-Kanten). Man könnte also versuchen, auf die sofortige Synchronisation zu verzichten und die Änderungsinformation erst bei zukünftigen Migrationen auf andere Server zu verbreiten.

[22] Lediglich die Änderungsinformation für einen Server, der nur Knoten innerhalb einer bedingten Verzweigung kontrolliert, die übersprungen werden wird, wird unnötig übertragen.

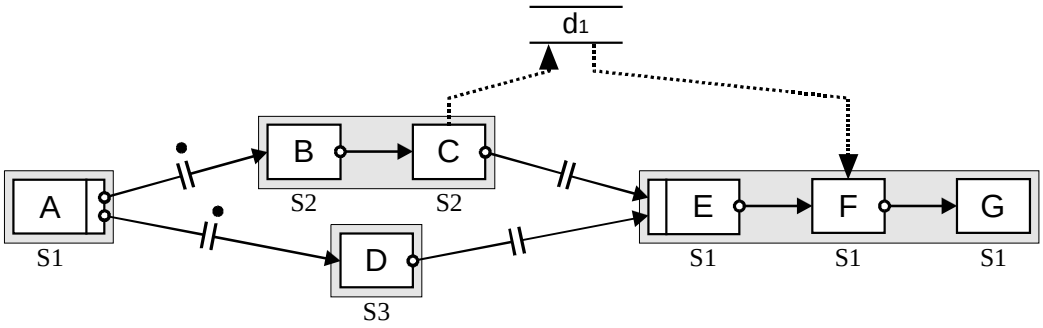


Abbildung 5-6: Synchronisation mit einer Teilmenge der aktuellen Server.

Wird in Abbildung 5-6 die Aktivität C und ihre datenabhängige Folgeaktivität F gelöscht, so muß der Server S3 darüber nicht sofort benachrichtigt werden. Mit der gleichen Argumentation könnte der Bearbeiter der Aktivität D danach eine Aktivität X zwischen F und G einfügen, welche d_1 liest und nun nicht mehr versorgt ist. Erkannt wird das Problem erst bei der Migration zum Joinknoten E bzw. sogar erst durch einen Fehler bei der Ausführung. Hier hätte also zumindest S3 von S2 die aktuelle Struktur des Graphen erfragen müssen.

Bleiben die Auswirkungen der Modifikation auf die aktuelle Partition beschränkt, z.B. wenn der Bearbeiter von B eine Aktivität Y zwischen B und C einfügt, muß in der Regel keine sofortige Synchronisation stattfinden. Allerdings kann es sogar in diesem Fall zu Problemen kommen, wenn beispielsweise Y ein Datenelement d_2 schreibt und im anderen Teilzweig eine Aktivität Z eingefügt wurde, die ebenfalls d_2 schreibt. Das parallele Schreiben eines Datenelements ist nicht zulässig [Rei00].

Aufgrund der zahlreichen Probleme und Sonderfälle werden diese Variante und die damit verbundenen Schwierigkeiten in Abschnitt 5.7 gesondert behandelt.

5.4 Migrationen bei geänderten WF-Instanzen

Im folgenden wird das in Abschnitt 5.3.3 eingeführte Verfahren betrachtet, bei dem Änderungen nur mit den aktuell an der Ausführung der Instanz beteiligten Servern synchronisiert werden. Den in der Zukunft an der WF-Ausführung beteiligten Servern muß die neue Struktur des Ausführungsgraphen bei den Migrationen mitgeteilt werden. Hierzu gibt es unterschiedliche Möglichkeiten, die mehr oder weniger hohe Übertragungskosten verursachen.

5.4.1 Generelle Alternativen

1. Eine naive Möglichkeit für die Weitergabe der aktuellen Struktur des WF-Graphen besteht darin, den kompletten Instanzgraphen zu übertragen. Um den Graphen nicht bei jeder Migration transferieren zu müssen, kann dessen aktuelle Versionsnummer zuvor mit der Versionsnummer des Migrationszielservers verglichen werden, so daß nur bei einem Unterschied weitere Kommunikation notwendig wird. Bei jeder dynamischen Änderung muß die für dieses Verfahren neu einzuführende Versionsnummer erhöht werden.

   Das Verfahren bringt allerdings zahlreiche Probleme mit sich. Beim Zusammentreffen paralleler Zweige über eine Migration sind die zu übertragenden Graphen zwar von der Struktur her gleich, sie enthalten aber alle nicht die vollständigen Zustandsmarkierungen. Die einfachste Möglichkeit, um beim Zielserver den neuen gemeinsam gültigen Zustand herzustellen ist, nach dem Eintreffen aller Migrationen die Zustandsmarkierungen vollständig zu entfernen und anhand der inzwischen vorhandenen Ausführungshistorie neu zu setzen. Historieneinträge, zu denen keine Knoten vorhanden sind, weil diese z.B. entfernt wurden, können ignoriert werden.

   Zusätzlich zum Graphen muß die Änderungshistorie übertragen werden. Fehlt diese, können später Änderungen nicht mehr nachvollzogen und daher auch nicht mehr automatisch rückgängig gemacht werden.

Darüber hinaus ist die Datenmenge für die Übertragung des kompletten Graphen relativ groß. Die Maßnahmen zum Minimieren der Kommunikationskosten, die für die Weitergabe der Zustandsinformation eingeführt wurden, werden wieder zunichte gemacht.

2. Eine bessere Lösung sieht ausschließlich die Übertragung der (kompletten) Änderungshistorie vor. Anhand der darin enthaltenen Informationen kann auf dem Zielserver der neue Ausführungsgraph rekonstruiert werden.

   Dabei ist folgendermaßen vorzugehen: Die eingegangene Ausführungshistorie wird Eintrag für Eintrag durchgearbeitet und die Statusmarkierungen des Instanzausführungsgraphen entsprechend aktualisiert. Wenn ein Eintrag verarbeitet wird, der eine Änderungsoperation aus der Änderungshistorie referenziert, wird die Änderung nur noch dann eingebracht, wenn der Eintrag aus der Ausführungshistorie neu in die Ausführungshistorie des Zielservers aufgenommen worden ist. Die Verweise auf eine bestimmte Änderung können nämlich in mehreren eintreffenden Teilhistorien enthalten sein, wenn die Änderung während der Ausführung von Aktivitäten innerhalb einer parallelen Verzweigung stattgefunden hat.

   Für das Einarbeiten einer Änderung muß die komplexe Änderungsoperation nicht nochmals neu berechnet werden. Es genügt, die angegeben Änderungsprimitive auf den Graphen anzuwenden.

3. Sinnvollerweise sollten nur die Einträge der Änderungshistorie übertragen werden, die dem Zielserver noch nicht bekannt sind. Da die Einträge der Änderungshistorie auf allen beteiligten Servern gleich geordnet sind, muß dazu lediglich bestimmt werden, welches der letzte Eintrag ist, der auf dem Zielserver vorhanden ist. Alle dem Quellserver bekannten neueren Einträge fehlen dort noch. Die Rekonstruktion des Graphen erfolgt wie in 2. beschrieben.

4. Es ist nicht notwendig, die kompletten Änderungshistorieneinträge zu übertragen. Für eine Rekonstruktion der Änderung ausreichend ist der in Definition 5-1 eingeführte verkürzte Historieneintrag, bei Zulassung temporärer Änderungen noch zzgl. der Gültigkeitsdauer der Änderung. Wenn die Änderungsoperation erneut berechnet wird, ergeben sich wieder die Änderungsprimitive. Sie können auf den Graphen angewandt und anschließend wieder in ausführlicher Form in der Änderungshistorie des Zielservers gespeichert werden.

   Hierdurch entsteht auf dem Zielserver allerdings erhöhter Berechnungsaufwand. Zum einen muß die Änderungsoperation neu berechnet werden, zum anderen muß dafür der komplette Instanzgraph aus der Datenbank gelesen werden. Im Fall der Anwendung der Elementaroperationen ist dies nicht notwendig. Es muß daher abgewägt werden, mit welchem Ziel optimiert werden soll.

5. Die größte Einsparung an Kommunikationsaufkommen wird erreicht, wenn 3. und 4. kombiniert werden, d.h. ein Teil der Änderungshistorie, bestehend aus verkürzten Einträgen, transferiert wird. Diese Variante wird im folgenden betrachtet.

5.4.2 Übertragen eines Teils der Änderungshistorie

Wie bei der Übertragung der Ausführungshistorie bietet sich für den Transfer von Einträgen der Änderungshistorie eine Schicken- und eine Holen-Variante an. Anhand von Abbildung 5-7 wird deutlich, daß beim Schicken-Verfahren redundante Übertragungen nicht vermeidbar sind. Das folgende Beispiel illustriert dies.

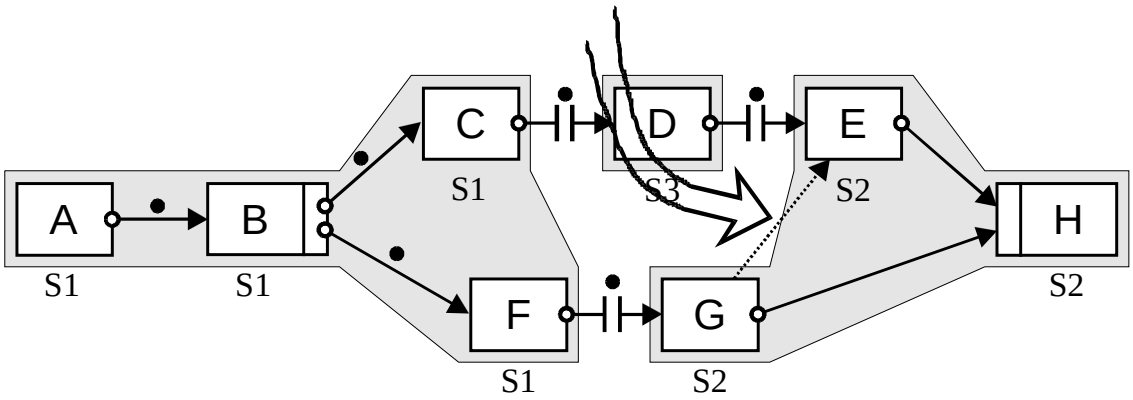


Abbildung 5-7: Redundantes Übertragen von Historieninformation im Schicken-Verfahren.

Der Bearbeiter von B veranlaßt das Einfügen der Synchronisationskante G→E. Bei der Migration $M_{C,D}$ wurde der zugehörige Änderungshistorieneintrag „789, insertSyncEdge(G, E)“ bereits an den Server S3 übermittelt. Der Historieneintrag muß auch bei den Migrationen $M_{F,G}$ und $M_{D,E}$ verschickt werden, da dem Server S1 bei dieser Methode nicht bekannt ist, ob die Migration $M_{D,E}$ schon stattgefunden hat oder nicht (bzw. S3 weiß nichts über $M_{F,G}$). So wird die Änderungsinformation zweimal an S2 geschickt, wenngleich sie dort nur beim ersten Eintreffen berücksichtigt wird.

Besser ist es daher, nach einem Holen-Protokoll vorzugehen und vom Zielserver zu erfragen, welches sein letzter Eintrag in der Änderungshistorie ist. Wenn dieser gleichzeitig mit der Übertragung der Migrationspunkte[23] die ID des letzten Eintrags aus seiner Änderungshistorie mitteilt, wird für die Rückfrage sogar kein zusätzlicher Kommunikationsschritt nötig. Die Übertragung aller Änderungshistorieneinträge, die dem Eintrag mit der übermittelten ID nachfolgen, kann zusammen mit der Ausführungshistorie durchgeführt werden. Der Migrationszielserver kann die erhaltenen Änderungshistorieneinträge bei sich anhängen.

Die Aktualisierung der Graphstruktur geschieht folgendermaßen: Die eingegangene Ausführungshistorie wird Eintrag für Eintrag durchgearbeitet und die Statusmarkierungen aktualisiert (vgl. Abschnitt 3.3.2). Wenn ein Eintrag verarbeitet wird, der eine Änderungsoperation aus der Änderungshistorie referenziert (wie z.B. „DDM(1234)“), so darf die Änderung nur dann eingearbeitet werden, wenn dieser Eintrag neu in die Ausführungshistorie des Zielservers aufgenommen wurde. Da die Anker zu den Änderungshistorieneinträgen in der Ausführungshistorie jedes Servers erzeugt werden, der zum Zeitpunkt der Änderung an der Ausführung beteiligt ist, wird ein DDM-Anker-Historieneintrag möglicherweise bei mehreren Migrationen zu einem Knoten mit mehr als einer Eingangskante innerhalb der Ausführungshistorie mit übertragen[24]. Er darf jedoch nur einmal in die Ausführungshistorie aufgenommen und eingearbeitet werden. Beim Zusammenführen der Historien muß daher wieder auf das ursprüngliche, nicht optimierte Verfahren, welches auch Duplikate erkennen kann, zurückgegriffen werde.

Für das Einarbeiten einer Änderung muß die mit dem verkürzten Änderungshistorieneintrag beschriebene Änderungsoperation noch einmal neu berechnet werden. Es ergeben sich wieder die Änderungsprimitive. Sie können auf den Graphen angewandt und anschließend in ausführlicher Form

---

[23] Die Migrationspunkte werden beim Übertragen von Ausführungshistorie im Holen-Verfahren verwendet, um sicherzustellen, daß keine redundanten Historieneinträge übermittelt werden (vgl. Abschnitt 3.4.2).

[24] Da sich die Einträge keinen Knoten zuordnen lassen, kann im Gegensatz zu den regulären Ausführungshistorieneinträgen keine Filterung vorgenommen werden, so daß redundante Übertragung unvermeidbar ist.

Eine Optimierung wäre möglich, wenn vorausgesetzt werden könnte, daß für alle Ankereinträge eigene Änderungshistorieneinträge existieren. Dann müßte auch ein Ankereintrag nur dann übertragen werden, wenn der referenzierte Eintrag übertragen wird. Für UDM[25]-Einträge gilt dies jedoch nicht, sie referenzieren derzeit einen bereits bestehenden Änderungshistorieneintrag.

in der Änderungshistorie des Zielservers gespeichert werden. Bei einem UDM[25]-Eintrag muß eine vorhandene Änderung wieder zurückgenommen werden.

5.5 Synchronisation der an der WF-Instanz aktuell beteiligten Server

Zur Synchronisation der aktuell an der Ausführung einer Instanz beteiligten Server müssen diese zuerst ermittelt werden. In $ADEPT_{distribution}$ ist jedoch bei parallelen Verzweigungen den Servern des betrachteten Teilzweigs nicht immer bekannt, wie weit die anderen Teilzweige fortgeschritten sind und welche Server die Ausführung der Aktivitäten dieses Zweiges kontrollieren. So weiß im Szenario aus Abbildung 5-7 der Server S3 nicht, ob gerade S1 die Aktivität F kontrolliert oder ob die Migration $M_{F,G}$ schon stattgefunden hat und damit S2 die Kontrolle übernommen hat. Von den beteiligten Servern muß ggf. aber Zustandsinformation eingeholt werden, um zu prüfen, ob eine Änderung zulässig ist. Schließlich muß die Änderung nach dem Alles-oder-Nichts-Prinzip allen beteiligten Servern bekannt gemacht werden.

Die korrekte Durchführung einer dynamischen Änderung ist im verteilten Fall nicht trivial. Die Schwierigkeiten entstehen dadurch, daß die Ausführung der WF-Instanz auf den anderen Servern fortschreitet und sogar Migrationen stattfinden können, während ein Benutzer eine Änderung vorbereitet. Da die Vorbereitung einer Änderung aufgrund von Benutzerinteraktion relativ lange dauert, kann es in der Regel keine Lösung sein, die Ausführung der Instanz für diese Zeit anzuhalten, auch wenn dies bei den meisten der heutigen „adaptiven“ WF-Modellen bzw. WfMS ohnehin der Fall ist.

5.5.1 Gegenseitiger Ausschluß bei Migrationen und Änderungen

Im Folgenden beziehen wir uns immer auf exakt eine WF-Instanz und ihren Ausführungsgraphen. Zwischen Änderungen unterschiedlicher WF-Instanzen ist volle Parallelität möglich.

Ausschluß zwischen Migrationen

Prinzipiell dürfen zu einer WF-Instanz gehörende Migrationen innerhalb verschiedener Teilzweige unabhängig voneinander gleichzeitig stattfinden. Anderenfalls wäre den Servern des WfMS eine zu starke Einschränkung der Unabhängigkeit auferlegt, da die Durchführung einer Migration sonst warten muß bis die komplette Datenübertragung der laufenden Migration abgeschlossen ist.

Allerdings muß jeder Migrationszielserver die bei ihm eingehenden Migrationen zur selben WF-Instanz als untrennbare Einheit betrachten und nacheinander abwickeln, da sonst die Gefahr besteht, daß Daten redundant übertragen werden. Im Fall von gleichzeitig bearbeiteten Migrationen werden vom Zielserver an beide Migrationsquellserver möglicherweise die gleichen Migrationspunkte geschickt. Bei sequentieller Durchführung werden zunächst nur einem Quellserver die Migrationspunkte geschickt und die von dort erhaltene Ausführungshistorie auf dem Zielserver eingearbeitet. Danach wird unter Berücksichtigung der inzwischen vorhandenen Informationen erneut die Menge der Migrationspunkte berechnet. Diese ist unterschiedlich zu der bei der ersten Migration ermittelten Menge. Falls die Migrationsquellserver über gemeinsame Informationen verfügen, werden damit die Aktivitäten aus den „Überlappungsbereichen“ weggelassen. So wird vermieden, daß Historieneinträge doppelt verschickt werden (vgl. Abschnitt 3.4.2).

Wenn die Dauer für die Datenübertragung relativ kurz gegenüber der Zeitspanne ist, die vom Beenden eines Arbeitsschrittes bis zur Aktivierung des Folgeschrittes auf einem anderen Server maximal toleriert wird, d.h. Migrationen nicht unbedingt schnellstmöglich abgewickelt werden müssen, kann es vertretbar sein, alle Migrationen einer Prozeßinstanz vollständig sequentiell durchzuführen. Dadurch gestaltet sich, wie später in Abschnitt 5.5.3 zu erkennen sein wird, das Protokoll zur Durchführung von Migrationen einfacher, da ein Kommunikationsschritt wegfällt.

---

[25] UDM = Undo Dynamic Modification; Eintrag in der Ausführungshistorie, der die Rückname einer temporären Änderung anzeigt.

Ausschluß zwischen dynamischen Änderungen

Bezogen auf dieselbe Instanz können mehrere dynamische Änderungen gleichzeitig vorbereitet werden. Sie müssen allerdings nacheinander ins System eingebracht werden, um eventuell auftretende Konflikte zwischen den Änderungen feststellen zu können. Anderenfalls besteht bei Operationen, deren Änderungsbereiche sich überlappen, die Gefahr, daß sich die Operationen vermischen und die Graphstruktur inkorrekt wird. Auch in Bezug auf Änderungen eines Datenflußschemas (z.B. Löschen und Hinzufügen von Datenkanten bezogen auf dasselbe Datenelement) können Konflikte zwischen konkurrierenden Änderungen bestehen.

Ausschluß zwischen Migrationen und dynamischen Änderungen

Problematischer ist die Handhabung von gleichzeitig stattfindenden Migrationen und dynamischen Änderungen. Ein Migrationsquellserver, der gerade die für den Zielserver benötigten Instanz-Daten überträgt und dabei eine Nachricht erhält mit der Aufforderung, den Ausführungsgraphen dieser Instanz zu verändern, muß die Information aus dieser Nachricht an den Migrationszielserver weitergeben. Dies läßt sich grundsätzlich verwirklichen.

Während des Umschaltens der WF-Kontrolle vom Quellserver auf den Zielserver muß sichergestellt sein, daß im System keine Nachrichten zum Zweck der Ausbreitung von dynamischen Änderungen mehr unterwegs sind. Sie würden den Migrationsquellserver zu einem Zeitpunkt erreichen, zu dem er nicht mehr zu den aktiven Servern zählt. Würde man diese Einschränkung nicht machen, d.h. könnten Änderungsaufträge auch nach der Weitergabe der Kontrolle beim Quellserver eintreffen, müßte er alle Änderungsaufträge und auch die zugehörigen Aufträge zum Aufheben von Sperren an den Migrationszielserver „nachsenden“. Neben den im Rahmen der Migration regulär übertragenen Informationen müßten dann auch die Sperren weitergegeben werden. Der Grund dafür ist, daß ohne Berücksichtigung der Sperren die Änderungsoperation möglicherweise nicht mehr zulässig ist, weil die Ausführung bereits zu weit fortgeschritten ist. Wenn Sperren weitergegeben werden, muß dies immer vollständig protokolliert werden oder der Sperrenerzeuger informiert werden. Anderenfalls kann nicht mehr nachvollzogen werden, wo Sperren gesetzt sind, so daß die Sperren auf den anderen Servern nicht mehr aufgehoben werden könnten.

Um diesen nicht unerheblichen Zusatzaufwand zu vermeiden[26], sollten Migrationen und die Ausbreitung von dynamischen Änderungen gegenseitig ausgeschlossen werden. Dazu kann ein Verfahren mit Semaphoren analog dem Leser/Schreiber-Problem, bei dem die Leser Priorität haben und nur warten müssen, wenn ein Schreiber aktiv ist, verwendet werden. Migrationen entsprechen dabei Lesern, und die Ausbreitung einer dynamischen Änderung einem Schreiber. Es dürfen mehrere Migrationen gleichzeitig stattfinden, jedoch darf eine Änderung immer nur exklusiv eingebracht werden. Eine vorteilhafte Eigenschaft bei dieser Vorgehensweise ist, daß sich nach dem Gewähren der Schreibsperre die Menge der aktuell an der Ausführung beteiligten Server nicht mehr ändern kann.

5.5.2 Feststellung der an der WF-Instanz aktuell beteiligten Server

Für das Einholen von Zustandsinformation über eine verteilte WF-Instanz und für das Ausbreiten von Änderungsinformation müssen die aktuell an der Instanz beteiligten Server *ActiveServers* ermittelt werden. Dabei ist zu berücksichtigen, daß bei Aufrufen an diese Server feststellbar sein muß, ob der aufrufende Server noch über die aktuelle Information in Bezug auf *ActiveServers* verfügt, d.h. in der Zwischenzeit darf keine die Menge *ActiveServers* verändernde Migration stattgefunden haben. Ansonsten besteht die Gefahr, daß bei Anfragen, die an alle Server aus *ActiveServers* gerichtet werden, ein neu dazugekommener Server vergessen wird oder ein nicht mehr beteiligter Server eine unnötige Aktion ausführen muß.

Es sind also grundsätzlich zwei Aufgaben zu erledigen:

1. Feststellung der aktuell an der Ausführung der Instanz beteiligten Server *ActiveServers*.

---

[26] Der Zusatzaufwand ist natürlich dann zu rechtfertigen, wenn er auch wirklich Vorteile bringt, z.B. wenn feingranulare, nur für bestimmte Aktivitäten wirksame, Sperren verwendet werden.

2. Bei der Kommunikation mit einem Server aus *ActiveServers* soll festgestellt werden, ob sich die Menge der aktuell beteiligten Server möglicherweise geändert hat, d.h. die Feststellung erfolgt ohne erneut die Menge der aktuellen Server abzufragen. Dieser Teil kann als Optimierung betrachtet werden.

In der in Abbildung 5-8 gezeigten WF-Instanz sind zum aktuellen Zeitpunkt die Server S1 (Aktivität D), S2 (Aktivität C) und S3 (Aktivität F) an der Ausführung beteiligt.

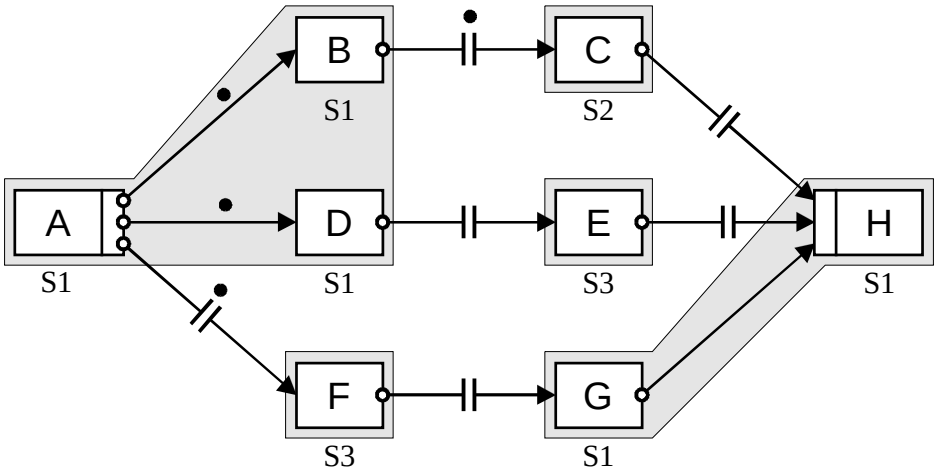


Abbildung 5-8: Bestimmung der aktuell an der Ausführung beteiligten Server.

Für die weitere Betrachtung führen wir folgende Definitionen ein:

Definition 5-2: Von einem Server S kontrollierte Aktivität.

Eine Aktivität wird aktuell von einem Server S kontrolliert, wenn sie dem Server S zugeordnet ist und sich im Zustand ACTIVATED, SELECTED oder RUNNING befindet. Hinzu kommen Aktivitäten des Servers S im Zustand COMPLETED, CANCELLED oder FAILED, bei denen sich eine ausgehende Kante im Migrationszustand MIGRATION, MIGRATING oder CHECKMIGRATION befindet. Eine Aktivität, die diese Bedingungen erfüllt, gilt als aktiv.

Definition 5-3: An der Ausführung einer Instanz beteiligter Server.

Ein Server S ist gerade an der Ausführung einer Instanz beteiligt, wenn er mindestens eine Aktivität der Instanz kontrolliert.

Bestimmung von *ActiveServers* mittels Broadcast

1. Zur Bestimmung der Menge *ActiveServers* bietet sich ein Broadcast-Verfahren an. Dabei wird an alle WF-Server des Systems die WF-Instanz-ID verschickt, und jeder Server, der mindestens eine Aktivität der Instanz kontrolliert, antwortet dem Anfragenden. Wenn man davon absieht, daß diese Methode bezüglich der Vermeidung unnötiger Nachrichten ungeschickt ist, bringt sie mindestens ein weiteres Problem mit sich. Wie soll mit Servern umgegangen werden, die gerade eine Migration durchführen? In einem solchen Fall könnte der Migrationsquellserver antworten. Der Migrationsquellserver gibt aber möglicherweise seine Kontrolle an der Ausführung des Workflow ab, und der Zielserver wird zwar in Zukunft an der Ausführung beteiligt sein, verfügt aber während der Migration noch nicht über den aktuellen Zustand der Instanz. Trifft die Anfrage zu einem ungünstigen Zeitpunkt ein, nämlich genau während der Übergabe der Kontrolle, kann es je nach Verfahren passieren, daß weder der Quell- noch der Zielserver oder aber beide antworten.

Die Richtigkeit der so festgestellten Menge der aktuellen Server ist daher in manchen Fällen fragwürdig.

2. Zur Überprüfung der Aktualität von *ActiveServers* - sie wird z.B. notwendig wenn ein Server S Zustandsinformationen einholt - muß sich jeder einzelne Server S_i merken, ob seit der letzten von S verschickten Broadcast-Anfrage von S_i eine ausgehende Migration stattgefunden hat. In einem solchen Fall muß S_i von S kommende Aufrufe, die für alle Server aus *ActiveServers* bestimmt sind, zurückweisen und S muß veranlaßt werden, *ActiveServers* neu zu bestimmen. Ansonsten könnten für die Bearbeitung der Instanz hinzugekommene Server vergessen werden, wenn S von einem veralteten Wert ausgeht. Um das Speichern der eingegangenen Anfragenachrichten zu vermeiden, könnte nach jeder Migration ein Broadcast gesendet werden, der die Information enthält, daß *ActiveServers* ungültig gemacht und bei Bedarf neu berechnet werden muß.

Koordinator-Lösung

Im Vergleich zum Broadcast ist eine Lösung, bei der die Information über *ActiveServers* an einer zentralen Stelle aktuell gehalten wird, in Bezug auf das Kommunikationsverhalten wesentlicher günstiger. Um keinen Flaschenhals zu schaffen, wird kein speziell für diese Aufgabe festgelegter Server verwendet. Als Koordinator bietet sich zum Beispiel der Startserver der Instanz an.

Zunächst finden wir eine Lösung für das erste Problem, die Bestimmung von *ActiveServers*. Voraussetzung hierfür ist, daß die Menge aktualisiert wird, wenn sie sich ändert. Bei einer Migration schickt der Migrationsquellserver S_{Source} dem Startserver - stark vereinfacht ausgedrückt - eine Mitteilung darüber, daß eine Migration von S_{Source} nach S_{Target} stattgefunden hat. Je nachdem, ob Migrationen serialisiert werden, bieten sich unterschiedliche Varianten an.

Variante 1: Migrationen sind vollständig parallel erlaubt.

Der Startserver unterhält für die Instanz eine Datenstruktur, in der zu den aktuellen Server-IDs die Anzahl der Aktivitäten vermerkt wird, die vom jeweiligen Server kontrolliert werden (siehe Definition 5-2). Für einen bestimmten Server ist ein Eintrag nur dann vorhanden, falls er an der Ausführung aktuell beteiligt ist. Wenn ein Server n Aktivitäten innerhalb einer parallelen Verzweigung kontrolliert, ist der Wert n beim Startserver gespeichert. Die Menge *ActiveServers* ergibt sich aus den Servern der aktuellen Einträge. Ein Nachteil des Verfahrens ist, daß bei parallelen Verzweigungs- und Joinknoten eine Sonderfallbetrachtung notwendig ist, um den Zähler konsistent zu halten[27]. Selbst wenn keine Migration stattfindet, müssen bei dieser Variante beim Betreten und Verlassen einer parallelen Verzweigung die Zähler des Startservers aktualisiert werden, da sich die Anzahl der kontrollierten Aktivitäten ändert!

Variante 2: Zumindest zum Zeitpunkt der Umschaltung der Kontrolle ist gegenseitiger Ausschluß der Migrationen gewährleistet, die Datenübertragungen anläßlich der Migrationen dürfen parallel stattfinden.

Der Startserver verwaltet die Menge *ActiveServers*. Ein Migrationsquellserver teilt dem Startserver mit, ob er weiterhin an der Ausführung der Instanz beteiligt sein wird („stay“) - dies ist der Fall, wenn er noch weitere Aktivitäten der Instanz kontrolliert - oder ob er die Kontrolle abgibt („log_off“). Je nach Meldung verbleibt der Quellserver in *ActiveServers*, oder er wird aus der Menge entfernt. Der Zielserver kann immer in die Menge aufgenommen werden.

Die Entscheidung zwischen „stay“ und „log_off“ und das darauffolgende Aktualisieren der Menge muß mit einem kritischen Abschnitt beim Migrationsquellserver und beim Startserver geschützt werden. Wenn etwa im Beispiel aus Abbildung 5-8 die Migrationen $M_{D,E}$ und $M_{F,G}$ gleichzeitig auf den Servern S1 und S3 beginnen, gehen beide Quellserver davon aus, daß sie die Kontrolle abgeben („log_off“). Findet $M_{D,E}$ zuerst statt, gilt nach dem Entfernen von S1 im Anschluß

[27] Auch wenn ein Server mehrere Aktivitäten am Beginn verschiedener Zweige kontrolliert, muß dafür nur eine Migration stattfinden. Bei einer Migration zu einem Joinknoten darf dessen Zähler nur einmal erhöht werden.

ActiveServers = {S2, S3}. Bei der Migration $M_{F,G}$ wird S3 fälschlicherweise aus der Menge entfernt, so daß *ActiveServers* = {S1, S2} anstatt *ActiveServers* = {S1, S2, S3} resultiert. Es muß also verhindert werden, daß eine aufgrund veralteter Information getroffene Entscheidung zur Inkonsistenz von *ActiveServers* führt.

Dementsprechend muß bei gleichzeitigem Beginn der Migrationen $M_{B,C}$ und $M_{D,E}$ in Abbildung 5-8 sichergestellt sein, daß die Entschlüsse, ob S1 die Kontrolle abgeben wird, einschließlich der auf jede Entscheidung folgenden Aktualisierung von *ActiveServers* für beide Migrationen nacheinander durchgeführt werden. Anderenfalls könnte es passieren, daß sich S1 bei beiden Migrationen für „stay“ entscheidet und somit fälschlicherweise in *ActiveServers* verbleibt. Durch den kritischen Abschnitt wird auch dieser Fehler ausgeschlossen.

Die Variante 2 soll im Weiteren betrachtet werden. Mit dem im folgenden in UML[28] Notation (vgl. [Par98]) beschriebenen Protokoll kann sie realisiert werden. Ergänzend sind im Protokoll noch persistente Sperren eingezeichnet, und zwar unterhalb der Server, von denen sie gehalten werden.

Die Semantik der Aufrufe entspricht der von synchronen RPCs[29]. Jeder Aufruf beinhaltet ein Acknowledge, d.h. eine Bestätigung darüber, daß der Auftrag angekommen und in eine Queue gestellt worden ist. Die anschließende Auftragsdurchführung geschieht asynchron. Jeweils am Ende einer Prozedur wird der Folgeauftrag ausgelöst.

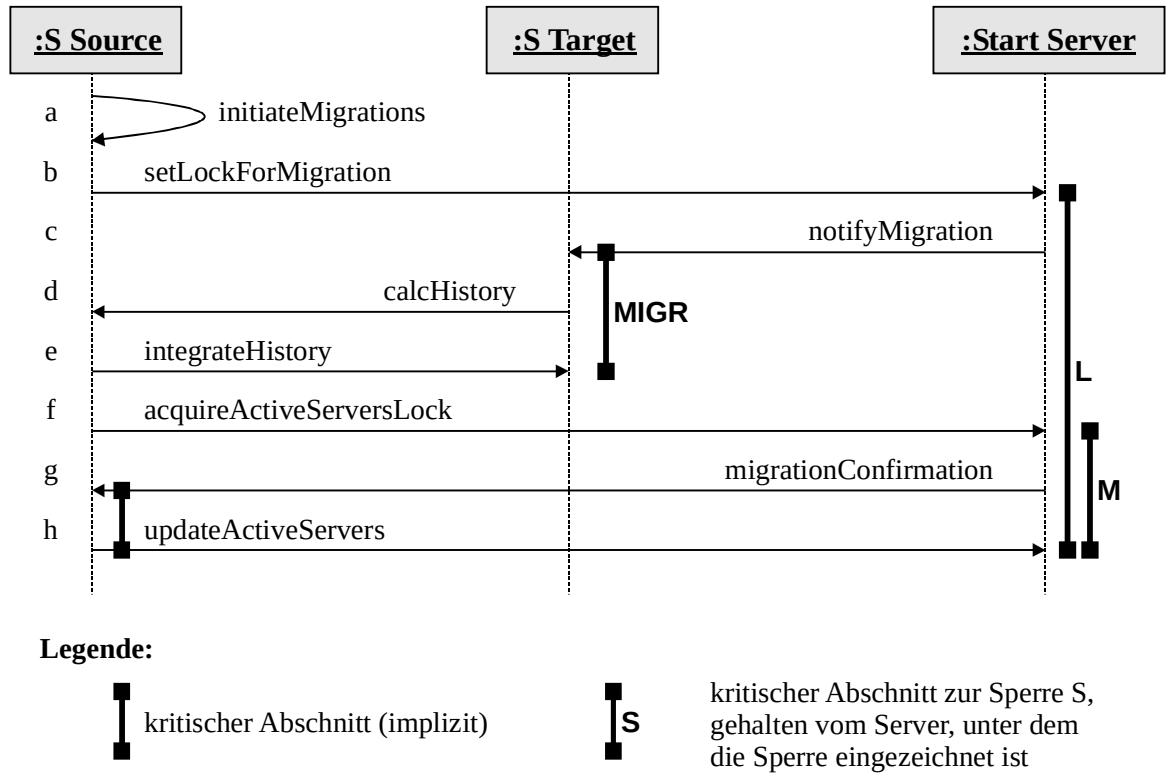


Abbildung 5-9: Protokoll zur Durchführung einer Migration.

Es folgen kurze Erläuterungen zum Protokoll. Sie beziehen sich immer auf die Prozedur, die wegen des zum angegebenen Zeitpunkt abgesandten Auftrages zu bearbeiten ist (z.B. a: *initiateMigrations*).

---

[28] Die UML (Unified Modeling Language) ist eine standardisierte Notation zur Beschreibung von statischen und dynamischen Aspekten objektorientierter Systeme. Sie vereint Aspekte verbreiteter Formalismen zur Spezifikation, Visualisierung und Dokumentation von OO-Systemen.

[29] RPC = Remote Procedure Call, entfernter Aufruf einer Prozedur, die auf einem anderen Rechner ausgeführt wird und deren Ergebnis an den Aufrufenden übermittelt wird (genaueres siehe [CDK95]).

a Die Migrationsquellaktivität auf S_{Source} wurde als beendet markiert, und der Migrationszielserver ist bekannt. Die Voraussetzungen zum Beginn der Migration sind somit erfüllt. Die Prozedur initiateMigrations wird auch dann ausgelöst, wenn für eine bislang wartende Migration (Kante im Migrations-Status CHECKMIGRATION[30]) eine Nachricht über den Zielserver eintrifft und in Folge dessen migriert werden muß. S_{Source} setzt den Zustand der Migrationskante auf MIGRATING.

b Um den gegenseitigen Ausschluß von Migrationen und der Schreibphase von dynamischen Änderungen zu erreichen (vgl. Abschnitt 5.5.1), muß beim Startserver eine Lesesperre L angefordert werden. Die zu L zugehörige Schreibsperre W wird im Rahmen des Protokolls zur Durchführung einer dynamischen Änderung beschrieben (siehe Abschnitt 5.5.5). Im Anschluß daran stößt der Startserver die Migration an. Im Vergleich zu der Variante, daß der Anstoß durch den Quellserver erfolgt, wird ein Kommunikationsschritt eingespart, weil sonst der Startserver den Quellserver benachrichtigen müßte, wenn die Sperre gewährt wurde.

c Auf S_{Target} wird eine Sperre MIGR[31] zum gegenseitigen Ausschluß von gleichzeitig eingehenden Migrationen (derselben Prozeßinstanz) beantragt. Sobald diese Sperre gewährt wird, kann er eine Migration nach dem Holen-Protokoll einleiten (vgl. Kapitel 4). Wenn der Server die Prozeßinstanz schon kennt, müssen Migrationspunkte berechnet werden. Sie werden, soweit vorhanden, im Rahmen des Folgeauftrags *calcHistory* übertragen.

d Der Quellserver berechnet anhand der erhaltenen Information, ob und welche Daten er zum Zielserver übertragen muß und überträgt diese im Rahmen des Folgeauftrags *integrateHistory* (eigentliche Migration der Daten).

e Beim Zielserver wird anhand der dort vorhandenen Template-Information eine Prozeßinstanz erstellt, sofern diese dort noch nicht existiert. Anschließend werden die erhaltenen Daten eingespielt und der Instanzausführungsgraph entsprechend der neuen Änderungs- und Ausführungshistorieneinträge angepaßt. S_{Target} setzt den Zustand der Migrationskante auf MIGRATED, hebt die Sperre MIGR auf und kann mit der Ausführung fortfahren.

f Um anschließend die Menge *ActiveServers* korrekt zu aktualisieren, muß gewährleistet sein, daß während dieser Zeit kein anderer Server das Gleiche tut. Daher wird vom Startserver die exklusive Sperre M angefordert. Der dadurch gegebene kritische Abschnitt verhindert, daß die auf Seite 84 unter Variante 2 beschriebenen Probleme entstehen können.

g Die Migration zum Zielserver wurde erfolgreich durchgeführt. Da *migrationConfirmation* nur ausgeführt werden kann, nachdem auf dem Startserver die Sperre M gewährt wurde, befindet sich auch auf dem Quellserver ein (impliziter) kritischer Abschnitt. Er ist notwendig zur korrekten Berechnung der Entscheidung, ob S_{Source} aus *ActiveServers* entfernt werden soll („Log_off“ oder „Stay“). Dem Startserver muß die Entscheidung mitgeteilt werden, der Zielserver S_{Target} wird immer in die Menge *ActiveServers* aufgenommen. S_{Source} setzt den Zustand der Migrationskante auf MIGRATED.

h Der Startserver aktualisiert die Menge *ActiveServers* und gibt anschließend die Sperren M und L wieder frei.

Im Protokoll nicht eingezeichnet ist die Möglichkeit, eine begonnene Migration zu Beginn von b und c mit *cancelMigration* abzubrechen. In diesem Fall werden alle bisher für diese Migration angeforderten Sperren wieder freigeben und der Migrations-Status der Migrationskante wieder auf MIGRATION gesetzt. Eine Referenz zu den Zustandsübergängen dieses Kantenattributs findet sich im Anhang A.

---

[30] Siehe hierzu auch Anhang A.

[31] Die Sperre MIGR hat nichts mit der Berechnung von *ActiveServers* zu tun. Sie ist auch in einem entsprechenden Protokoll eines Systems, das keine dynamischen Änderungen ermöglicht, notwendig.

Fehlerbehandlung

Um zusätzlichen Aufwand im Fall von Fehlern zu sparen, könnte man ein Two-Phase-Commit-Protokoll [Dad96] einsetzen. Darüber hinaus muß das parallele Schreiben eines Objektes durch unterschiedliche Transaktionen möglich sein, da sonst im Rahmen der Transaktion keine Shared-Sperren (hier: Sperre L) realisiert werden können. Die Migrationstransaktion würde dann in Schritt a von S_{Source} begonnen und in Schritt h vom Startserver abgeschlossen. Beim Ausfall eines Servers während des Protokolls wird die komplette Transaktion zurückgesetzt.

Steht kein 2PC zur Verfügung, muß im Fehlerfall ein ausgefallener Server nach dem Wiederanlauf, ausgehend von seinem letzten Zustand, die erforderlichen Aktionen erneut durchführen. Es wird davon ausgegangen, daß der ursprüngliche Auftrag persistent gespeichert wurde und somit vorhanden ist. Die zur Auftragsbearbeitung gehörende Transaktion, bei deren Ausführung der Server ausgefallen ist, wurde zurückgesetzt, so daß die Bearbeitung nochmals korrekt von vorne begonnen werden kann. Wegen der Anforderung persistenter Sperren ist darauf zu achten, daß eine At-Most-Once Semantik der Nachrichtenübertragung gewährleistet ist. Ansonsten würde endlos gewartet, wenn eine exklusive Sperre zweimal angefordert wird. Bei nicht exklusiven Sperren würde ohne zusätzliche Maßnahmen der Zähler nicht mehr korrekt arbeiten.

Während der Startserver ausgefallen ist, können keine Migrationen bei den dort gestarteten WF-Instanzen durchgeführt werden. Von einem Ausfall eines Migrationsquell- oder -zielservers sind nur die Migrationen von bzw. zu diesem Server betroffen, die sich gerade in der Durchführung befanden. Ebenfalls vorübergehend blockiert sind Migrationen zu einem Server, auf dem eine vom abgestürzten Quellserver angeforderte Sperre MIGR gewährt wurde. An allen anderen Stellen können die betroffenen Prozeßinstanzen in der Ausführung fortgesetzt werden. Falls ein Quellserver S_Q, der bereits eine Sperre M angefordert hat, abstürzt, kann bis zur Freigabe der Sperre nach dem wieder Hochfahren von S_Q keine Aktualisierung von *ActiveServers* durchgeführt werden. Dies stellt jedoch keine wesentliche keine Beeinträchtigung für die Ausführung dar. Dynamische Änderungen sind aufgrund der Lesesperre L während dem Ausfall jedoch nicht möglich. Diese müßten aber ohnehin warten, bis der ausgefallene Server wieder verfügbar ist, da Server, von denen eine Änderung ausgeht, zuvor Ihre Zustandsinformation aktualisieren müssen (siehe dazu auch Abbildung 5-11).

Weitere Informationen zur Fehlerbehandlung finden sich in Kapitel 6 (Implementierung) in den Abschnitten 6.3.3 und 6.4.3. Dort wird eine Variante vorgestellt, welche mit nicht persistenten Sperren auskommt, die jedoch nicht mehr in allen Fehlerfällen eine automatische Fortführung der betroffenen WF-Instanzen ermöglicht.

Optimierungen

Nun zur Lösung der zweiten Aufgabe, der Schaffung einer Möglichkeit zur Feststellung, ob sich die Menge der aktuell beteiligten Server möglicherweise geändert hat. Hierfür kann ein Verfahren mit einem Timestamp verwendet werden (vgl. [RBD98]). Bei jeder Migration wird auf dem Startserver der Timestamp erhöht und dieser anschließend dem Migrationsquellserver mitgeteilt. Ein beliebiger Server S, der die Menge *ActiveServers* erfragt, erhält den aktuellen Timestamp mitgeliefert. Kommuniziert jetzt der Server S mit einem Server A aus der Menge *ActiveServers*, können deren zwei Timestamps verglichen werden. Hat der Timestamp von A einen höheren Wert als derjenige von S, so hat zwischenzeitlich von A ausgehend eine weitere Migration stattgefunden und der bei S vorhandene Wert von *ActiveServers* ist veraltet. Die Menge *ActiveServers* muß erneut erfragt werden, und, falls eine zusammengehörige Reihe von Aufrufen an alle aktuell beteiligten Server unterbrochen wurde, müssen die Aufrufe jetzt für die bisher noch nicht berücksichtigten Server fortgesetzt werden.

Die Menge *ActiveServers* kann durch einen einfachen Aufruf von get*ActiveServers*[32] beim Startserver erfragt werden. Häufig ist aber auch eine Optimierung möglich, z.B. falls sich die Ausführung innerhalb einer Sequenz befindet. In dem Fall, daß nur ein Server an der Ausführung von Aktivitäten

[32] Siehe Abbildung 5-11, die hier verwendete Funktion getActiveServers ist isoliert vom dort beschriebenen Protokoll betrachten (ohne Folgeaufruf, lediglich Übermittlung des Ergebnisses).

einer parallelen Verzweigung beteiligt ist, kann die Abfrage unter Umständen entfallen. Ein Server kann aber nur dann feststellen, daß er zur Zeit der einzige ist, wenn er entweder genauso viele Aktivitäten kontrolliert wie parallele Zweige in den geöffneten Verzweigungen[33] enthalten sind oder er sicher weiß, daß die letzten Aktivitäten aller auf die Anzahl noch fehlenden Zweige beendet wurden.

Bei dieser Optimierung wird der Server des Joinknoten manchmal noch nicht zu *ActiveServers* dazugezählt, auch wenn schon einige Zweige zu ihm migriert sind. Der Fall tritt genau dann ein, wenn momentan nur ein einziger Server Aktivitäten aus der parallelen Verzweigung kontrolliert und der Joinknoten einem anderen Server zugeordnet ist. Dieser andere Server wird infolgedessen bei einer anstehenden Aktualisierung des Graphen nicht berücksichtigt. Dies stellt jedoch kein Problem dar, weil die dort noch fehlenden Informationen bei kommenden Migrationen zum Joinknoten übertragen werden. Vorher kann die Joinaktivität nicht ausgeführt werden, da noch nicht alle eingehenden Kanten signalisiert sind.

Algorithmus zur optimierten Bestimmung der aktuell an der Ausführung beteiligten Server

```
output
   currentServers ∈ P(S): Die aktuell an der Ausführung beteiligten Server
begin
   server = aktueller Server;
   activitiesS = vom Server server kontrollierte Aktivitäten;
   j = outermostJoinAct(n);
   if (j = NULL) then
      currentServers = {server}; // keine parallele Verzweigung
      return currentServers;
   completedActCount = 0;
   openBranchCount = #incomingControlEdges(j);
   s = correspondingSplitActivity(j);
   for each ( si | si ∈ c_succ*(s) ∧ (branchOutType(si) = ALL_Of_ALL) ∧
                    state(si) ∈ {COMPLETED, CANCELLED, FAILED} ) do
      ji = correspondingJoinActivity(si);
      // alle geöffneten parallelen Verzweigungen überprüfen
      if ( state(ji) = NOT_ACTIVATED ) then
         openBranchCount = openBranchCount + #incomingControlEdges(ji) - 1;
         // die Aktivitäten am Ende der parallelen Zweige zählen, die schon beendet sind
         for each ( a ∈ c_pred(ji) ) do
            if ( state(a) ∈ {COMPLETED, CANCELLED, FAILED} ∧ a ∉ activitiesS ) then
               completedActCount = completedActCount +1;
   if ( #activitiesS + completedActCount = openBranchCount ) then
      currentServers = {server};
   else
      currentServers = getActiveServers(); // aktuelle Server müssen explizit erfragt werden[32]
end
```

Algorithmus 5-1: Optimierte Bestimmung der aktuell an der Ausführung beteiligten Server.

5.5.3 Einholen von Zustandsinformation

Um zu überprüfen, ob eine dynamische Änderung einer WF-Instanz möglich ist, muß deren aktueller Zustand bekannt sein. Ist sichergestellt, daß alle Aktivitäten in den Teilzweigen vom gleichen Server kontrolliert werden, so ist dies bereits gewährleistet. Wenn mehrere Server an der Ausführung beteiligt

[33] Mit einer geöffneten parallelen Verzweigung ist eine Verzweigung gemeint, deren Splitknoten bereits durchgeführt wurde und deren Joinknoten noch nicht aktiviert werden konnte. Diese Betrachtungsweise berücksichtigt auch verschachtelte parallele Verzweigungen.

sind, bedeutet dies, von diesen Servern Informationen einzuholen. Hierfür bieten sich zwei unterschiedliche Verfahren an, die anhand des folgenden Beispiels vorgestellt werden sollen.

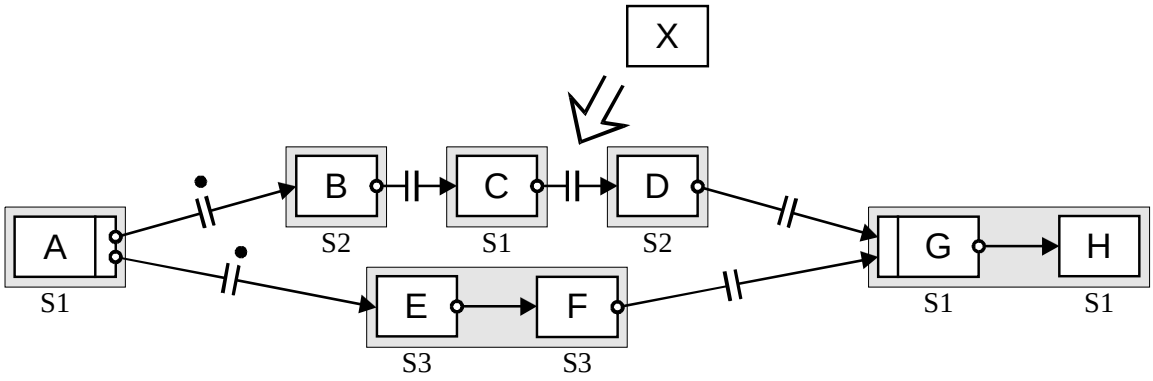


Abbildung 5-10: Synchronisation einer dynamischen Änderung.

Im Graph aus Abbildung 5-10 soll die Aktivität X zwischen C und D eingefügt werden. Die Änderung geht vom Bearbeiter der Aktivität E im anderen Teilzweig aus. Für diese Modifikation wird an den Ausführungszustand der Instanz vorausgesetzt, daß sich die Aktivität D nur im Zustand NOT_ACTIVATED oder ACTIVATED befinden darf. Zwei unterschiedliche Vorgehensweisen bieten sich hier zur Feststellung an.

a) Von allen aktuell beteiligten Servern wird die WF-Kontrollinformation geholt und damit der eigene Zustand aktualisiert. Zur Vermeidung unnötiger Kommunikation kann nach einem ähnlichen Protokoll wie bei einer Migration vorgegangen werden, mit der Ausnahme, daß jetzt keine Quell- und Zielaktivitäten feststehen. Außerdem müssen die Migrationspunkte in feineren Einheiten angegeben werden, zusätzlich zu Knoten-ID und Iterationsnummer kommt jetzt die letzte Zustandsänderung. Bisher wurden lediglich Migrationspunkte zu „COMPLETED"-Historieneinträgen übertragen, so daß diese Angabe entfallen konnte. Der Änderungsserver S3 berechnet mit Algorithmus 3-3 die Migrationspunkte M und teilt diese zunächst dem am günstigsten erreichbaren Server aus *ActiveServers* mit. Daraufhin überträgt dieser die bei ihm vorhandenen Nachfolgern von M entsprechenden Ausführungshistorienabschnitte an S3, wo die Information sogleich eingearbeitet wird. Im Anschluß werden die Migrationspunkte neu berechnet und mit dem nächstgünstigeren Server fortgefahren, bis von allen aktuell beteiligten Servern die Information geholt wurde.

   Mit Hilfe des aktuellen Gesamtzustandes der Instanz kann dann ein herkömmlicher Test, ob die Änderung zulässig ist, durchgeführt werden. Der Algorithmus aus dem zentralen Fall kann unabhängig von der gewählten Änderungsoperation wiederverwendet werden. Wenn der Änderungsserver später nicht mehr an der Ausführung beteiligt ist, wurden ihm auf diese Weise Informationen übermittelt, die dort größtenteils nicht gebraucht werden. Mit einer anderen im Anschluß vorgestellten Vorgehensweise kann das vermieden werden.

b) Nur über die Aktivitäten, deren Zustände für die gewählten Änderungen zu überprüfen sind, werden Informationen geholt. Die Informationen können dann so eingearbeitet und in der Datenbank gespeichert werden, daß die Algorithmen aus dem zentralen Fall anwendbar bleiben. Dieses Verfahren wirkt zwar unsauber, da unter Umständen „Lücken"[34] im Graphen entstehen, dafür müssen zunächst weniger Daten übertragen werden. Falls der Server, von dem die Änderung ausgeht, später nicht mehr an der Ausführung beteiligt werden sollte, wird durch das Verfahren die unnötige Übertragung von Änderungshistorie vermieden. Anderenfalls werden später im Zuge einer eingehenden Migration wie bisher die fehlenden Änderungshistorieneinträge eingespielt und

[34] Der Zustand eines Knotens, dessen Ausführung von einem anderen Server kontrolliert wird, wird z.B. auf ACTIVATED gesetzt, obwohl dessen Vorgänger mit dem Zustand NOT_ACTIVATED markiert ist.

die „Lücken“ wieder geschlossen. In diesem Fall wurde kein Kommunikationsvolumen eingespart. Um eine zukünftige Erweiterung der Änderungsoperationen zu ermöglichen, muß jede Änderungsoperation die Menge der Knoten zur Verfügung stellen, von denen Zustandsinformationen benötigt werden.

Im Beispiel muß der Zustand des Knotens D überprüft werden. Bei statischen Serverzuordnungen könnte der Server S2, der bei Ausführung von D die Instanz kontrolliert, gefragt werden. Diese Möglichkeit entfällt bei variablen Serverzuordnungen. Hier müssen alle aktuell beteiligten Server der Reihe nach oder parallel nach Zustandsinformationen über D gefragt werden. Sollen Informationen über mehr als einen Knoten besorgt werden, so ist es selbstverständlich besser, dies innerhalb der gleichen Anfrage abzuwickeln.

Die Verwendung der optimierten Variante b) macht nur dann Sinn, wenn die Wahrscheinlichkeit, daß der die Änderung auslösende Server später noch einmal an der Ausführung des Workflow beteiligt wird, gering ist. In der Regel wird daher das Verfahren nach a) gewählt werden.

5.5.4 Sperren

Während eine dynamische Änderung durchgeführt werden soll, müssen die Daten einer WF-Instanz zeitweise gesperrt werden, um Inkonsistenzen zu verhindern. Der gegenseitige Ausschluß zwischen Migrationen und dynamischen Änderungen (vgl. Abschnitt 5.5.1) reicht nicht aus, da sich der Zustand einer Instanz auch bei der Ausführung von Aktivitäten ändert. Ein Problem tritt z.B. dann auf, wenn eine Aktivität beendet wird und anschließend die Ausgangskanten markiert werden, dabei aber gleichzeitig dieser Aktivität eine weitere ausgehende Kante hinzugefügt werden soll. Das Ausführungsergebnis, d.h. ob der neue Zweig sofort durchlaufen wird, ist zufällig. Da das Beenden einer Aktivität und das Durchführen einer dynamischen Änderung in eigenen Transaktionen durchgeführt werden, sorgt hier der Transaktionsschutz des DBMS dafür, daß solche Fehler ausgeschlossen werden.

Wenn keine Sperren benutzt werden, kann nicht garantiert werden, daß zum Zeitpunkt, an dem die Änderungsoperation in die Datenbank eingebracht wird, die Ausführung des Workflow auf einem der Server nicht schon so weit fortgeschritten ist, daß die Änderung nicht mehr möglich ist. Die Sperren müssen daher auf allen aktuell beteiligten Servern aktiviert werden.

Es bieten sich zwei verschiedene Varianten für Sperren zum Verhindern des Fortschreitens des WF an. Beiden gemeinsam ist, daß sie das Starten von Aktivitäten und das Markieren der Ausgangskanten verhindern. In einem Fall gilt die Sperre für alle Aktivitäten einer Instanz, im anderen Fall nur für bestimmte Aktivitäten.

1. Sperre für alle Aktivitäten einer Instanz

   Eine solche Sperre ist sinnvoll, wenn sie nur relativ kurz im System verbleibt. Durch sie wird das Fortschreiten der Ausführung der Instanz unterbunden. Pro Instanz ist nur eine Sperre notwendig. Falls Migrationen nicht schon mit einem anderen Mechanismus gesperrt wurden, d.h. insbesondere der Sperre zum gegenseitigen Ausschluß von Migrationen und dynamischen Änderungen , werden jetzt gleichzeitig auch alle Migrationen verhindert.

   Im Beispiel aus Abbildung 5-10 wird die abgebildete Prozeßinstanz bei den Servern S2 und S3 wie beschrieben gesperrt, es ist jetzt nur mehr das Beenden von Aktivität E möglich.

2. Sperren für bestimmte, vom Änderungsserver berechnete, Aktivitäten

   Die Sperre betrifft nur einen Teil der Aktivitäten, und zwar genau diejenigen vor dem berechneten Änderungsbereich. Für die gewählte Änderungsoperation unkritische Aktivitäten bleiben von der Sperre ausgenommen. Da das Fortschreiten in der Ausführung des Workflow weiterhin eingeschränkt möglich ist, können solche Sperren länger verbleiben als die nach der ersten Methode behandelten. Sie können schon in der Vorbereitungsphase einer Änderung gesetzt werden, damit ausgeschlossen wird, daß die Änderung wegen des Fortschreitens in einem parallelen Zweig aufgrund von Zustandskonflikten undurchführbar wird. So werden dynamische Änderungen gegenüber der Ausführung bevorzugt behandelt. Es müssen mehrere Sperrsätze existieren können, sonst können nicht gleichzeitig mehrere Änderungen vorbereitet werden.

Bei diesem Verfahren sind auch bei vorhandenen Sperren weiterhin Migrationen möglich. Migrationen und dynamische Änderungen einer WF-Instanz dürfen gleichzeitig stattfinden. Daher sind zwei Dinge zu berücksichtigen: Es muß sichergestellt sein, daß die Sperren überall ankommen. Zunächst wird an alle Server aus *ActiveServers* eine Sperranforderung geschickt. Falls danach noch weitere Server in die Menge *ActiveServers* aufgenommen wurden, müssen auch diesen Servern noch die Sperren geschickt werden, solange bis *ActiveServers* unverändert ist. Es ist auch möglich, daß das Setzen einer Sperre fehlschlägt, da die zu sperrende Aktivität selbst schon zu weit fortgeschritten ist. Daher muß in jedem Fall eine Bestätigung an den Erzeuger der Sperre verschickt werden, ob die Sperre gesetzt werden konnte.

Die Sperren gehören zum Zustand der Instanz und müssen bei Migrationen weitergegeben werden. Dabei muß entweder der Sperrenerzeuger davon in Kenntnis gesetzt werden, damit die Sperren später wieder aufgehoben werden können, oder die Weitergabe protokolliert werden, so daß ein Request zum Aufheben der Sperre „nachgesendet“ werden kann.

Für die Änderungsoperation aus Abbildung 5-10 genügt es, das Markieren der Kanten nach Ausführung der Aktivität C zu sperren, das Starten von C ist erlaubt. Die Sperre wird allen Servern aus *ActiveServers* mitgeteilt. Es würde sogar ausreichen, nur die Server der betroffenen Zweige zu informieren, d.h. in diesem Fall S2. Da aber ohnehin zu allen aktuell beteiligten Servern eine Kommunikation nötig wird[35], wird die Sperre an S2 und S3 geschickt. Daraus entsteht kein Problem, da die Sperre im Teilzweig von S3 nicht wirksam wird. Bei der Migration $M_{B,C}$ muß die Sperre an S1 weitergegeben werden und S3 darüber informiert werden.

Wenn die Ausführung des Workflow gegenüber dynamischen Änderungen vorrangig behandelt wird, kommt man mit dem ersten, bedeutend einfacheren Verfahren aus. Lediglich in der relativ kurzen Durchführungsphase der Änderung, in der keine Benutzerinteraktion mehr stattfindet, muß gesperrt werden. So wird die Beeinträchtigung, d.h. die Verzögerung von Migrationen durch die Sperren, für die Server von Aktivitäten aus parallelen Zweigen nur noch in wenigen Fällen feststellbar.

Bei der zweiten Variante kommt noch hinzu, daß die Information zur Ausbreitung von dynamischen Änderungen unter Umständen ebenfalls „nachgesendet“ werden muß, da währenddessen weiterhin Migrationen möglich sind.

Im folgenden soll daher die Variante 1 als Grundlage genommen werden.

5.5.5 Durchführung einer Änderung

Im UML Sequenzdiagramm aus Abbildung 5-11 wird eine Möglichkeit gezeigt, wie eine dynamische Änderung im verteilten Fall durchgeführt werden kann. Ergänzend sind im Protokoll noch persistente Sperren eingezeichnet, und zwar bei den Servern, von denen sie gehalten werden (Legende: siehe Abbildung 5-9).

Die Semantik der Aufrufe entspricht der von RPCs. Jeder Aufruf beinhaltet ein Acknowledge, d.h. eine Bestätigung daß der Auftrag angekommen und in eine Queue gestellt worden ist. Die anschließende Auftragsdurchführung, d.h. die Abarbeitung einer Prozedur mit dem selben Namen wie die erhaltene Message, geschieht asynchron. Jeweils am Ende einer Prozedur wird der Folgeauftrag ausgelöst.

Die Änderungsoperation besteht aus zwei Phasen. In der ersten Phase wird die am Server vorhandene Zustandsinformation aktualisiert und die Änderung vorbereitet. Dabei werden noch keine Sperren gesetzt.

Ist die Durchführung der Änderung nach dem vorhandenen Informationsstand durchführbar und entscheidet sich der Benutzer, die Änderung durchzuführen, wird Phase 2 eingeleitet. Hier werden Sperren gesetzt, damit die eingeholte Zustandsinformation garantiert nicht veralten kann. Steht der Änderung auch jetzt nichts im Wege, wird sie auf allen aktuell an der Ausführung beteiligten Servern durchgeführt und die Sperren aufgehoben.

---

[35] Aus dem gleichen Grund wie in Abschnitt 5.5.3 b): Zunächst ist nicht bekannt, welcher der Server für einen bestimmten Zweig zuständig ist.

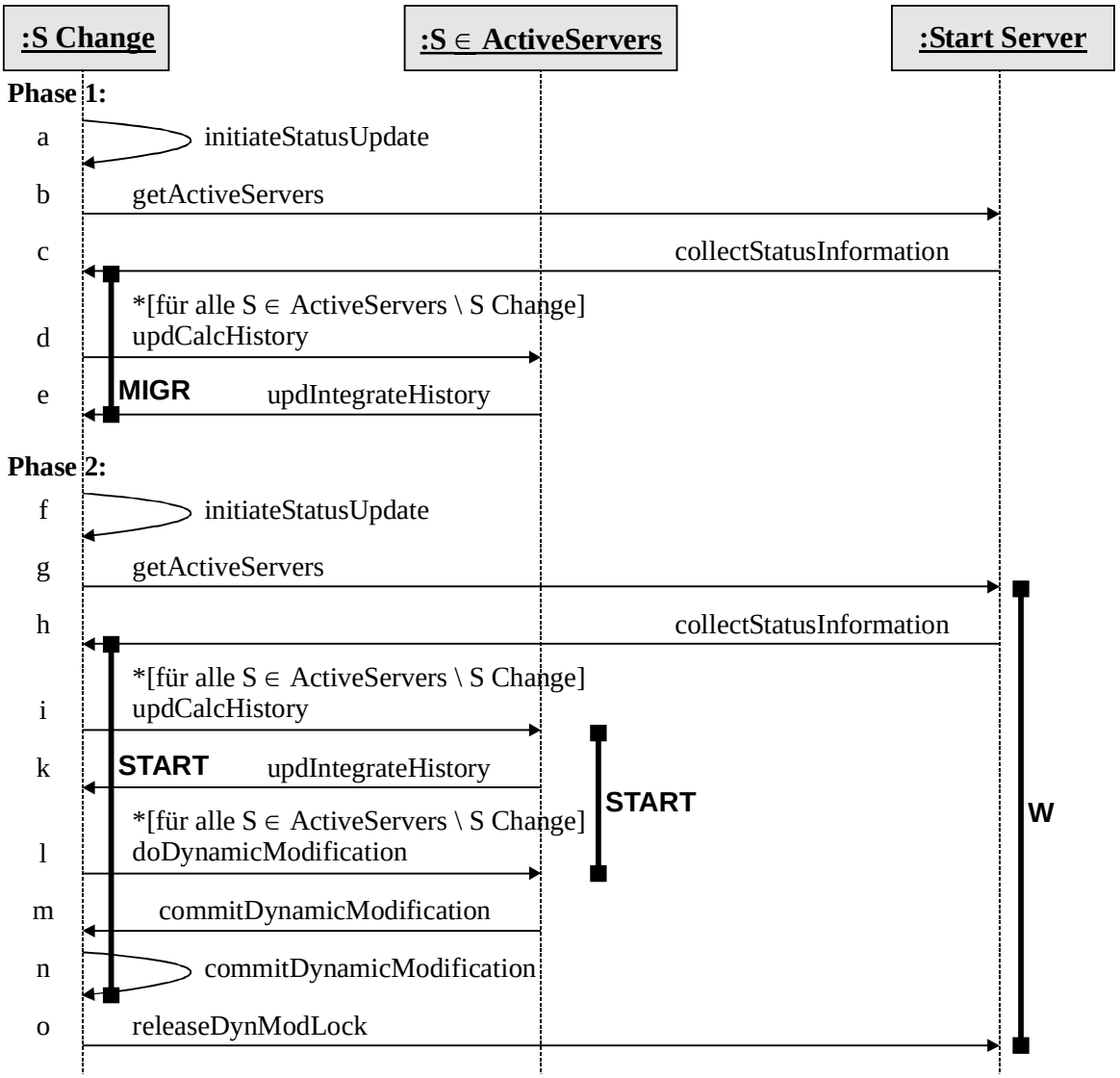


Abbildung 5-11: Protokoll zur Durchführung einer dynamischen Änderung.

Phase 1:

a Ein Benutzer gibt an, daß er eine Änderung durchführen möchte. Damit löst er eine Aktualisierung der Zustandsinformation durch *initiateStatusUpdate* aus.

b Die Menge *ActiveServers* wird erfragt.

c Der StartServer stößt beim Server S_{Change} das Einholen von Zustandsinformation an. Zuerst wird auf S_{Change} eine Sperre MIGR angefordert, um gleichzeitig eingehende Migrationen für die Instanz zu verhindern und danach die Migrationspunkte berechnet. Im Anschluß wird nacheinander von allen aktuell an der Ausführung beteiligten Servern Zustandsinformation geholt.

d Der Server bekommt die zuvor neu berechneten Migrationspunkte übermittelt und kann daraufhin berechnen, welche Ausführungshistorienabschnitte er übertragen muß.

e S_{Change} spielt die neu erhaltenen Historieneinträge bei sich ein. Wenn er noch nicht die Information von allen Servern aus *ActiveServers* eingeholt hat, berechnet er die Migrationspunkte neu und fährt noch einmal mit d fort. Anderenfalls ist die Aktualisierung abgeschlossen, die Sperre MIGR kann freigegeben werden und der Benutzer kann mit der Vorbereitung der Änderungsoperation beginnen.

Phase 2:

f Ein Benutzer gibt an, daß er eine Änderung endgültig durchführen möchte. Damit löst er eine Aktualisierung der Zustandsinformation durch *initiateStatusUpdate* aus.

g Die Menge *ActiveServers* wird erfragt, nachdem eine Schreibsperre W zum gegenseitigen Ausschluß von Migrationen und dynamischen Änderungen und auch dynamischen Änderungen untereinander gesetzt wurde. Die zugehörige Lesesperre L kann dem Protokoll zur Durchführung von Migrationen entnommen werden (siehe Abschnitt 5.5.2).

h Der StartServer stößt beim Server S_{Change} das Einholen von Zustandsinformation an. Zuerst wird auf S_{Change} eine Sperre START zum Verhindern des Fortschreitens bei der Ausführung beantragt und danach die Migrationspunkte berechnet. Im Anschluß wird nacheinander von allen aktuell an der Ausführung beteiligten Servern Zustandsinformation geholt.

i Der Server bekommt die zuvor neu berechneten Migrationspunkte übermittelt und kann daraufhin nach dem erfolgreichen Setzen einer Startsperre START berechnen, welche Ausführungshistorien-abschnitte er übertragen muß.

k S_{Change} spielt die neu erhaltenen Historieneinträge bei sich ein. Wenn er noch nicht die Information von allen Servern aus *ActiveServers* eingeholt hat, berechnet er die Migrationspunkte neu und fährt noch einmal mit i fort. Anderenfalls ist die Aktualisierung abgeschlossen und die Änderungsoperation kann durchgeführt werden. Läßt sich die Änderung durchführen, kann sie dauerhaft in die Datenbank eingebracht und auf die anderen aktiven Server ausgebreitet werden.

l Die Änderung wird endgültig in die Datenbank eingebracht und die Sperre START wieder aufgehoben.

m Von allen Servern, die die Änderung bei sich einbringen, wird eine Bestätigung erwartet.

n Wenn die letzte Bestätigung eintrifft, kann auch der Server S_{Change} seine Sperre START wieder freigeben.

o Die Sperre W wird freigegeben. Falls durch die Änderung ein Server neu an der Ausführung beteiligt wird, muß er jetzt in die Menge *ActiveServers* aufgenommen werden.

Im Protokoll nicht eingezeichnet ist die Möglichkeit, eine begonnene Dynamische Änderung zu Beginn von g und i mit *cancelDynamicModification* abzubrechen. In diesem Fall werden alle bisher für diese Operation angeforderten Sperren wieder freigeben.

Um zusätzlichen Aufwand im Fall von Fehlern zu sparen, könnte man ein Two-Phase-Commit-Protokoll einsetzen. Die Änderungstransaktion würde dann in Schritt f von S_{Change} begonnen und in Schritt o vom Startserver abgeschlossen. Steht kein 2PC zur Verfügung, muß S_{Change} bzw. der Startserver im Fehlerfall ausgehend von seinem letzten Zustand die erforderlichen Aktionen erneut durchführen. Wenn eine Änderung bei der letzen Überprüfung als durchführbar gilt, kann verlangt werden, daß die anderen an der Ausführung beteiligten Server diese bei sich einbringen, nachdem der Hinderungsgrund abgestellt wurde. Beim Anfordern der persistenten Sperren des Startservers ist darauf zu achten, daß eine At-Most-Once Semantik der Nachrichtenübertragung gewährleistet ist. Ansonsten würde endlos gewartet, wenn eine exklusive Sperre zweimal angefordert wird.

5.5.6 Beispiel

Im Beispiel in Abbildung 5-12 werden einzelne Stufen im Fortschreiten des Protokolls zur Durchführung einer dynamischen Änderung gezeigt.

Ein an Server S1 angemeldeter Bearbeiter möchte die Aktivität X nach Aktivität C und vor Aktivität D einfügen. Der Server S1 kennt bisher nur die in 1.) angedeutete Zustandsinformation, daher muß er eine Aktualisierung einleiten und sich dafür zunächst vom Koordinator die Menge *ActiveServers* geben lassen. Anschließend kann er seine Migrationspunkte (hier: „E, RUNNING“) an S2 schicken, damit ihm dieser die noch fehlenden Ausführungshistorieneinträge schicken kann. Dies ist hier lediglich der Historieneintrag zu „B, ACTIVATED“, der bei S1 eingearbeitet wird (vgl. 2.)). Der Benutzer kann jetzt die Änderung incl. der Folgeänderungen vorbereiten.

Bevor die Änderung endgültig durchgeführt werden kann, muß S1 nach dem Anfordern der Sperre W beim Koordinator erneut seine Zustandsinformation auf den neuesten Stand bringen (siehe 3.)). Da in der Zwischenzeit die Migration $M_{B,C}$ stattgefunden hat, befindet sich jetzt der Server von C, in diesem Fall S3, an Stelle von S2 in der Menge *ActiveServers*. S1 bekommt die benötigte Historieninformation von S3, der ebenso wie S1 eine Sperre zum Verhindern des Fortschreitens in der Ausführung setzt.

Der aktuelle Zustand läßt die Änderung noch zu, die jetzt durchgeführt werden kann. Die Sperren werden im Anschluß aufgehoben. Nach der Änderung entspricht der bei S1 vorhandene Zustand des Graphen dem in 4.) dargestellten. S3 hingegen kennt jetzt zwar die neue Aktivität X, nicht jedoch den Zustand von S1, d.h. der Aktivitäten E, F und G. Seine beiden letzten Ausführungshistorieneinträge sind „C, RUNNING“ und „DDM(1234)“.

S2 kennt die neue Graphstruktur noch nicht, da er zum Zeitpunkt der Änderung nicht mehr an der Ausführung beteiligt war. Sein letzter Ausführungshistorieneintrag ist „B, COMPLETED“.

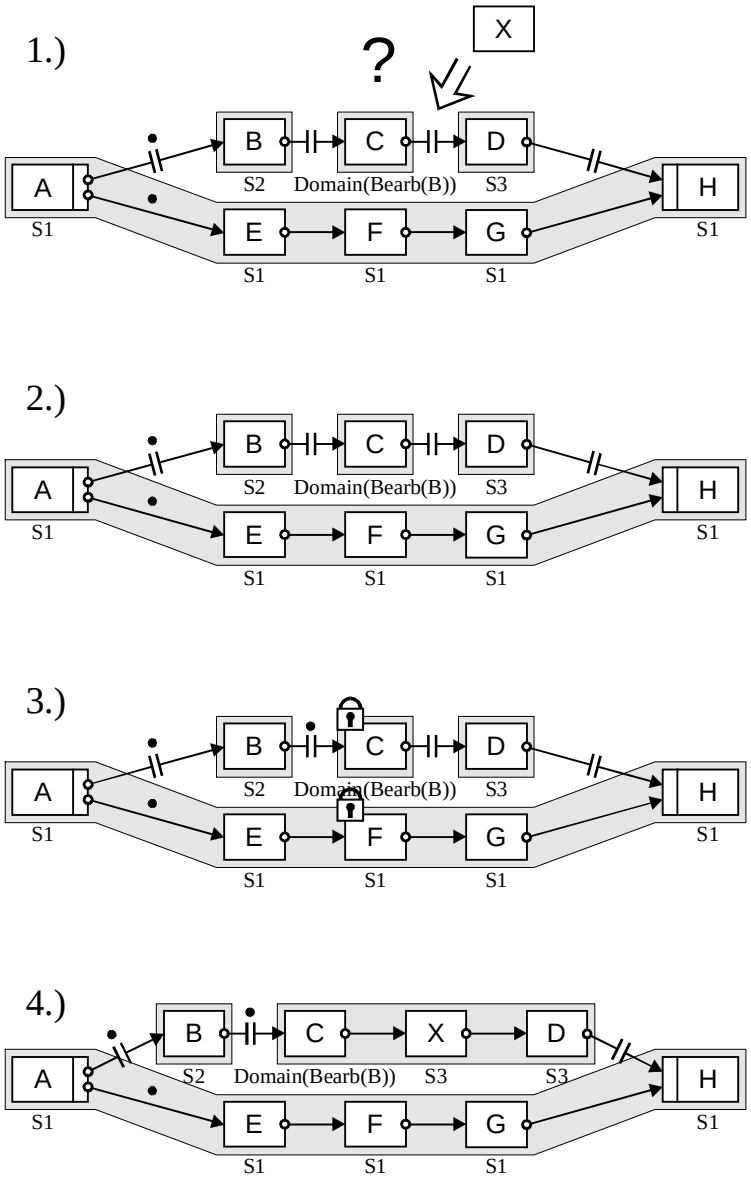


Abbildung 5-12: Ablauf einer exemplarischen dynamischen Änderung

5.6 Serverzuordnungen und dynamische Änderungen

Bei Verwendung variabler Serverzuordnungsausdrücke ergeben sich zusätzliche Problemstellungen im Zusammenhang mit dynamischen Änderungen. So muß einer neu eingefügten Aktivität ein gültiger und möglichst günstiger Serverzuordnungsausdruck zugewiesen werden. Serverzuordnungen, die sich auf eine gelöschte Aktivität beziehen, können nicht mehr ausgewertet werden. Idealerweise wird bereits im Rahmen des Löschens dafür gesorgt, daß dieses Problem bei der Ausführung nachfolgender Aktivitäten nicht auftreten kann. Für die beiden genannten Änderungsoperationen werden im folgenden mehrere Lösungsansätze zu den möglichen Problemen skizziert.

5.6.1 Einfügen von Aktivitäten

Der Serverzuordnungsausdruck einer neu eingefügten Aktivität N und eventuell innerhalb der Änderungsoperation erzeugter Split- und Joinknoten ist dann zulässig, wenn er ausgewertet werden kann. Dies ist der Fall, wenn der Server der neuen Aktivität statisch zugeordnet wird oder sich der Zuordnungsausdruck auf eine vor N garantiert ausgeführte Vorgängeraktivität bezieht. Um möglichst wenig zusätzliche Migrationen notwendig zu machen, sollte der Server einer direkten Vorgänger- oder Nachfolgeraktivität von N (über Kontroll- und Sync-Kanten) gewählt werden. Ein noch besseres Ergebnis bezüglich der Kommunikationskosten kann erzielt werden, wenn die potentiellen Bearbeiter von N berücksichtigt werden. Mögliche Verfahren zur Festlegung des Serverzuordnungsausdrucks sind:

1. Der Ausdruck der neuen Aktivität wird vom Initiator der Änderung vorgegeben, weil die Ausführung der Aktivität zwingend auf einem bestimmten Server erfolgen muß. Im Rahmen der Operation erzeugte leere Knoten werden automatisch dem Server einer direkten Vorgängeraktivität zugeordnet.
2. Den Aktivitäten werden Serverzuordnungen von Vorgängeraktivitäten zugeordnet. Im Falle von mehreren Vorgängern wird anhand eines beliebigen Kriteriums einer von ihnen ausgewählt. Die Zuweisung des Ausdrucks einer Nachfolgeraktivität ist nicht immer möglich, da die Möglichkeit besteht, daß sich deren variabler Serverzuordnungsausdruck auf eine Vorgängeraktivität bezieht, die nicht in der Menge der Vorgänger der neuen Aktivität enthalten ist.
3. Wie 2., jedoch wird diejenige Vorgänger- oder zulässige Nachfolgeraktivität verwendet, die zu den günstigsten Migrationskosten führt. Die Berechnung kann mit Hilfe der Kommunikationskostenmatrix durchgeführt werden. Das Ergebnis hängt neben den Verbindungskosten zwischen den Servern von der Menge der Daten ab, die in Folge einer bestimmten Entscheidung zum Server des neuen Knotens übertragen werden müssen. Darüber hinaus macht es keinen Sinn, dem neuen Knoten einen Server zuzuordnen, der in der weiteren Ausführung des Workflow nicht mehr vorgesehen ist.
4. Neben den Migrationskosten werden auch die bei der Ausführung der Aktivität entstehenden Kommunikationskosten berücksichtigt. Jetzt wird die Verteilung der Bearbeiter der neuen Aktivität zur Berechnung benötigt. Wenn mehrere zusammenhängende Aktivitäten eingefügt werden, kann es auch vorkommen, daß die gesamten Kommunikationskosten voraussichtlich niedriger sein werden, wenn eine zusätzliche Migration in Kauf genommen wird. Der gewählte Server muß dann nicht mehr zwingend der Server einer im WF-Graph schon vorhandenen Vorgänger- oder Nachfolgeraktivität sein.
5. In Folge der Änderung werden alle Serverzuordnungen von noch nicht aktivierten Aktivitäten neu bestimmt. Wegen des hohen Berechnungsaufwandes zur Laufzeit ist diese Variante jedoch nicht realistisch. Darüber hinaus wird zusätzlicher Aktualisierungsaufwand notwendig.

Die durch die neu eingefügten Aktivitäten notwendig gewordene Aktualisierung der Serverklassen ist durchzuführen.

Wenn die zur Berechnung benötigten Daten zur Verfügung stehen, ist die Variante 4. zu bevorzugen, da sie mit vertretbarem Berechnungsaufwand (günstigsten Server für einen Block von Aktivitäten bestimmen) zu einer in Bezug auf die Kommunikationskosten günstigen Lösung führt.

5.6.2 Löschen von Aktivitäten

Wenn eine Aktivität D gelöscht wird, auf die nachfolgende Aktivitäten in ihren Serverzuordnungsausdrücken Bezug nehmen, muß sichergestellt werden, daß diese Folgeaktivitäten weiterhin einem Server zugeordnet werden können. Hierfür gibt es unterschiedliche Lösungsmöglichkeiten:

1. Es werden keine besonderen Maßnahmen ergriffen. Ein Teil der Serverzuordnungsausdrücke kann unter Umständen nicht mehr ausgewertet werden. Hier greift jetzt eine Fehlerbehandlung, die in diesem Fall beispielsweise den Startserver den Instanz zuordnet. In Folge dessen ergeben sich neue, nicht erwünschte Migrationen und eine zusätzliche Belastung des Startservers der WF-Instanz. Die ursprünglich beabsichtigte günstige Verteilung geht verloren.

2. Alle Ausdrücke „Server(D)“ werden durch den Serverzuordnungsausdruck von D ersetzt. Bei den Ausdrücken mit dem Wert „Domain(Bearb(D))“ führt dieses Vorgehen nicht zum ursprünglich beabsichtigten Ergebnis. Statt dessen kann die Wahl eines Bearbeiters von D simuliert werden, indem entsprechend der Bearbeiterwahrscheinlichkeit von D zufällig ein Bearbeiter gewählt und dann der Server aus dessen Teilnetz verwendet wird. Die Ausdrücke werden durch diesen Server ersetzt.

   Die vorberechneten Serverklassen müssen angepaßt werden. Das Ersetzen der Ausdrücke muß protokolliert werden, damit es im Fall der Rücknahme der Änderung ebenfalls wieder rückgängig gemacht werden kann.

3. Eine in Bezug auf das Kommunikationsverhalten weiter verbesserte Lösung unterscheidet sich von der Variante 2. lediglich in der Art und Weise, wie die Ausdrücke „Domain(Bearb(D))“ ersetzt werden. Die Bearbeiterzuordnungen, gemäß denen die Serverzuordnungen ursprünglich berechnet wurden, werden jetzt berücksichtigt[36].

   Zunächst muß die Menge M aller Aktivitäten mit der Serverzuordnung „Domain(Bearb(D))“ bestimmt werden, in deren Vorgängermenge über Kontroll- und Sync-Kanten keine weitere Aktivität mit diesem Serverzuordnungsausdruck enthalten ist. Die Menge M stellt somit eine Teilmenge aller Aktivitäten mit der Serverzuordnung „Domain(Bearb(D))“ dar, und zwar genau die Aktivitäten, die in der Menge der Nachfolger über Kontroll- und Sync-Kanten von D (nicht aber einer anderen Aktivität mit dem Zuordnungsausdruck „Domain(Bearb(D))“) liegen. Die Zuordnungsausdrücke der Aktivitäten aus M werden durch den Serverzuordnungsausdruck von D ersetzt. Für jede Aktivität A dieser Menge werden die über Kontroll- und Sync-Kanten nachfolgenden Aktivitäten mit dem Ausdruck „Domain(Bearb(D))“ ermittelt und deren Werte auf „Domain(Bearb(A))“ geändert. Falls die Menge M nur eine Aktivität enthält, bleibt die ursprüngliche Serverklasse erhalten. Anderenfalls, wenn sich Aktivitäten unterschiedlicher paralleler Zweige erstmals auf die gelöschte Aktivität beziehen, zerfällt die Serverklasse.

Wenn das Zerfallen der Serverklasse in Kauf genommen werden kann, ist Variante 3. die günstigere Alternative. Hier beziehen sich die Aktivitäten auf den Server im Domain des Bearbeiters eines tatsächlich gewählten Schrittes statt auf den Server eines simulierten Bearbeiters der gelöschten Aktivität. Muß die Serverklasse erhalten bleiben, sollte Variante 2. gewählt werden.

---

[36] Das Ersetzen der Benutzerzuordnungen, die sich auf eine gelöschte Aktivität beziehen, wird in dieser Arbeit nicht betrachtet. In manchen Fällen muß auch, analog des Vorgehens zur Sicherstellung von Schreibern für nachfolgende Leser bei der Betrachtung des Datenfluß, auf das Löschen verzichtet werden bzw. müssen Folgeaktivitäten mit gelöscht werden (vgl. [Hen97]).

5.7 Optimierung für lokale Änderungen

In vielen Fällen ist es wünschenswert, dynamische Änderungen einschließlich abhängiger Folgeänderungen nicht sofort mit allen an der Ausführung beteiligten Servern synchronisieren zu müssen. Dies gilt insbesondere, wenn ein anderer paralleler Zweig von einem Server kontrolliert wird, der weit entfernt und über langsame Leitungen angebunden ist, und die Änderungen keinen Einfluß auf Aktivitäten dieses Servers haben. Dann kann so ein deutlich besseres Kommunikationsverhalten erzielt werden.

Ein im folgenden vorgestellter Ansatz geht davon aus, daß Änderungsinformationen zunächst nicht zu anderen Servern ausgebreitet werden müssen, wenn sie lediglich den Server, von dem die Änderung ausgeht, betreffen. Es gibt jedoch noch weitere Fälle, in denen eine Optimierung möglich ist. Im folgenden wird eine Liste von Bedingungen hierfür aufgestellt. Dabei werden zunächst nur die Aspekte betrachtet, die den Kontrollfluß betreffen. Im Anschluß werden mit der Berücksichtigung des Datenflusses hinzu kommende Einschränkungen analysiert. Anhand eines abschließenden Beispiels werden die wesentlichen Zusammenhänge veranschaulicht.

Die Server besitzen unter Anwendung des beschriebenen Verfahrens für eine gewisse Zeit eine unterschiedliche Graphstruktur derselben Instanz. Wenn Server z.B. bei Migrationen oder im Rahmen von nicht lokalen dynamischen Änderungen ihre Zustandsinformation weitergeben, müssen sie dabei auch die in der Zwischenzeit vorgenommen lokalen Änderungen übertragen und integrieren. Nach dieser Synchronisation verfügen die beteiligten Server wieder über die für sie notwendigen aktuellsten Informationen zu Ausführungszustand und Graphstruktur.

5.7.1 Betrachtung ohne Datenfluß

In diesem Abschnitt werden zwei unterschiedliche Optimierungsansätze skizziert, die in einem WfMS alternativ eingesetzt werden können. Es müssen immer alle lokalen Änderungsoperationen nach derselben Variante durchgeführt werden. Eine Kombination der Ansätze kombiniert positive Eigenschaften beider Varianten, macht aber auch zusätzliche Einschränkungen erforderlich. Sie soll daher nicht näher betrachtet werden.

Variante 1: Voraussetzung für die Anwendung dieser Variante ist, daß alle durch die Änderung betroffenen Aktivitäten[37] vom selben Server kontrolliert werden, und zwar demjenigen, von dem die Änderung ausgeht. Außerdem müssen sie sich auf der gleichen Partition befinden. In diesem Fall kann eine Modifikation vollständig lokal durchgeführt werden. Der gegenseitige Ausschluß von dynamischen Änderungen untereinander bleibt auf diesem Server bestehen, egal ob es sich um eine lokale oder eine globale Änderung handelt. Das Verfahren kann mit dem in Abbildung 5-11 beschriebenen Protokoll kombiniert werden.

Diese Optimierungsvariante kann beim Einfügen oder Löschen von Aktivitäten in einer (Teil-)Sequenz, deren Aktivitäten dem selben Server zugeordnet sind, angewendet werden. Sie eignet sich ebenfalls für Änderungen innerhalb unterschiedlicher Zweige einer parallelen Verzweigung, sofern die relevanten Aktivitäten der gleichen Partition angehören.

[37] Dies sind Aktivitäten, deren Attribute verändert werden, oder bei denen Kanten hinzugefügt oder entfernt werden. Lediglich durch das Einfügen von leeren Knoten im Rahmen einer Änderungsoperation betroffene Aktivitäten zählen hier nicht dazu. Der leere Knoten n1Y in Abbildung 5-13 hat beispielsweise keinen Einfluß auf die Ausführungssemantik der Aktivitäten A und B. Aufgrund dessen können in so einem Fall jedoch nicht mehr die Elementaroperationen verwendet werden, um eine lokale Änderung weiterzugeben. Bei Folgeänderungen, d.h. beim Zusammenführen der Änderungen, können Probleme entstehen (siehe Abbildung 5-13, auf S1 wurde die Aktivität X und auf S2 die Aktivität Y lokal eingefügt). Die Änderung „Einfügen von X“ muß beim Einspielen der Änderungshistorie auf S2 neu berechnet werden, um eine zulässige Blockstruktur zu erhalten.

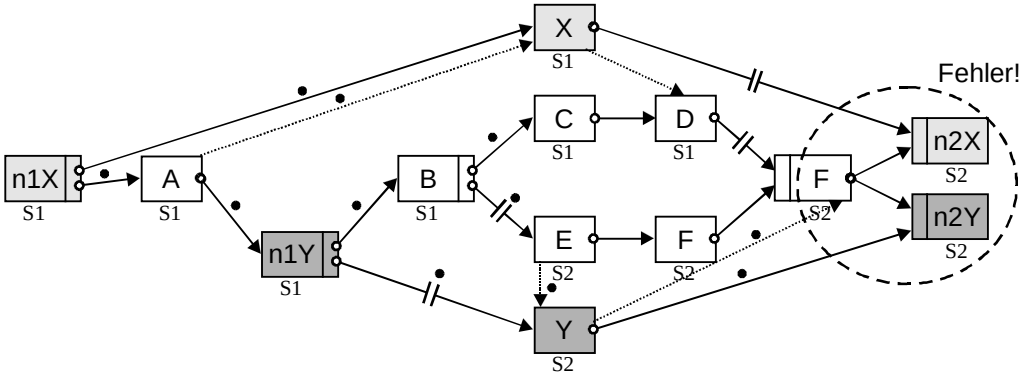


Abbildung 5-13: Fehlerhafte Graphstruktur durch Elementaroperationen

Variante 2: Diese Variante benötigt im Gegensatz zu Variante 1 keine besonderen Voraussetzungen. Durch die Änderung werden n Zweige mit Aktivitäten von unterschiedlichen Servern bzw. dem gleichen Server innerhalb einer anderen Partition betroffen. Der Fall, daß alle aktuell an der Ausführung beteiligten Server von der Änderung betroffen sind, ist ein Spezialfall hiervon.

Bei dieser Variante ist zunächst die *für die Änderung relevante Teilmenge der aktuell an der Ausführung beteiligten Server* zu bestimmen. Ein relevanter Server ist ein solcher Server, der den Zweig einer Vorgängeraktivität (über Kontroll- und Sync-Kanten) der betroffenen Aktivitäten kontrolliert. Da die Menge *ActiveServers* keine Angabe darüber enthält, welche Aktivitäten und damit welche der parallelen Zweige ein Server kontrolliert, sind für die Bestimmung zusätzliche Maßnahmen notwendig[38].

Mit der nachfolgend beschriebenen Vorgehensweise können vom Initiator der Änderung alle relevanten Server festgestellt und in ein Protokoll eingebunden werden: Zunächst sind die Anfangsaktivitäten aller geöffneten parallelen Verzweigungen zu bestimmen, welche Vorgänger (über Kontroll- und Sync-Kanten[39]) der von der Änderung betroffenen Aktivitäten sind. Alle Server dieser Aktivitäten werden der Reihe nach kontaktiert. Bei dem Aufruf müssen sie, falls sie gerade aktiv an der Ausführung der betroffenen Instanz beteiligt sind, die von ihnen kontrollierten Aktivitäten A zurückgeben und ab jetzt alle Migrationen im Zeitraum der dynamischen Änderung über den Initiator der Änderung abwickeln. Dieser bekommt die Aufgabe des Koordinators, damit er über die für ihn relevante Teilmenge der aktiven Server informiert ist. Wenn ein Server den Aufruf erhält, der nicht mehr an der Ausführung beteiligt ist oder von dem ausgehend eine Migration stattgefunden hat[40], beantwortet er den Aufruf mit dem Namen des Servers (bzw. den Namen der Server), an die er die Kontrolle übergeben hat. Die Aufrufe werden solange nacheinander an die Server aus dem Vorgängerbereich der dynamischen Änderung geschickt, bis alle benötigten aktiven Server festgestellt wurden. Dies ist der Fall, wenn es über Kontroll- und Sync-Kanten keinen Pfad mehr vom Startknoten zu einer der von der Änderung betroffenen Aktivitäten gibt, der nicht über eine Aktivität der Menge A führt.

Jetzt kann die Änderung mit dem aus Abschnitt 5.5.5 bekannten Protokoll eingeleitet werden, mit dem Unterschied, daß nicht der Startserver, sondern der Initiator der Änderung als Koordinator

---

[38] Auf einen ständigen Koordinator wie z.B. den Startserver, der die Menge *ActiveServers* bereitstellt, kann bei Verwendung dieses Verfahrens verzichtet werden! Zu dem Preis, daß jetzt vor der Durchführung einer dynamischen Änderung Mehraufwand nötig ist, kann bei der regulären Ausführung (Migrationen) Synchronisationsaufwand gespart werden.

[39] Eine mögliche weitergehende Optimierung, die nicht näher beschrieben werden soll, berücksichtigt neben allen Kontrollkanten nur Sync-Kanten im Zustand NOT_SIGNALED. Dadurch kann die Menge der Server, mit denen synchronisiert wird, weiter reduziert werden. Allerdings ergeben sich dann zusätzliche Einschränkungen in Bezug auf den Datenfluß.

[40] Hierzu zählen an dieser Stelle nur die Migrationen, die von Folgeaktivitäten (über Kontroll- und Sync-Kanten) des Splitknotens der geöffneten parallelen Verzweigung ausgingen.

eingesetzt wird. Da dieser Koordinator nicht zwangsläufig für alle zu diesem Zeitpunkt aktiven Server zuständig ist[41], müssen auf den einzelnen Servern jetzt zusätzlich zur Startsperre noch Sperren „MIGR“ zum Verhindern von gleichzeitig eingehenden Migrationen (der betroffenen Prozeßinstanz) und anderen dynamischen Änderungen gesetzt werden. Die Sperre „MIGR“ muß nur auf dem Server des Initiators der Änderung gesetzt werden, damit dieser seine Zustandsinformation korrekt aktualisieren kann. Die anderen Sperren müssen auf allen an der Durchführung der Änderung beteiligten Servern beantragt werden. Kann eine Sperre zur Verhinderung gleichzeitiger dynamischer Änderungen nicht gewährt werden, ist die Änderungsoperation abzubrechen. Da mehrere dynamische Änderungen und somit Koordinatoren zur selben Zeit aktiv sein dürfen, können sonst Verklemmungen entstehen.

Bevor die Änderung durchgeführt werden kann, muß der Initiator bei sich die von den beteiligten Servern bis zum aktuellen Zeitpunkt getätigten lokalen Änderungen, die in seinem Instanzausführungsgraphen noch nicht vorhanden sind, nachtragen. Er muß sich von den relevanten Servern außerdem Informationen über den aktuellen Ausführungszustand holen.

Nach Durchführung der Änderung und Ausbreitung zu den beteiligten Servern verliert der Initiator seine Koordinatorfunktion. Diese Server arbeiten nach dem Aufheben ihrer Sperren wie auch die anderen aktiven Server wieder ohne Koordinator weiter.

5.7.2 Betrachtung mit Datenfluß

Für die meisten praktisch relevanten Änderungsoperationen ist die alleinige Betrachtung des Kontrollflusses nicht ausreichend, da zu fast allen Aktivitäten auch gelesene und/oder geschriebene Daten gehören. Voraussetzung für die Durchführung einer lokalen Änderung ist zunächst die Erfüllung der entsprechenden Bedingungen für den Kontrollfluß je nach der gewählten Variante 1 oder 2 (vgl. Abschnitt 5.7.1). Zusätzlich müssen weitere Voraussetzungen bezüglich der Ein- und Ausgabeparameter der von der Änderungsoperation betroffenen Aktivitäten erfüllt sein. Wenn es nicht ausdrücklich anders erwähnt wird, gelten die formulierten Bedingungen für beide alternative Varianten. An zwei Stellen müssen jedoch bei Verwendung der Variante 1 zusätzliche Einschränkungen in Bezug auf die Ausgabeparameter der einzufügenden bzw. zu löschenden Aktivität in Kauf genommen werden.

Bei einem WfMS, das Optimierungen für lokale Änderungen nach Variante 2 realisiert, muß zwar in einigen Fällen mit anderen Servern kommuniziert werden, in denen dies bei Variante 1 nicht nötig wäre. Dafür sind dann lokale Änderungen insgesamt häufiger anwendbar, da sie weniger strengen Zusatzanforderungen bezüglich des Datenfluß unterliegen. In der Regel wird Variante 2 das bessere Ergebnis erzielen.

Beispielhaft werden zwei wichtige Operationen betrachtet, das Einfügen und das Löschen von Aktivitäten. Statt diesen könnten auch die darunterliegenden Basisoperationen herangezogen werden. Für jede Operation werden notwendige Bedingungen für die Zulässigkeit einer Durchführung als lokale Änderung festgehalten.

Einfügen von Aktivitäten

Die Eingabeparameter einer Aktivität können im Zusammenhang mit dem Einfügen von Aktivitäten innerhalb lokaler dynamischer Änderungen keine neuen Probleme erzeugen. Für sie gelten nur die bei jeder Einfügeoperation einzuhaltenden Bedingungen, daß sie versorgt sein müssen.

Wird eine neue Aktivität N eingefügt, die auch Ausgabeparameter besitzt, muß verhindert werden, daß durch die Änderung paralleles Schreiben im Graphen stattfindet. Dies tritt ein, wenn in einem parallelen Zweig eine Aktivität X und in einem anderen Zweig unabhängig davon eine Aktivität Y lokal eingefügt wird, die beide das Datenelement d beschreiben. In den folgenden Fällen kann dieses Problem nicht auftreten und eine lokale Änderung ist somit möglich:

a) Das beschriebene Datenelement ist ein neues Datenelement.

[41] Er ist nur für die Teilmenge der Server zuständig, die an der Durchführung der Änderung beteiligt sind.

b) Unter Anwendung der Optimierung nach 5.7.1 Variante 2 genügt es, wenn eine Vorgängeraktivität (über Kontroll- und Sync-Kanten) von N, die sich nach dem letzten Splitknoten einer geöffneten parallelen Verzweigung befindet, schreibt. Aufgrund der Konstruktion des Verfahrens wird mit allen Servern, die einen parallelen Zweig kontrollieren, in denen sich eine solche Vorgängeraktivität befindet, eine Synchronisation durchgeführt. So kann festgestellt werden, ob in einem dieser Zweige eine Aktivität lokal eingefügt wurde, die ebenfalls d beschreibt, so daß ein Konflikt besteht.

   Eine weitere Möglichkeit zur lokalen Änderung besteht außerdem, wenn eine Nachfolgeraktivität SUCC (über Kontrollkanten) von N oder eines direkten Nachfolgers von N über eine Sync-Kante (incl.), die sich vor dem nächsten Joinknoten einer geöffneten parallelen Verzweigung befindet, schreibt. Dabei kann in keinem Fall eine neue, noch unbekannte Aktivität das Datenelement schreiben, die sich in einem Zweig befindet, mit dessen Server nicht synchronisiert wird. In einem solchen Zweig kann, vor allen Nachfolgern von SUCC über Kontroll- und Sync-Kanten innerhalb der parallelen Verzweigung, keine Aktivität eingefügt worden sein, welche d beschreibt, da bei einer solchen Einfügeoperation paralleles Schreiben zu der bereits vorhandenen Aktivität SUCC auftreten würde. Zur Veranschaulichung ist das Beispiel in Abbildung 5-14 geeignet, welches in Abschnitt 5.7.3 erklärt wird.

   Bei Verwendung von Variante 1 gilt statt dessen folgende Einschränkung: Das Datenelement wird von keiner anderen Aktivität der parallelen Verzweigung, welche nicht von dem Server kontrolliert wird, auf dem die lokale Änderung stattfindet, geschrieben. Alternativ kann bei Variante1 eine andere Einschränkung gewählt werden, die dazu führt, daß nur die Operation *serialInsert* unterstützt wird, diese allerdings weiteren Fällen. Hierzu muß das Datenelement bereits von einer Vorgängeraktivität von N (über Kontrollkanten), die sich nach dem letzten Knoten mit mehr als einer Ausgangskante befindet oder von einer Nachfolgeraktivität von N beschrieben werden, die sich vor dem nächsten Knoten mit mehr als einer Eingangskante befindet.

Im Fall von mehreren Ausgabeparametern müssen die Bedingungen a) oder b) für alle beschriebenen Datenelemente gelten.

Löschen von Aktivitäten

Die Eingabeparameter einer Aktivität können im Zusammenhang mit dem Löschen von Aktivitäten innerhalb lokaler dynamischer Änderungen keine neuen Probleme erzeugen. Auch bisher schon spielen sie beim Löschen keine Rolle.

Wenn eine Aktivität D mit Ausgabeparametern gelöscht wird, muß sichergestellt werden, daß alle nachfolgenden Leser noch mit Daten versorgt sind. Dies gilt auch schon bisher.

- Unter Anwendung der Optimierung nach 5.7.1 Variante 2 sind keine zusätzlichen Probleme möglich. Der Server, der das lokale Löschen durchführt, kennt bereits alle neu eingefügten Aktivitäten, die der zu löschenden Aktivität über Kontroll- und Sync-Kanten nachfolgen und von ihr geschriebene Datenelemente lesen.
- Bei Verwendung der Optimierung nach 5.7.1 Variante 1 besteht jedoch die Möglichkeit, daß in einem Nachfolgebereich der zu löschenden Aktivität eine Aktivität lokal eingefügt wurde, die sich darauf verläßt, von der zu löschenden Aktivität mit Daten versorgt zu werden[42] (Aktivität X liest das von C geschriebene Datenelement d_1 in Abbildung 5-15). Der Änderungsserver weiß jedoch nichts darüber, daß eine solche neue Aktivität existiert, da die Einfügeoperation lokal vorgenommen wurde.

  Es muß daher verhindert werden, daß eine Aktivität innerhalb einer lokalen Änderungsoperation gelöscht wird, wenn nach dem Löschen nicht mehr sichergestellt ist, daß nachfolgend lokal

[42] Hinzu kommt der Fall, daß sich der Serverzuordnungsausdruck einer lokal eingefügten Aktivität auf die zu löschende Aktivität bezieht. Er soll hier nicht weiter betrachtet werden. Bei der Wahl einer geeigneten Strategie in Abschnitt 5.6 kann das Problem nicht auftreten.

eingefügte Aktivitäten mit Daten versorgt werden können. In den folgenden Fällen kann dieses Problem nicht auftreten:

a) Irgendein Vorgänger von D hat das Datenelement bereits beschrieben.

b) Ein Nachfolger von D innerhalb der gleichen Partition beschreibt das Datenelement obligat.

c) Die Funktion writerExists(D, d) ergibt true für das Datenelement, d.h. es existiert garantiert ein Schreiber von d in der Menge der Vorgänger der Aktivität D. Diese Bedingung ist nicht mit a) identisch, da hierbei auch obligate Schreiber berücksichtigt werden, die noch nicht ausgeführt wurden.

Im Fall von mehreren Ausgabeparametern müssen die Bedingungen a), b) oder c) für alle beschriebenen Datenelemente gelten.

5.7.3 Fallbeispiele

Im WF-Graph der Abbildung 5-14 wurde die Aktivität X nach Aktivität H und vor Aktivität F eingefügt, initiiert durch den Server S4. Das Einfügen wurde als lokale Änderungsoperation durchgeführt, optimiert nach der Variante 2 aus Abschnitt 5.7.1. Eine Synchronisation wurde nur mit den Servern S3, S5 und S4 - dem Koordinator der Änderung - durchgeführt, weil diese Server Zweige kontrollieren, die den von der Änderung betroffenen Aktivitäten vorangehen. Die Graphstruktur von S1 wird wie bei globalen dynamischen Änderungen bei einer Migration zu K ($M_{F,K}$, $M_{X,K}$, $M_{I,K}$ oder $M_{O,K}$) aktualisiert. Der (aktive) Server S2 ist nicht in die Änderung involviert.

Angenommen, X schreibt ein im Graph bereits vorhandenes Datenelement d. Jetzt wäre nur noch dann eine lokale Synchronisation möglich, wenn auch die Aktivitäten F, G, H oder M das Datenelement schreiben[43]. In einem anderen Fall sei angenommen, daß eine Sync-Kante F→D existiert. Wenn d nur von D geschrieben wird, kann die beschriebene Änderung nicht lokal durchgeführt werden, da mit S2 nicht synchronisiert wird! Auf S2 könnte eine lokale Einfügeoperation stattgefunden haben, die eine Aktivität Y zwischen C und D eingefügt hat, die ebenfalls d beschreibt. Diese Änderung wäre in Konflikt mit dem Einfügen von X.

Durch diese Bedingungen wird verhindert, daß in unterschiedlichen parallelen Zweigen lokal eingefügte Aktivitäten das Datenelement d beschreiben und es so zu einem unzulässigen parallelen Schreiben kommen kann. Sind die in Abschnitt 5.7.2 formulierten Bedingungen an den Datenfluß nicht erfüllt, muß global, d.h. auch mit S2 synchronisiert werden.

Die Kriterien an den Datenfluß erscheinen zunächst sehr streng. Sie sind jedoch ausreichend, um in den meisten praktisch relevanten Fällen, in denen lokale Änderungen vorteilhaft sind, diese auch durchführen zu können. Bei dem Beispiel aus Abbildung 5-14 ist das lokale Einfügen einer Aktivität Z, die Datenelement e schreibt, beispielsweise zwischen N und O auf S5 möglich, wenn Aktivität M, N oder O das Datenelement ebenfalls beschreiben und dies keine Aktivität eines anderen Zweiges tut. Verschachtelte parallele Verzweigungen sind möglich, hierbei kann sich eine Änderungsoperation sogar auf Teilzweige einer inneren Verzweigung beschränken. Die Optimierung kann auch in Verbindung mit Schleifen angewendet werden. Aufgrund der Blockstruktur der Ausführungsgraphen ist eine Schleife stets auf einen Teilzweig einer parallelen Verzweigung beschränkt und/oder ein Schleifenkörper enthält eine bzw. mehrere vollständige parallele Verzweigungen.

[43] Aktivität F ist direkter Nachfolger von X über eine Sync-Kante, G, H und M sind Vorgänger über Kontroll- und Sync-Kanten innerhalb der geöffneten parallelen Verzweigung. Die zu X parallelen Aktivitäten E, I, N und O dürfen das Datenelement d nicht schreiben. Dies wird immer automatisch bei einer Einfügeoperation geprüft. Aktivität C und D können ohnehin nicht schreiben, wenn die Bedingung, daß F, G, H oder M schreibt, erfüllt ist.

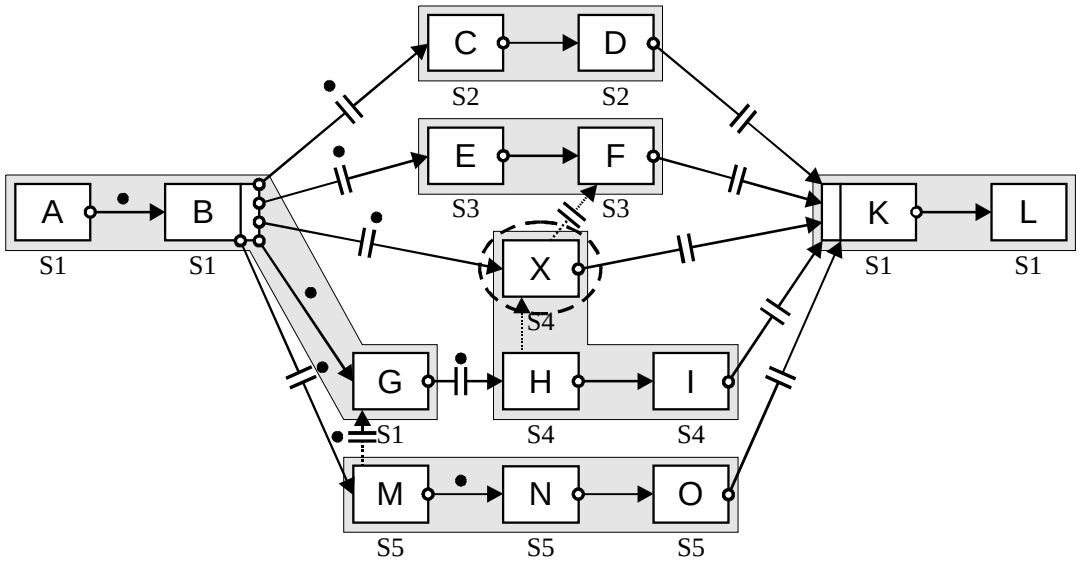


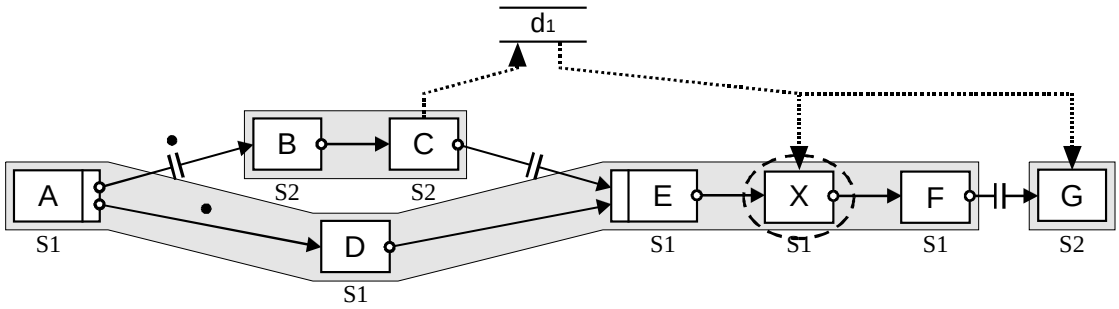
Abbildung 5-14: Lokale Synchronisation dynamischer Änderungen.

Das folgende Beispiel zeigt den Unterschied zwischen den beiden in Abschnitt 5.7.1 dargestellten Varianten. Es soll zunächst die Variante 1 angenommen werden. Abbildung 5-15 verdeutlicht, aus welchem Grund Aktivitäten mit Ausgabeparametern nicht ohne weiteres in einer lokalen Änderungsoperation gelöscht werden dürfen. Vom Server S1 aus wurde die Aktivität X zwischen E und F eingefügt. X liest das Datenelement d_1, welches durch C geschrieben wird. Die Modifikation konnte lokal durchgeführt werden, da die Optimierungsregel für Variante 1 erfüllt ist. Dem Server S2 ist die Änderung zum aktuellen Zeitpunkt somit nicht bekannt.

S2 möchte, nach Möglichkeit ebenfalls lokal, auch eine Änderung am Graphen vornehmen. Aktivität C soll einschließlich ihrer datenabhängigen Folgeaktivität G gelöscht werden. Ohne Berücksichtigung des Datenflusses ist keine Synchronisation mit allen aktiven Servern, d.h. hier mit S1, notwendig. Da jedoch keine weitere Aktivität vor C das Datenelement d_1 beschreibt, muß die Änderung global synchronisiert werden. S2 aktualisiert seine Graphstruktur, die dann auf dem gleichen Stand wie bei S1 ist. S2 stellt fest, daß nach dem Löschen die Aktivität X nicht mehr mit Eingabedaten versorgt ist und bricht infolgedessen z.B. die Änderung ab.

Wenn der WF-Server lokale dynamische Änderungen nach Variante 2 durchführt, wird der Server S2 bereits in die erste Änderungsoperation, dem Einfügen von X, einbezogen. Bei der Überprüfung der Folgeänderung verfügt er so bereits über die nötige Information.

Graph aus der Sicht von S1



Graph aus der Sicht von S2

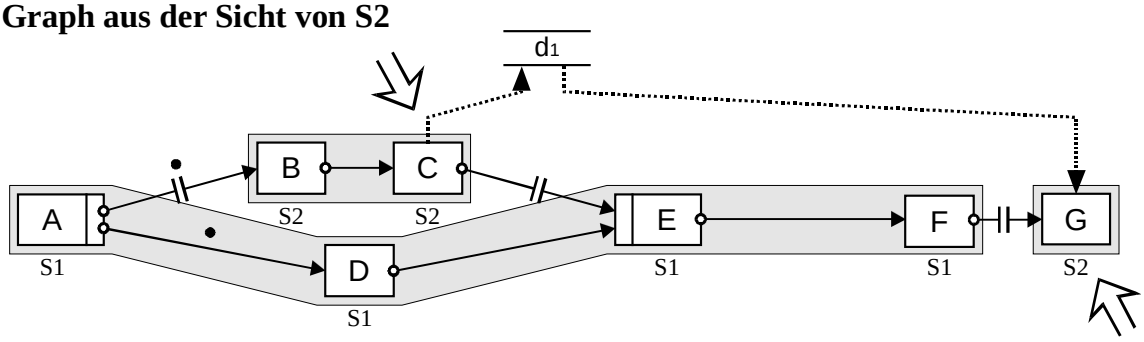


Abbildung 5-15: Lokale Synchronisation dynamischer Änderungen - mit Datenfluß.

6 Integration der Verteilungsschicht in den ADEPT-Prototyp

Die in den Kapiteln 3 bis 5 erarbeiteten Konzepte wurden in den bestehenden ADEPT-Prototyp integriert. Dadurch wird die im Prototyp bislang noch nicht existierende Verteilungsschicht realisiert.

6.1 Stand des Prototyps vor der Diplomarbeit

Zunächst erfolgt eine kurze Beschreibung der Ausgangsbasis. Seit 1997 wird in der Abteilung Datenbanken- und Informationssysteme der Universität Ulm ein Prototyp entwickelt, mit dem die im Rahmen des ADEPT-Projektes entwickelten Konzepte demonstriert werden können. Die Entwicklung erfolgt in der Programmiersprache Java, als zugrunde liegende Datenbank wird Oracle 8 verwendet. Die Anbindung an die Datenbank wird mit JDBC Typ 4 realisiert, d.h. auf der Seite des WF-Servers (der Client für den Datenbankserver ist) wird ein komplett in Java geschriebener Datenbanktreiber verwendet, der dem JDBC-Standard genügt.

Die Client-Server-Kommunikation wird über Java RMI abgewickelt. Aufträge an den Server gelangen dort über Queues zur Bearbeitung durch die Aktivitätsträger. Im Anschluß an die Auftragsdurchführung, welche innerhalb einer Transaktion geschieht, erhält der Client eine Antwort. Darüber hinaus können an weitere Clients Aktualisierungsnachrichten verschickt werden. Der Aufbau der WfMS-Architektur kann im Anhang D nachvollzogen werden. Die beiden dort abgebildeten Grafiken wurden mit freundlicher Genehmigung aus [HSB99] entnommen.

Bei der Erweiterung des Prototyps wurden die nachstehenden Ziele verfolgt: Primär soll die verteilte Ausführung von Workflows ermöglicht werden. Außerdem sollen die bereits implementierten dynamischen Änderungsoperationen auch im verteilten Fall korrekt durchführbar sein.

Es wurden nicht in jedem Fall die bei den Konzepten als beste Möglichkeit dargestellten Varianten implementiert. Statt dessen wurde der Schwerpunkt darauf gelegt, ein stimmiges Gesamtsystem zu erstellen. Alle bisher zulässigen Konstrukte und Änderungsoperationen sind auch bei verteilter Ausführung durchführbar. Unnötiges Kommunikationsvolumen wird dennoch weitgehend vermieden, auch wenn diesbezüglich und in Bezug auf die Reduzierung von benötigter Rechenleistung und Datenbankzugriffen noch Optimierungen möglich sind.

Bei der Integration der Verteilungsschicht wurden grundlegende Änderungen am Prototyp weitgehend vermieden. Nach wie vor kann ein ADEPT-Server auch als eigenständiger Server im Einzelbetrieb betrieben werden.

Welche der Varianten für die Implementierung ausgewählt wurden, kann Abschnitt 6.2 entnommen werden. Darüber hinaus notwendige Erweiterungen, die im Konzeptteil noch nicht beschrieben wurden, werden in Abschnitt 6.3 erläutert. An welchen Stellen noch Optimierungen vorgenommen werden können und in welchem Umfang zukünftige Erweiterungen bereits berücksichtigt werden, wird in Abschnitt 6.4 beschrieben.

6.2 Umgesetzte Konzepte

Die implementierten Verfahren wurden mit der Prämisse ausgewählt, daß sowohl statische als auch variable Serverzuordnungen unterstützt werden. Dynamische Änderungen sollen möglich sein, ohne daß dadurch erheblicher Zusatzaufwand in Gestalt von neu benötigten Methoden außerhalb der Klasse, die das entsprechende Protokoll realisiert, entsteht. Die reguläre Ausführung der Workflows soll möglichst wenig beeinträchtigt werden.

6.2.1 Voraussetzungen

Jeder ADEPT-Server muß auf einem eigenen Rechner, d.h. mit eigener IP-Adresse, laufen. Er benötigt eine eigene Datenbank bzw. zumindest einen eigenen Satz Tabellen. Im Idealfall befindet sich die Datenbank lokal auf dem selben Server wie der WF-Server. Da für die verteilte Ausführung symbolische Servernamen vergeben werden, muß die entsprechende Zuordnung zu IP-Adressen für den Prototyp in einer Konfigurationsdatei festgelegt werden (siehe Anhang E.1).

In der Datenbank jedes WF-Servers müssen bereits bestimmte im Gesamtsystem gültige Informationen vorhanden sein. Dies sind Prozeß- und Aktivitätenvorlagen mit referenzierten Objekten sowie das Organisationsmodell. Da diesbezüglich noch kein Replikationsmechanismus vorhanden ist, werden zunächst die Daten von einem bestimmten Server, der als Primärserver gilt, übernommen. Änderungen sollten nur dort vorgenommen werden und anschließend durch den WF-Administrator auf die anderen Server repliziert werden. Die hierfür benötigte Funktionalität wurde implementiert. Sie kann zukünftig ohne weiteres durch einen aufwendigeren Mechanismus ersetzt werden.

Zunächst gilt: Prozeßvorlagen von laufenden Prozessen dürfen nicht verändert werden. Es müssen statt dessen neue Versionen erzeugt werden. Anderenfalls könnten nämlich bereits bestehende Prozeßinstanzen, die nach der alten Vorlage erstellt wurden, nach Migrationen zu einem bisher nicht an der Ausführung beteiligten Server dort nicht mehr identisch reproduziert werden. Dort erstellte Instanzen entsprechen der neuen Vorlage. Das System wäre inkonsistent geworden.

Wenn allerdings Vorlagenänderungen auch auf bereits bestehende Instanzen angewendet werden würden, entständen keine Inkonsistenzen zwischen den Servern. Mit dieser Thematik, der Schemaevolution in WfMS, beschäftigen sich weitere Forschungsarbeiten der Abteilung DBIS, zu Replikationsmechanismen verweise ich auf [CDK95].

6.2.2 Ausführung von WF-Instanzen

Eine neue Instanz kann auf einem beliebigen Server gestartet werden. Dem Attribut „ausführender Server“ der Startaktivität wird dann dieser Server zugewiesen, so daß hierauf eine Bezugnahme möglich wird. Vor jeder Aktivierung einer neuen Aktivität wird geprüft, ob diese noch auf dem gleichen Server angesiedelt ist. Nur dann kann sie anschließend auch aktiviert werden, anderenfalls wird für die betreffende Kante in der Datenbank eine Migration vermerkt und, wenn möglich, im Anschluß durchgeführt.

Eine Migration läuft nach dem in Abbildung 5-9 beschriebenen Protokoll ab. Wenn ein Server erstmalig an der Ausführung der Instanz beteiligt ist, initialisiert er zunächst aus der bei ihm vorhandenen Prozeßvorlage und Attributen der bereits existierenden Prozeßinstanz eine neue Prozeßinstanz. Durch Einspielen der vom Quellserver erhaltenen kompletten Änderungshistorie, der Ausführungshistorie und der Datenelemente wird diese auf den aktuellen Stand gebracht.

Für die Übertragung der Zustandsinformation wird das in Abschnitt 3.4.2 beschriebene Holen-Verfahren verwendet. Ist die Prozeßinstanz bereits bekannt, können sog. Migrationspunkte berechnet werden, die dem Ausgangsserver bekanntgegeben werden. Dieser kann dann die Historieninformation berechnen, die er dem Zielserver noch mitteilen muß. Somit wird redundante Übertragung vermieden. Alle Ausführungshistorieneinträge mit Ausnahme der Einträge zu übersprungenen Knoten einer bedingten Verzweigung („SKIPPED“) werden berücksichtigt.

Datenelemente werden mit den jeweiligen Historienabschnitten übertragen, die den Knoten entsprechen, von denen sie geschrieben wurden. Dies entspricht weitgehend dem beschriebenen Verfahren zur Übertragung kleiner Datenelemente. Große Datenelemente sind im Prototyp bislang noch nicht vorgesehen, daher wird für sie keine eigene Behandlung durchgeführt. Mit Hilfe der so erhaltenen Differenzinformation kann der Zustand ebenfalls aktualisiert werden.

Darüber hinaus wird bei jeder Migration die vom Koordinator verwaltete Menge *ActiveServers*, d.h. die momentan an der Ausführung der Instanz beteiligten Server, aktualisiert. Als Koordinator wird der Startserver der Prozeßinstanz verwendet. Auf dem Ausgangsserver wird im letzen Ausführungshistorieneintrag der Quellaktivität wird im Attribut MIGRATION der tatsächliche Zielserver protokolliert.

Falls an einem Knoten mit mehreren Eingangskanten dessen Serverzuordnungsausdruck nicht von jedem Migrationsquellserver ausgewertet werden kann, wird ein erweitertes Protokoll notwendig (vgl. Abschnitt 4.6). Die Server, welche den Ausdruck nicht auswerten können, warten mit ihrer Migration und registrieren sich bei einem bestimmten Server. Andere Server, die den Ausdruck auswerten können, teilen das Ergebnis dem Koordinator-Server mit, woraufhin dieser bei den wartenden Servern die Migrationen anstoßen kann. Er behält das Ergebnis, damit evtl. weitere anfragende Server dieses mitgeteilt bekommen können. Auch für diese Vermittler-Aufgabe wird der Startserver verwendet.

6.2.3 Dynamische Änderungen

Dynamische Änderungen werden bei der Durchführung mit allen aktuell an der Ausführung beteiligten Servern synchronisiert. Sie können von jedem Server aus, der die Prozeßinstanz schon kennt, ausgelöst werden. Dabei wird zunächst die aktuelle Zustandsinformation eingeholt und überprüft, ob die Änderung zulässig ist. Für eine endgültige Durchführung müssen Sperren gesetzt werden. Das verwendete Protokoll entspricht dem in Abbildung 5-11 beschriebenen.

Die Änderungsoperationen werden als komplexe Ausdrücke weitergegeben, welche die Operation auf einer abstrakten Ebene beschreiben [Rei00]. Sie lauten in natürlicher Sprache z.B. „Einfügen von Aktivität X nach den Aktivitäten A und B und vor Aktivität G in der Prozeßinstanz 12345.“. Informationen zu mit der Änderung zusammenhängenden Zustandsänderungen am Graphen wird ebenfalls hier vermerkt, da dafür kein gesonderter Ausführungshistorieneintrag geschrieben wird[44]. Auf dem Zielserver wird die Operation erneut ausgeführt. Sie muß natürlich durchführbar sein, sonst hätte sie auf dem Server, von dem die Änderung ausgeht, nicht angewendet werden können. Dadurch wird der Graph dort in derselben Weise manipuliert, so daß der Instanzgraph am Ende wieder auf allen Servern identisch ist.

Da neue Datenbankobjekte grundsätzlich auf jedem Server unabhängig erstellt werden[45], müssen beim Vorgehen nach der beschriebenen Methode zusätzliche Vorkehrungen getroffen werden, damit die Graphen auch identisch bleiben. Sonst würden in der Ausführungshistorie Einträge zu Knoten protokolliert, welche auf anderen Servern gar nicht identifizierbar wären. Im Änderungshistorieneintrag werden deswegen zusätzlich die IDs der neu erstellten Knoten abgelegt, so daß diese zum Erzeugen von Knoteninstanzen mit bestimmter ID verwendet werden können, wenn die Änderung auf einem anderen Server auf dessen Kopie des Graphen angewendet wird. Nach dem selben Schema muß mit neu erstellten Datenelementen verfahren werden, damit geschriebene Werte auf allen Servern dem korrekten Datenelement zugeordnet werden können.

Identische Graphen bedeutet hier also nicht, daß jedes im Instanzgraphen enthaltene Objekt auf jedem Server dieselbe ID haben muß. Jedoch müssen alle Knoten und Datenelemente der Prozeßinstanz die gleiche ID haben, und auch die Graphstruktur und alle Referenzen zu weiteren Datenbankobjekten müssen identisch sein. Eine neue Kante kann somit auf unterschiedlichen Servern durchaus eine unterschiedliche ID haben, jedoch müssen Quell- und Zielknoten dieser Kante überall dieselben IDs haben.

Weitergabe der Änderungsinformation bei Migrationen

Bei Migrationen erfolgt die Weitergabe der Änderungsinformation durch die Änderungshistorie. Auf dem Zielserver noch nicht bekannte Einträge werden an diesen weitergegeben und dort vor dem Einspielen der Ausführungshistorie berücksichtigt. So wird gewährleistet, daß neu eingefügte Knoten in jedem Fall gemäß ggf. vorhandener Ausführungshistorieneinträge markiert werden können. Das Einspielen von Änderungs- und Ausführungshistorieneinträgen findet nämlich nicht in der exakten Reihenfolge zueinander, in der diese Einträge ursprünglich geschrieben wurden, statt. Nur für jede Historie einzeln betrachtet wird die Einhaltung der Reihenfolge gewährleistet. Dies liegt daran, daß die in der Ausführungshistorie enthaltenen DDM-Anker (vgl. Abschnitte 5.1 und 5.4.2) nicht übertragen werden. Sie verbleiben lediglich in der Historie der Server, die zum Zeitpunkt der Änderung aktiv waren. Zum Rücksetzen einer Änderung reicht es aus, wenn diese Information dort verfügbar ist. Eine weitere Funktion haben die DDM-Anker nicht. Somit unterscheidet sich das Verfahren etwas von dem in Abschnitt 5.4.2 beschriebenen. Für Undo-Modifikationen wird jetzt allerdings vorausgesetzt, daß hierfür auch Einträge in der Änderungshistorie aufgenommen werden, da sie sonst bei Migrationen nicht weitergegeben würden.

---

[44] Ein nachträglich erzeugter Eintrag zu einem neuen Knoten, dessen Nachfolger bereits ausgeführt wurden, würde die Korrektheit der Historie verletzen, wenn er einfach angehängt würde.

[45] Dies ist ausdrücklich erwünscht. In die ID eines Datenbankobjektes gehen die IP-Adresse des Servers, auf dem dieses erzeugt wurde, und eine von einem Datenbank-Sequencer erzeugte eindeutige Nummer ein. Nur so können in ADEPT auf unterschiedlichen Servern erzeugte Objekte ohne weiteres in eine gemeinsame Tabelle aufgenommen werden, ohne daß eine Verletzung eines Primärschlüssels (Objekt-ID) entstehen kann.

Die korrekte Ordnung der Einträge der Änderungshistorie ist auf allen Servern gewährleistet, da alle parallel an der Ausführung beteiligte Server zum Zeitpunkt einer Änderungsdurchführung diese Historie in gleicher Weise aktualisieren und Änderungen nur nacheinander endgültig durchgeführt werden können.

Serverzuordnungen und dynamische Änderungen

Beim dynamischen Einfügen neuer Aktivitäten werden den zugehörigen Knoten und eventuell im Zusammenhang erzeugten neuen Knoten eindeutige Serverzuordnungsausdrücke zugewiesen. Es wird jeweils der Server des Vorgängers über Kontrollkanten genommen, im Fall von mehreren eingehenden Kanten des Vorgängers der niedrigsten Knoten-ID.

Da im Prototyp das dynamische Löschen dergestalt durchgeführt wird, daß ein Knoten lediglich als gelöscht gekennzeichnet wird, ansonsten aber keine Änderungen vorgenommen werden und sich auch die Ausführungssemantik nicht ändert, wird garantiert für jeden Knoten ein Bearbeiter in der Ausführungshistorie vermerkt. Somit können keine Probleme aufgrund nicht mehr berechenbarer Serverzuordnungen entstehen. Ein gelöschter Knoten liest und schreibt lediglich keine Datenelemente mehr. Theoretisch wäre es für viele Aspekte sicher besser, beim Löschen den Graph so umzubauen, daß die Ausführungssemantik erhalten bleibt, der gelöschte Knoten jedoch nicht mehr vorhanden sein muß. Dann könnte auch nicht der Fall auftreten, daß nur aufgrund von gelöschten Knoten zu einem anderen Server migriert wird. Es müßten jedoch eventuell Serverzuordnungsausdrücke ersetzt werden, die sich auf einen gelöschten Knoten beziehen (vgl. Abschnitt 5.6).

6.3 Besondere Lösungen

An einigen Stellen wurden für die Implementierung besondere Lösungen notwendig, die bei den Konzepten noch nicht oder nur sehr knapp beschrieben wurden. Dies betrifft beispielsweise Verfahren zur Zurückstellung von Client-Aufträgen oder verschiedene Sperren. Zum Teil handelt es sich auch um Erweiterungen zu bereits beschriebenen Verfahren wie der Berechnung von Migrationspunkten.

6.3.1 Zustandsinformation im auftragsorientierten Protokoll

Wie auch die Clients kommunizieren die Server miteinander über Java RMI. Dabei werden Aufträge in eine Queue des angesprochenen Servers gestellt. Diese Aufträge werden innerhalb einer Transaktion bearbeitet. Mit Ausnahme der in der Datenbank vorhandenen Graph- und Historieninformation und den in Abschnitt 6.3.2 beschriebenen Sperren wird bei Migrationen über die einzelnen Kommunikationsschritte hinweg keine Zustandsinformation auf dem Quell- und Zielserver gehalten. Dies bedeutet, daß in jedem Teilauftrag die vollständigen, genau diese Migration beschreibenden Parameter enthalten sein müssen, damit der Auftrag wie gewünscht durchgeführt werden kann. Insbesondere müssen Prozeßinstanz-ID und die ID von Quell- und Zielknoten sowie deren Servernamen enthalten sein. Zusammen mit dem OP-Code des Teilauftrages kann das gewünschte Ergebnis ermittelt und der nächste notwendige Schritt angestossen werden.

Im Protokoll zu dynamischen Änderungen (vgl. Abbildung 5-11) kann diese Lösung nur noch teilweise verwendet werden. Wenn im letzten Teilabschnitt die Änderungsinformation gleichzeitig zu den anderen beteiligten Servern ausgebreitet wird, muß mit dem Fortfahren im Protokoll gewartet werden, bis von jedem der Server eine Bestätigung angekommen ist. Hierzu werden auf dem Server des Initiators der Änderung Objekte gespeichert, die mit der Prozeßinstanz-ID incl. der User-ID des Auslösers eindeutig bezeichnet werden. Hier wird unter anderem der Fortschritt im Protokoll festgehalten. Nach Abschluß des Protokolls werden diese temporären, in der aktuellen Implementierung des Prototypen nicht persistenten, Objekte wieder gelöscht.

Bei rein sequentieller Ausbreitung der Änderungsinformation könnte eine Liste der noch zu bearbeitenden und der vollständigen Active Servers von dem die Änderung auslösenden Server an einen anderen beteiligten Server mitgeschickt werden, so daß keine zusätzliche Zustandsinformation über die Teilaufträge hinweg gespeichert werden müßte. Allerdings erhöht sich dann die übertragene Datenmenge etwas, und die problemlos mögliche parallele Durchführung wird nicht genutzt.

6.3.2 Sperren

Zunächst gelangen die Teilaufträge jedes Protokolls in der Reihenfolge des Eintreffens in die Eingangsqueue des angesprochenen Servers. Sie könnten daher vollständig asynchron durchgeführt werden. Allerdings bestehen zwischen einigen Aufträgen Abhängigkeiten, die beachtet werden müssen. So darf beispielsweise für eine gegebene Prozeßinstanz immer nur eine eingehende Migration zur selben Zeit stattfinden. Weitere eingehende Migrationen müssen gesperrt, d.h. hier zurückgestellt werden. Im Prototyp sind Sperren zur Zeit nicht persistent. Sie werden in einer schlüsselindizierten Datenstruktur im Hauptspeicher gehalten, deren Schlüssel die eindeutigen Bezeichner der Sperre sind. Die Schnittstellen sind aber so vorbereitet, daß die momentan verwendeten Sperren gegen persistente Sperren identischer Funktionalität ausgetauscht werden können.

Prinzipiell wird dabei immer folgendermaßen vorgegangen: Vor jedem kritischen Abschnitt wird überprüft, ob dieser betreten werden darf. Wenn dies möglich ist, wird vom LockManager registriert, daß der kritische Abschnitt betreten wurde. Anderenfalls wird der komplette Auftrag zurückgestellt und erst dann wieder zur Ausführung in die ursprüngliche Queue eingebracht, wenn ein anderer Auftrag den kritischen Abschnitt verläßt und freigibt.

Die Vorgehensweise kann Algorithmus 6-1 in Pseudocode entnommen werden. Hierbei handelt es sich um einen Ausschnitt aus der Implementierung eines Protokolls, das eine Sperre zur Verhinderung gleichzeitig eingehender Migrationen einer Prozeßinstanz benötigt. AuftragA entspricht dem Teilauftrag desjenigen Servers innerhalb des Protokolls, bei dem der kritische Abschnitt beginnt, AuftragB dem Teilauftrag, bei dem die Sperre wieder freigegeben wird. Wird die Sperre gewährt, kann Auftrag A durchgeführt werden. Anderenfalls wird der Auftrag zurückgestellt und erst dann ein erneuter Versuch zur Durchführung begonnen, wenn die Sperre an anderer Stelle freigegeben wird (hier: innerhalb AuftragB).

```
AuftragA(params, request) {
    pass = LockManager.p(connection, „MIGR“, piId, request);
    if (pass) {
        Auftrag A durchführen
    } else {
        nichts tun, Methode wird wieder verlassen
    }
    return result;
}

AuftragB(params, request) {
    Auftrag B durchführen
    LockManager.v(connection, „MIGR“, piId);
    return result;
}
```

Algorithmus 6-1: Zurückstellen gesperrter Aufträge (verkürzte Darstellung).

Vier verschiedene Paare solcher Sperren wurden für unterschiedliche Zwecke vorgesehen.

1. startAction / endAction

   Hierbei handelt es sich um einen Zähler, der festhält, wie viele Protokolle (d.h. Migrationen oder dynamische Änderungen) begonnen wurden. Ein Server kann erst dann regulär heruntergefahren werden, wenn alle begonnen Aktionen beendet oder abgebrochen wurden. So wird sichergestellt, daß ein Protokoll nicht wegen eines beendeten Servers inmitten des Ablaufs unterbrochen wird und die im Rahmen des Protokolls beantragten Sperren auf anderen Servern wieder aufgehoben werden.

2. waitFor / stopWaiting

   Diese Sperre wird zum Zurückstellen von Client-Requests verwendet. Der Beginn einer dynamischen Änderung, das Einholen der vom Client benötigten Information, wird so in zwei Phasen aufgeteilt. In der ersten Phase aktualisiert der Server seine Zustandsinformation durch Anfragen bei den anderen aktuell an der Ausführung beteiligten Servern. Danach gelangt der ursprüngliche Client-Request wieder in die Queue und kann wie bisher im nicht verteilten Fall beantwortet werden.

3. p / v (exklusive Sperre)

   Mit dieser exklusiven Sperre wird unter anderem die gleichzeitige Durchführung eingehender Migrationen einer Prozeßinstanz verhindert. Sie findet sowohl Verwendung bei regulär eingehenden Migrationen, als auch wenn die Zustandsinformation eines Prozeßgraphen im Vorfeld einer dynamischen Änderung aktualisiert wird. Durch Verwendung einer anderen eindeutigen ID kann dieser Sperrentyp für viele weitere Zwecke, in denen gegenseitiger Ausschluß gefordert ist, verwendet werden.

4. p / v (mehrere Leser, nur ein Schreiber)

   Um mehrere Migrationen einer Prozeßinstanz zum Beispiel in unabhängigen parallelen Zweigen gleichzeitig zu erlauben, jedoch Migrationen und die endgültige Durchführung dynamischer Änderungen gegenseitig auszuschließen, wurde dieser Sperrentyp implementiert. Migrationen werden dabei als „Leser“, dynamische Änderungen als „Schreiber“, welche den kritischen Abschnitt nur exklusiv betreten können, betrachtet. Die Leser haben Priorität. Erst wenn der letzte Leser den kritischen Abschnitt verläßt, wird ein Schreiber aufgeweckt. Verläßt dieser den kritischen Abschnitt, weckt er alle wartenden Leser auf. Sind davon keine vorhanden, kann ein anderer Schreiber aufgeweckt werden.

6.3.3 Zusätzliche Fehlerbehandlung

Zur Vermeidung von inkonsistenten Zuständen, die ein Fortführen einer Prozeßinstanz unmöglich machen würden, ist eine korrekte Behandlung auftretender Fehler notwendig. Im ADEPT-Prototyp werden Aufträge an andere Server mittels Java-RMI verschickt und dort in eine Queue gestellt. Damit wird wie bei RPC bestätigt, ob der Auftrag angekommen ist, die Bearbeitung selbst erfolgt asynchron. Jede Ausführung eines Teilauftrages wird bei erfolgreicher Durchführung auf dem jeweiligen Server mit einem Commit der zugehörigen DB-Transaktion beendet.

Mögliche Fehler

Folgende Fehler können dabei auftreten:

1. Ein Server, der einen Auftrag entgegen nehmen soll, ist nicht verfügbar, weil auf dem Ziel-Rechner kein ADEPT-Server läuft, dieser gerade heruntergefahren wird oder ein Netzwerkproblem aufgetreten ist. Damit kann der Auftrag nicht entgegengenommen werden. Der „Client“, d.h. ein anderer Server, kann dies erkennen, da er entweder keine Auftragsbestätigung für den RMI-Aufruf bzw. eine Fehlermeldung erhält oder eine RMI-Exception auftritt. Die RMI-Exception wird abgefangen und in eine Fehlermeldung an die Methode, welche die Kommunikation durchführen sollte, umgesetzt, so daß der Server dadurch weiter laufen kann.

2. Während der Ausführung eines Auftrags tritt auf einem der Server ein Fehler auf. Mit Fehler ist hier ein technisches Problem gemeint, z.B. eine Exception aufgrund eines Programmierfehlers. Aufträge, die aufgrund bestimmter Voraussetzungen an den Zustand einer Prozeßinstanz nicht ausgeführt werden können, werden im Rahmen des Protokolls ordnungsgemäß abgebrochen und gelten nicht als Fehler.

3. Einer der beteiligten Server stürzt während der Abarbeitung eines Protokolls ab. Dabei kann es sich um den Initiator der Aktion oder einen der anderen Beteiligten handeln. Solche Fehler führen bei demjenigen Server, der auf die Bestätigung eines RMI-Aufrufes wartet, zu einem Fehler wie unter 2. genannt, oder aber sie werden nicht erkannt, wenn der andere Server sich bei seinem Absturz schon in der Auftragsbearbeitung befindet.

Behandelte Fehler

Eine der häufigsten Fehlerquellen in einem ansonsten einwandfrei funktionierenden verteilten System ist, daß ein Server, zu dem man einen Auftrag schicken möchte, gerade nicht erreichbar ist. Im Prototyp wird dieser Fehler dadurch abgefangen, daß jedes Protokoll in der Anfangsphase, d.h. beim erstmaligen Kontaktieren eines Servers, abgebrochen wird, wenn einer der Server nicht bereit ist, einen Auftrag entgegenzunehmen.

Beim Abbrechen eines Auftrages wird dafür gesorgt, daß bisher für diesen Auftrag gesetzte Sperren auf dem Initiator und anderen Servern wieder zurückgenommen werden. Ansonsten könnte die Ausführung der Instanz nicht fortgeführt werden, da Sperren im System verbleiben würden, die nicht mehr freigegeben werden. Konkret bedeutet dies in den beiden Protokollen zu Migrationen und Dynamischen Änderungen:

- Migration: Hierbei handelt es sich um eine vom System ausgelöste Aktion, die nach dem Beenden eines Arbeitsschrittes notwendig werden kann. Ausgehende Migrationen werden in einer eigenen Transaktion durchgeführt. So wird vermieden, daß die mit dem Beenden des Arbeitsschrittes verbundene Transaktion zurückgesetzt werden muß, nur weil gerade ein Zielserver nicht erreichbar ist. Für einen Benutzer wäre das nicht vermittelbar. Zudem sollen die von ihm berechneten Daten möglichst bald in der Datenhistorie abgelegt werden.

  Migrationen zu jedem Zielserver werden einzeln betrachtet. Wenn ein Zielserver nicht erreichbar ist, können die noch ausstehenden Migrationen später durch einen besonderen Befehl des WF-Administrators an den Server bestimmt und wiederholt werden. Dies gilt auch für die mit einer Migration verbundenen Aktionen wie das Registrieren bei einem Koordinator, falls der Zielserver nicht selbst berechnet werden kann, die dann ebenfalls wiederholt werden.

  Migrationen, die aufgrund eines Serverabsturzes mitten im Protokoll nicht durchgeführt werden, können mit geringem Zusatzaufwand ebenfalls wiederholt werden. Dies liegt daran, daß die bei einer Migration zu übertragende Information auf der Grundlage darauf, welche Information der Zielserver schon kennt, berechnet wird. Selbst wenn in dessen Datenbank also schon Historien-information eingespielt wurde, entstehen keine Inkonsistenzen. Zur Wiederaufnahme der Migration in einem solchen Fall muß ein Quellserver nach dem wieder Hochfahren in seiner Datenbank nach Kanten im Migrations-Status MIGRATING suchen und diese auf MIGRATION zurücksetzen. Eine Referenz zu den Zustandsübergängen dieses Kantenattributs findet sich im Anhang A. Die Sperre auf dem Zielserver für eingehende Migrationen MIGR muß zunächst freigegeben werden, falls sie vom abgestürzten Quellserver beantragt wurde, wodurch auf dem Zielserver eventuell wartende andere Migrationen angestossen werden. Vom Startserver gehaltene Sperren, die vom abgestürzten Quellserver beantragt wurden, müssen ebenfalls aufgehoben werden. Danach kann das Migrationsprotokoll regulär von vorne begonnen werden.

  Um zu verhindern, daß auf einem Server ein Teil der eingehenden Migrationen blockiert ist, weil ein anderer abgestürzter Quellserver noch nicht wieder hochgefahren ist, können die Sperren MIGR periodisch überprüft werden. Falls sie von einem Server beantragt wurden, der nicht verfügbar ist, können sie zusammen mit den korrespondierenden Sperren des Startservers aufgehoben werden. Auch dynamische Änderungen sind dann nicht mehr unnötig blockiert.

  Bei einem Absturz des Zielservers kann der Quellserver zur Wiederaufnahme des Protokolls genauso verfahren. Es muß jedoch darauf geachtet werden, daß integrateHistory-Aufrufe, die aus einem calcHistory-Aufruf noch vor dem Absturz resultieren, ignoriert werden, damit keine Historieneinträge mehrfach angehängt werden[46]. Hierfür kann ein Zähler verwendet werden, der persistent festhält, wie oft ein Server gestartet wurde.

  Entsprechendes gilt für den Startserver, auch er darf nach einem Absturz und dem damit verbundenen Verlust der Sperren keine „alten“ Teilaufträge mehr akzeptieren. Eine andere, meist jedoch nicht akzeptable Möglichkeit ist die, nach dem Absturz eines Servers alle gerade in irgendeinem Protokoll beteiligten Server neu zu starten.

[46] Es wird die optimierte Variante zum Zusammenführen von Historie angenommen, bei der auf Duplikatfilterung verzichtet wird.

- Dynamische Änderung: Im Gegensatz zu einer Migration ist eine Änderungsoperation eine direkt vom Benutzer bewußt eingeleitete Aktion. Sie wird komplett abgebrochen, wenn zu Beginn ein zur Durchführung benötigter Server nicht erreichbar ist. Hier ist somit keine automatische Wiederholmöglichkeit vorgesehen. Eine Wiederholung kann jedoch auf Benutzerwunsch, d.h. durch erneute Aktion bzw. dem Versuch die Aktion erneut auszuführen veranlaßt werden. Diese Vorgehensweise sollte vermittelbar sein, da die Durchführung der Änderung im Gegensatz zur Ausführung eines Arbeitsschrittes sehr wohl von anderen Servern abhängt (zumindest wenn sog. „lokale Änderungen" nicht betrachtet werden, dann tritt aber auch kein Fehler wegen eines nicht erreichbaren anderen Servers auf, da ja keiner kontaktiert werden muß!).

Nicht behandelte Fehler

Fehler, die daraus entstehen, daß ein Server später im Verlauf des Protokolls ausfällt, während er einen Auftrag ausführt, werden nicht behandelt. Es verbleiben möglicherweise Sperren im System bzw. es gehen Sperren verloren, und die Prozeßinstanz kann nicht mehr korrekt ausgeführt werden. Um dennoch eine Fortführung zu ermöglichen, wären persistente Sperren und ein Mechanismus notwendig, der die nicht mehr benötigten Sperren beseitigt.

Das gleiche gilt im Prototyp für während der Ausführung eines Auftrages auftretende technischen Fehler. Man könnte zwar in einem solchen Fall eine Meldung an den Initiator des Auftrages verschicken, daß der Auftrag abgebrochen werden soll. Ohne Verwendung z.B. eines 2PC-Protokolls[47], mit dem es auch möglich ist, Zustandsänderungen von noch nicht gemeinsam mit commit abgeschlossenen Transaktionen auf anderen Servern wieder rückgängig zu machen, macht eine eigenständige Behandlung solcher Fehler allerdings keinen Sinn. Sie ist unvollständig und hilft nur in einem Teil der Fehlerfälle, nämlich dann, wenn noch keine Änderungen an den Graphen auf anderen Servern vorgenommen wurden. Anstelle eines 2PC wären natürlich auch Operationen denkbar, die die zuvor ausgeführten Transaktionen rückgängig machen, d.h. kompensieren würden. Beispielsweise müßte eine auf einem anderen Server bereits eingebrachte dynamische Änderung wieder zurückgenommen werden, wenn der Initiator den gesamten Änderungsvorgang aufgrund eines Fehlers abbricht.

Ebenso könnte allerdings auch verlangt werden, daß die Änderung auf allen Servern, bei denen die Änderung schon eingebracht wurde, bestehen bleibt. Bei den anderen Servern, auf denen die Änderungseinbringung an technischen Problemen scheiterte, muß die Änderung nach dem Beheben des Problems zwingend eingebracht werden, bevor die Ausführung fortgesetzt wird. Die Änderung kann in jedem Fall angewendet werden, da ihre Zulässigkeit vom Initiator verifiziert wurde. Auch mit dieser Methode kann auf ein 2PC verzichtet werden.

Herunterfahren eines Servers

Wenn nicht persistente Sperren eingesetzt werden, darf ein Server erst dann heruntergefahren werden, wenn keine Sperren mehr gesetzt sind. Anderenfalls könnte ein Server, der sich noch in der Auftragsbearbeitung befindet, unbeabsichtigt heruntergefahren werden, so daß er von anderen keine Teilaufträge mehr annehmen kann und somit als ausgefallen gelten würde. Laufende Protokolle wären unterbrochen, und für solche Fälle ist derzeit keine Fehlerbehandlung vorgesehen.

Außerdem ist es auch möglich, daß an manchen Sperren noch durchzuführende Client-Aufträge warten, die nach Freigabe der Sperre wieder in die Eingangsqueue gestellt werden müssen. Diese Aufträge würden verloren gehen.

Erst nachdem alle Aufträge bearbeitet wurden, an denen ein bestimmter Server beteiligt ist, darf dieser Server heruntergefahren werden. In der Implementierung wird das Herunterfahren automatisch verhindert, falls noch Aufträge aktiv sind.

In dem Zustand, indem der Server sich auf das Herunterfahren vorbereitet, darf er keine neuen Aufträge mehr annehmen. Es muß jedoch gewährleistet werden, daß er diejenigen Teilaufträge annimmt, die zur Fertigstellung eines bestehenden Auftrages führen. Ebenso müssen natürlich Storno-

[47] Für Java wird zur Zeit noch keine geeignete Implementierung angeboten.

Aufträge und Aufträge zum Aufheben von Sperren zugelassen werden. Auch diese Anforderung wurde berücksichtigt.

6.3.4 Sonstiges

Für jeden Knoten sind pro Iteration mehrere Ausführungshistorieneinträge möglich. Ein Eintrag existiert für jeden Zustandswechsel des Knotens, z.B. von NOT_ACTIVATED zu ACTIVATED usw. Im Verfahren zur Berechnung der Migrationspunkte (vgl. Algorithmus 3-3) wird davon ausgegangen, daß zu einem Knoten bei einer Migration entweder alle Historieneinträge übertragen werden oder keine. Bei der Aktualisierung des Zustandes im Vorfeld von dynamischen Änderungen muß diese Bedingung jedoch nicht erfüllt sein. Zu Aktivitäten, die zwar schon aktiviert wurden, jedoch noch nicht abgeschlossen sind, wird zunächst lediglich ein Teil der insgesamt zu erwartenden Historieneinträge übertragen. Um redundante oder unvollständige Übertragung zu verhindern, müssen die Migrationspunkte daher in feineren Einheiten angegeben werden.

Ein Migrationspunkt dient somit beispielsweise zu folgender Mitteilung: „Die Einträge zu Vorgängerknoten von Knoten A in Iteration 2 und die drei ersten Einträge zu Knoten A in Iteration 2 sind auf dem Zielserver bereits bekannt“. Bei der Berechnung der Vorgänger unter Berücksichtigung der Historie, also z.B. dem abzuziehenden Teil der Historie, wird die feinere Granularität der Migrationspunkte entsprechend berücksichtigt.

6.4 Ansatzpunkte für zukünftige Erweiterungen

Sowohl in Bezug auf die Performanz als auch auf die Funktionalität sind natürlich Verbesserungen und Erweiterungen möglich. Einerseits konnten nicht alle Optimierungsmöglichkeiten berücksichtigt werden, da vor allem ein stimmiges Gesamtsystem erstellt und nicht Details bis ins letzte optimiert werden sollten. Zum anderen handelt es sich um einen Prototyp, bei dem noch nicht alle existierenden Konzepte vollständig integriert sind, der jedoch in Teilaspekten erweiterbar sein muß.

6.4.1 Performance-Optimierungen

- Um beim Einspielen von mehrfachen Änderungen das Anpassen der Graphstruktur zu beschleunigen, sollten die Änderungshistorieneinträge nacheinander auf einem im Hauptspeicher gehaltenen Graphen durchgeführt werden. Bislang wird jede Änderungsoperation einzeln nachgeführt und der komplette Graph anschließend wieder in der Datenbank gespeichert. Zur Umsetzung der Optimierung sind geringfügige Änderungen in den Softwareschichten zwischen dem Aufruf einer Änderungsoperation und dem letztendlichen Durchführen der Änderung auf dem Graphen im Hauptspeicher notwendig.

  Eine noch performantere Durchführung kann erreicht werden, wenn statt der komplexen Änderungsoperation die zugehörigen Elementaroperationen übertragen und auf den nachzuführenden Graphen angewandt werden. Das Lesen des kompletten Graphen und Neuberechnen der Änderung fällt damit völlig weg. Hierfür müssen bei dynamischen Änderungen die Elementaroperationen in die Historie gespeichert werden, was derzeit zwar vorgesehen, aber noch nicht realisiert ist.

- Weitere in den Konzepten bereits vorgesehene Optimierungen können umgesetzt werden. So können gleichzeitige Migrationen zum gleichen Server über mehrere Kanten zu einer Migration zusammengefasst werden. Dies betrifft Migrationen nach einem parallelen Verzweigungsknoten sowie zu einem neu eingefügtem Knoten. Bislang werden die Migrationen für jede Kante nacheinander durchgeführt. Das Kommunikationsvolumen wird bei damit zwar nur geringfügig reduziert, da durch das Verfahren ja bereits redundante Übertragung ausgeschlossen wird. Allerdings wird die Anzahl an Kommunikationen und in nicht unerheblichem Maße der Berechnungsaufwand reduziert.

- Im Laufe des Fortschreitens einer Prozeßinstanz kann als Koordinator ein anderer, näher an den Servern der momentan ausgeführten Knoten liegender Server als der Startserver verwendet werden. Es ist darauf zu achten, daß der gültige Koordinator von allen an der Ausführung beteiligten Servern zu jedem Zeitpunkt eindeutig bestimmt werden kann und der bisherige

Koordinator seine Informationen an den neuen Koordinator weitergibt. Auch ist eine explizite Anfrage nach der Menge *ActiveServers*, um die Zustandsinformation des Graphen zu aktualisieren, in einigen Fällen nicht nötig, wenn gerade ohnehin nur ein Server an der Ausführung beteiligt ist. Bei einer Migrationen in einer reinen Sequenz kann teilweise Protokoll-Overhead eingespart werden, da es hierbei nicht notwendig ist, die Menge *ActiveServers* explizit mit einem kritischen Abschnitt zu schützen.

- Schließlich kann das in Abschnitt 3.6.3 beschriebene optimierte Verfahren zum Holen von großen Datenelementen integriert werden, sobald im Prototyp solche Datenelemente vorgesehen werden.

6.4.2 Erweiterung der Funktionalität

Wenn weitere dynamische Änderungsoperationen unterstützt werden sollen, müssen diese auch im verteilten Fall durchgeführt werden können. Bei der Implementierung muß lediglich ein entsprechender Änderungshistorieneintrag mit berücksichtigt werden. Dann kann die neue Funktion analog der bereits vorhandenen Änderungsoperationen aufgerufen werden.

Für das dynamische Rücksetzen wurden insoweit Vorkehrungen getroffen, als daß die benötigte Historieninformation auf den entsprechenden Servern bereit gehalten wird. Die Anker zu dynamischen Änderungsoperationen werden in den Ausführungshistorien aller zum Zeitpunkt der Änderung aktiven Server verzeichnet. Für das Rücksetzen selbst müssen die in der Historie protokollierten Server wieder angesprochen werden, da nur sie die Kompensation der Aktivitäten vollständig durchführen können. Ein entsprechendes Verfahren, das möglicherweise optimiert werden kann, ist zu erstellen.

Erweiterte Serverzuordnungsausdrücke können ausgewertet werden, wenn der primitive Parser diesbezüglich erweitert wird. Auch könnte im WF-Editor eine Konsistenzprüfung der Ausdrücke durchgeführt werden, so daß garantiert nur Vorgängeraktivitäten referenziert werden. Noch besser wäre eine automatische Berechnung der Serverzuordnungen auf der Grundlage der WF-Graphen, des Organisationsmodells und weiterer Information beispielsweise in Bezug auf die Kommunikationskosten wie in [BD00] vorgeschlagen.

6.4.3 Ideen zur weitergehenden Fehlerbehandlung

Wenn zur Kommunikation persistente Queues verwendet werden, kann auf die Fehlerbehandlung bei nicht erreichbaren Servern verzichtet werden. Im Fall eines nicht erreichbaren Servers verbleibt ein Auftrag solange in der Ausgangsqueue des Auftraggebers, bis der Auftragnehmer diesen in seiner Eingangsqueue hat. Er garantiert, daß er den Auftrag durchführen kann und eine Antwort in seine Ausgangsqueue stellen wird. Tritt bei der Durchführung des Auftrags ein technischer Fehler auf, wird die Transaktion, mit der der Auftrag aus der Warteschlange genommen wurde, zurückgesetzt und der Auftrag muß solange wiederholt ausgeführt werden, bis er fehlerfrei durchgeführt wurde.

Allerdings kommt es mit einem solchen System unter Umständen zu einer stark verzögerten Auftragsdurchführung. Außerdem bringt diese Methode gegenüber der nicht persistenten Speicherung von Aufträgen Nachteile in der Performanz mit sich. Dafür, daß dieselben Aufträge nicht mehrfach bearbeitet werden, muß das Persistent-Queue-System sorgen. Dies kann beispielsweise mit einer Auftrags-ID dadurch gewährleistet werden, daß die Eingangsqueue nur solche Aufträge annimmt, die einen höheren Zählerwert haben als bisher eingetroffene, die ja unter Umständen nicht mehr in der Queue sind, wenn sie schon erfolgreich bearbeitet wurden.

Bei der Umstellung auf persistente Queues sollten auch persistente Sperren eingesetzt werden. Nur so kann ein stimmiges Gesamtsystem erhalten werden.

Der geringste Programmieraufwand zur Fehlerbehandlung müßte betrieben werden, wenn ein 2PC zur Verfügung stände. Hierbei kann beispielsweise eine gesamte Änderungsoperation, d.h. alle Teilaufträge innerhalb des Protokolls, als eine Transaktion betrachtet werden. Alle im Laufe des Protokolls vorgenommenen Veränderungen können bei einem auftretenden Fehler zurückgesetzt werden. Das System kümmert sich automatisch um die Behandlung ausgefallener beteiligten Server, egal in welcher Rolle diese agieren. Im Vergleich zu der Lösung mit persistenten Warteschlangen wird die Performanz hierbei allerdings noch stärker eingeschränkt.

7 Zusammenfassung und Ausblick

Zum Abschluß werden noch einmal die wesentlichen Ergebnisse dieser Diplomarbeit zusammengefaßt. Unterschiede und Gemeinsamkeiten im Vergleich mit verwandten Arbeiten werden kurz dargestellt und danach ein Ausblick gegeben.

7.1 Zusammenfassung

In der Diplomarbeit wurden Verfahren entwickelt, um flexible Workflows auch im Fall der Verwendung mehrerer WF-Server zu ermöglichen. Zunächst wurden mit der Betrachtung der verteilten Ausführung die Grundlagen dafür gesetzt. Naive, in Bezug auf die Kommunikationskosten teure Lösungen lassen sich fast immer vermeiden. Bei einer Migration muß nie die komplette Prozeßinstanz übertragen werden. Mit der Übertragung von Teilen der Ausführungshistorien werden deutliche Einsparungen erzielt, redundante Übertragung kann hier sogar vollständig vermieden werden.

Bei den Datenelementen erscheint eine getrennte Behandlung je nach Größe sinnvoll. So kann ein weiterer Kostenfaktor, die Anzahl an Kommunikationen, minimiert werden, wenn kleine Datenelemente redundanzfrei mit der Ausführungshistorie übertragen werden. Eine Auswirkung ergibt sich letztendlich aber auch auf das Kommunikationsvolumen. Zur Übertragung großer Datenelemente lohnt sich eine Optimierung nach dem „Holen-Verfahren“, denn so kann unnötige Übertragung vermieden werden. Vielfach müssen Datenelemente nicht in allen Versionen verfügbar gemacht werden, zum Teil werden sie vom Migrationszielserver auch gar nicht benötigt. Oft kann erst dann, wenn eine Verzweigungsentscheidung getroffen wurde, bestimmt werden, welche Daten tatsächlich gelesen werden. Durch das Abholen großer Daten erst unmittelbar vor Aktivierung einer Aktivität wird unnötige Kommunikation vermieden.

Variable Serverzuordnungen dienen dazu, ohne hohen Berechnungsaufwand erst zur Laufzeit einer konkreten WF-Instanz einen möglichst günstigen Server für die Ausführung bestimmter Aktivitäten festzulegen. Dieses Konzept wird in der Diplomarbeit berücksichtigt. Manche der für den statischen Fall entwickelten Verfahrensalternativen können fast unverändert übernommen werden. Somit mußte lediglich für ein speziell bei variablen Serverzuordnungen auftretendes Problem eine komplett neue Lösung entwickelt werden: die Migration zu einem Joinknoten, dessen Server nicht von allen Vorgängern bestimmt werden kann.

Zur Durchführung von dynamischen Änderungen wurde ein Protokoll erarbeitet, das eine Synchronisation mit den aktuell an der Ausführung beteiligten Servern realisiert. Auch hier erfolgt die erforderliche Datenweitergabe in Form einer Historienübertragung. Zur Ausbreitung einer dynamischen Änderung wird bei deren Durchführung und bei Migrationen zum jeweiligen Server der dort benötigte Teil der Änderungshistorie übertragen. Eine mögliche Optimierung hierbei ist, die Änderung zum Durchführungszeitpunkt nur mit einer Teilmenge der aktuell an der Ausführung beteiligten Server zu synchronisieren. Dies erfordert deutlichen Mehraufwand zur Bestimmung der Menge dieser Server, kann aber Vorteile bringen, wenn beispielsweise ein Teilzweig des Workflow gerade von einem weit entfernten Server kontrolliert wird und sich die lokale Änderung nur auf diesen Zweig auswirkt. Die Verfügbarkeit des WfMS wird erhöht, da die Server, die die anderen Teilzweige kontrollieren, nicht in die lokale Änderung einbezogen werden. Je nach den Bedürfnissen eines Unternehmens kann im WfMS ein passendes, mehr oder weniger striktes, Synchronisationsverfahren eingesetzt werden.

Am Prototyp des ADEPT-WfMS wurden umfangreiche Erweiterungen vorgenommen, so daß jetzt ein Demosystem zur Veranschaulichung der wesentlichen Konzepte der verteilten Ausführung in Verbindung mit dynamischen Änderungen verfügbar ist.

7.2 Verwandte Arbeiten

Um einen Überblick über wichtige verteilte WfMSe zu bekommen verweise ich zunächst auf [BD98a] und [BD99b]. Darin werden verschiedene Architekturen für skalierbare WfMSe miteinander verglichen. In vielen der Systeme sind wichtige Aspekte wie z.B. eine globale Rollenauflösung oder Migrationen von Workflows nicht berücksichtigt. Mit Ausnahme von WIDE [CGP+96] weist keines der dort betrachteten anderen Systeme eine Möglichkeit für dynamische Änderungen auf. Verweise zu Arbeiten, die sich mit dieser Thematik beschäftigen, finden sich in [Hen97] oder [RBD99].

In einigen der Systeme sind weitere Einschränkungen enthalten, die für ADEPT nicht gelten. MOBILE [SNS99] erreicht die Verteilung, in dem ein Workflow in Subworkflows aufgeteilt wird und diese dann auf unterschiedlichen Servern ausgeführt werden können. An den Ein- und Ausgangspunkten der Subworkflows ist immer eine Synchronisation mit dem übergeordneten Server notwendig, der so besonders stark belastet wird. Der Server, der die Ausführung eines sehr großen Workflows auf der höchsten Ebene kontrolliert, kann also leicht überlastet werden.

INCAS [BMR94] und Exotica/FMQM [AMG95] sind voll verteilte WfMSe. Hier werden die Daten schrittweise verteilt. Bei INCAS wird dabei ein hohes Datenvolumen erzeugt, da immer die komplette Beschreibung der Prozeßinstanz mit allen Version der Datenelemente weiter geschickt wird. Es wird keine Aussage über Verfahren zur Vermeidung redundanter Übertragung gemacht.

Einen völlig anderen Ansatz verfolgt das CORBA-basierte METEOR$_2$ [DKM+97]. Bei diesem System können die Objekte, d.h. Daten und Tasks, zwar anfänglich gezielt auf verschiedene Server verteilt werden, aber es gibt keine Migrationen. Zur Laufzeit werden nur Referenzen auf Daten übertragen. Bei häufigen entfernten Zugriffen während der Ausführung entsteht damit ein hohes Kommunikationsvolumen. Die Berücksichtigung dynamisch adaptierbarer Workflows ist für zukünftige Forschungsarbeiten vorgesehen. Auch BPAFrame [SM96] basiert auf CORBA.

Das WfMS Mentor [MWW+98] basiert auf der verteilten Ausführung von State- und Activity-Charts. Ähnlich wie in der hier vorliegenden Arbeit werden auch dort unterschiedliche Verfahren zur Ausführung diskutiert, die einen unterschiedlichen Optimierungsgrad in Bezug auf die Anzahl und die Größe der übertragenen Nachrichten aufweisen. Ein grundlegendes Problem des beschriebenen Ansatzes ist eine fehlende echt parallele Ausführung verschiedener Zweige, die von unterschiedlichen Servern kontrolliert werden. Aufgrund der Semantik der State- und Activity-Charts wird auf jeder Partition die gleiche Anzahl an Schritten ausgeführt. Neue Schritte können immer erst dann gestartet werden, nachdem auf allen Partitionen die vorangehenden Schritte beendet wurden. Tatsächlich bleibt die parallele Ausführung so immer auf Gruppen von parallelen Aktivitäten beschränkt, die nur jeweils eine Aktivität pro parallelem Zweig enthalten. Es ist nicht möglich, auf einem Server die Sequenz der beiden Aktivitäten A und B auszuführen, während ein anderer Server C ausführt. Außerdem können nur sehr einfach strukturierte Graphen automatisch durch das dort entworfene optimierte Synchronisationsverfahren unterstützt werden. Bei komplexen Graphen erfolgt ein hohes Maß an redundanter Kommunikation.

Viele der Systeme verwenden für alle Datenbankzugriffe ein Two-Phase-Commit-Protokoll [Dad96]. Die Kommunikation mit anderen Servern wird so sehr teuer und empfindlich gegenüber zeitweiligen Serverausfällen. ADEPT stellt geringere Anforderungen an das System. Es kommt mit einem zuverlässigen Kommunikationssystem mit exactly-once Semantik und persistenten Queues mit lokalen Transaktionen aus. WIDE [CGP+96] bietet ein mehrstufiges Transaktionskonzept mit Sagas, Kompensationsmechanismen und verschachtelten Transaktionen. Auf der untersten Ebene wird der Transaktionsmechanismus des zugrundeliegenden DBMS benutzt.

Teilweise finden sich in der Literatur Lösungsansätze zu speziellen Problemstellungen. Beispielsweise behandelt Exotica FMDC [AGK+96] Anforderungen an ein WfMS mit mobilen Klienten, die für das WfMS zeitweise nicht verfügbar sind. Durch zukünftige neue Anwendungsformen von WfMS werden sicherlich weitere besondere Aufgabenstellungen zu lösen sein.

7.3 Ausblick

Neben dem weiteren Ausbau des ADEPT-Prototyps (vgl. Abschnitt 6.4) können auch die bestehenden Konzepte erweitert werden. So wurde das dynamische Rücksetzen im verteilten Fall zwar vorgesehen, ein Verfahren zur Durchführung muß aber noch entwickelt werden, bevor es in den Prototyp aufgenommen werden kann. Zahlreiche weitere Forschungsgebiete wie z.B. Interworkflowabhängigkeiten können die besonderen Aspekte im Zusammenhang mit Verteilung berücksichtigen.

Zwischen den vorgeschlagenen Varianten, beispielsweise zur Übertragung der WF-Kontrolldaten, kann ein detaillierter Kostenvergleich durchgeführt werden. Dazu sind jedoch zusätzliche Daten in Bezug auf das verwendete System und über exemplarische Workflows notwendig. Dann könnten die Kosten für Abfrage- und Lesezugriffe des DBMS, für den Datentransfer und auch Rechenoperationen exakt bestimmt, gewichtet summiert und einander gegenübergestellt werden. Insbesondere die Bestimmung der Kosten für die Ausführung von Methoden unterschiedlicher Java-Objekte ist sicher nicht trivial und hängt außerdem noch von der eingesetzten Java-Implementierung und der Plattform ab.

Wenn eine detaillierte Auflistung der Kosten erstellt wurde, können die Werte zu den Kommunikationskosten bei der Berechnung einer günstigen Verteilung berücksichtigt werden. Dabei können auch Szenarien identifiziert werden, bei denen das Kommunikationsaufkommen so gering ist, daß auch ein einfacheres zentrales WfMS zum Einsatz kommen kann und solche Fälle, in denen ein verteiltes System zur Bewältigung der Last unbedingt nötig ist.

In eine andere Richtung gehen Überlegungen zu Problemen, die beispielsweise bei der Kooperation oder Fusion von Unternehmen und damit auch der Integration ihrer Informationssysteme auftreten. Jetzt müssen verschiedene WfMS und möglicherweise auch Workflows zusammenarbeiten und/oder ineinander übergeführt werden.

Ein Ziel der WfMC [WMC99] ist es, über einheitliche Schnittstellen Interoperabilität zwischen den WfMS verschiedener Hersteller zu erreichen. Die Schnittstellen können auch dazu benutzt werden, die WfMS-Funktionalität durch Kooperation mit Projektmanagement-Tools oder einem Dokumentenmanagementsystem zu erweitern. Da ein WfMS in der Regel aus Komponenten aufgebaut wird, bieten sich ebenfalls Erweiterungsmöglichkeiten. Einzelne Komponenten wie z.B. das Organisationsmodell können so durch Standardlösungen oder Eigenentwicklungen ersetzt werden.

Ein auf Basis von CORBA[48] realisiertes WfMS ist gegenüber diesen Anforderungen von sich aus schon relativ gut gerüstet. Semantische Probleme bei der Kooperation müssen hingegen auf einer anderen Ebene gelöst werden. Auch die Verwendung von XML[49] beispielsweise zur Spezifikation von Ablaufbeschreibungen ist sicher ein richtiger Schritt auf dem Weg zu mehr Offenheit.

Die Entwicklung von flexiblen und in breiten Anwendungsszenarien einsetzbaren Standard-WfMS bietet die Chance, zu einer noch besseren Akzeptanz und damit einer weiteren Ausbreitung der WF-Technologie zu gelangen.

---

[48] Common Object Request Broker Architecture, Näheres siehe [CDK95] oder http://www.corba.org.

[49] Extensible Markup Language, Näheres siehe http://www.w3.org/XML/.

Anhang

A Zustandsübergänge des Migrationstate einer Kante

Der Abbildung A-1 können die möglichen Werte des Attributs Migrationstate einer Kante entnommen werden. In Tabelle A-1 werden die Zustände und die Übergangsbedingungen genauer beschrieben. Die Zustände gelten jeweils nur für die aktuelle Schleifeniteration. Zu Beginn einer neuen Iteration werden analog dem Setzen des Kantenstatus auf NOTSIGNALED alle im Schleifenkörper enthaltenen Kanten wieder als NOTSET betrachtet (nicht eingezeichnet).

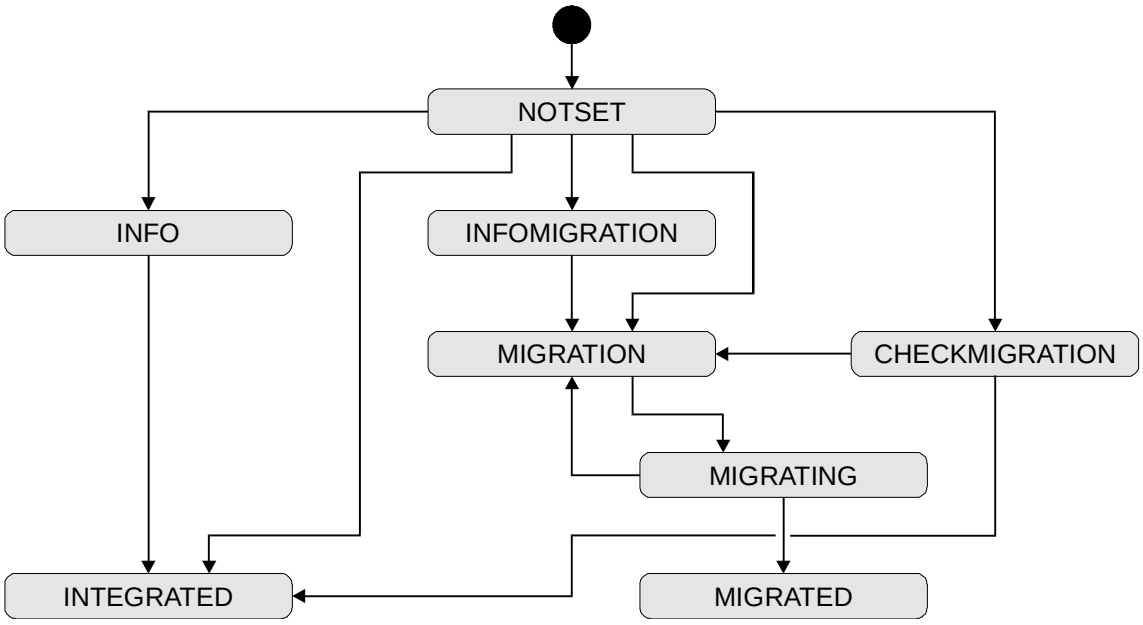


Abbildung A-1: Zustandsübergänge des Migrationstate einer Kante.

Migrationstate	Beschreibung und Bedingungen für den Eintritt in einen Folgezustand
NOTSET	Dies ist der Anfangszustand einer Kante. Er ändert sich beim Markieren der aus einem Knoten ausgehenden Kanten in einen der fünf möglichen Folgezustände. Die Auswahl hängt davon ab, welchem Server der nachfolgende Knoten zugeordnet ist und an welcher Stelle im Graphen sich der Ursprungsknoten befindet. Sollte der Migrationstate CHECKMIGRATION erreicht werden, darf die Kante nicht signalisiert werden.
INFO	Es findet keine Migration statt, aber es muß ein anderer Server über den Migrationszielserver informiert werden. Wenn dies geschehen ist, wechselt der Zustand nach INTEGRATED.
INTEGRATED	Der Endzustand, falls Quell- und Zielaktivität auf dem gleichen Server angeordnet sind.
INFOMIGRATION	Es findet eine Migration statt, zuvor muß jedoch noch ein anderer Server über den Migrationszielserver informiert werden. Wenn dies geschehen ist, wechselt der Zustand nach MIGRATION.
MIGRATION	Der Zielserver, zu dem migriert werden muß, steht fest. Wenn die Migration begonnen wird, wechselt der Zustand nach MIGRATING.
CHECKMIGRATION	Der Server, auf dem der nachfolgende Knoten ausgeführt werden wird, ist

Migrationstate	**Beschreibung und Bedingungen für den Eintritt in einen Folgezustand**
	noch nicht bekannt. Daher wird beim Koordinator um Erteilung einer Nachricht erbeten, sobald der Zielserver feststeht. Beim Erhalt der Nachricht wird die Kante markiert und der Zustand wechselt je nach Zielserver zu INTEGRATED oder zu MIGRATION.
MIGRATING	Eine Migration wird gerade durchgeführt. Wenn eine Migration erfolgreich abgeschlossen wurde, geht der Zustand nach MIGRATED über. Wird eine Migration abgebrochen, wird der Zustand auf MIGRATION zurückgesetzt, damit später ein erneuter Versuch zur Migration unternommen werden kann.
MIGRATED	Der Endzustand, falls Quell- und Zielaktivität unterschiedlichen Servern zugeordnet sind.

Tabelle A-1: Mögliche Werte für den Migrationstate einer Kante.

B Java-Klassen der Verteilungsschicht

Die Verteilungsschicht wurde komplett im Package WFMS.Server.Kernel.DistributionLayer erstellt. Sie besteht aus Klassen, die die in Kapitel 5 beschriebenen Protokolle abwickeln und Klassen, die hierbei benötigte Funktionalität zur Verfügung stellen. Teilweise wird die Funktionalität auch außerhalb des Packages benötigt, z.B. um Serverzuordnungsausdrücke auszuwerten oder Sperren zu setzen.

Im Zusammenarbeitsdiagramm in Abbildung B-1 wird eine Übersicht der Klassen gezeigt. Dem Diagramm ist zu entnehmen, welche Klassen Dienste anderer Klassen in Anspruch nehmen.

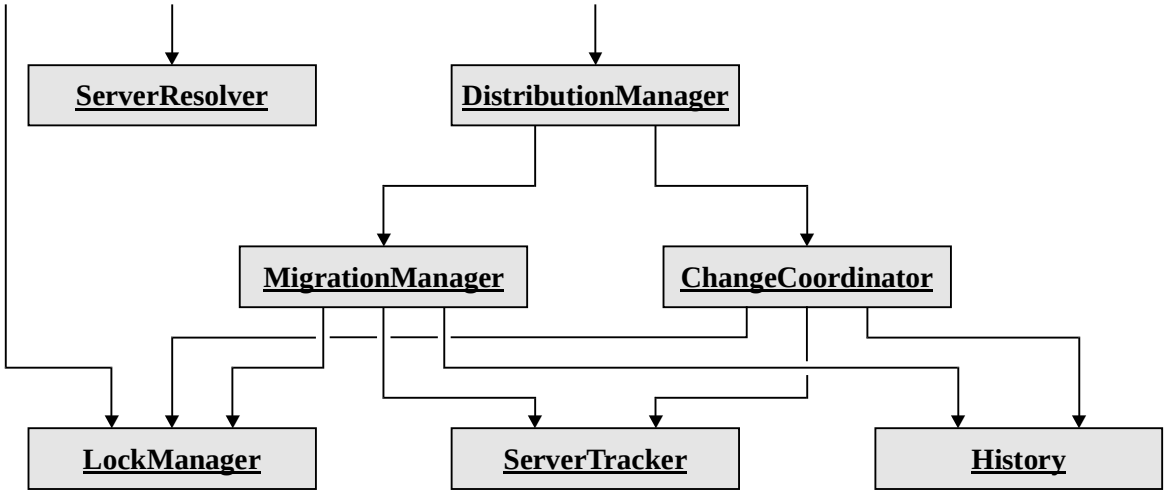


Abbildung B-1: Zusammenarbeitsdiagramm der Klassen aus der Verteilungsschicht.

B.1 DistributionManager

Der DistributionManager ist gewissermaßen die Schaltstelle der Verteilungsschicht. Hier werden eingehende Requests an die zuständigen Dienstklassen MigrationManager und ChangeCoordinator verteilt. Migrationen und dynamische Änderungen werden mit Hilfe des DistributionManager des eigenen Servers angestoßen.

Außerdem befindet sich hier die Methode, mit der Requests an andere Server übermittelt werden. Sie wird von den Dienstklassen benutzt, um Aufträge an andere Server zu schicken und so im Protokoll fortzufahren. Ein Auftrag kann jedoch nicht nur an einen anderen Server geschickt werden, sondern mit einem speziellen OP-Code können auch berechnete Ergebnisse zur Übermittlung an Clients in die entsprechende Queue gestellt werden.

Auch die Transaktionsbearbeitung der Verteilungsschicht ist hier angesiedelt. Jeder Teilauftrag wird innerhalb einer Transaktion durchgeführt.

B.2 MigrationManager

Die Klasse MigrationManager enthält die Methoden, die im Rahmen des Protokolls zur Durchführung von Migrationen benötigt werden. Jede Methode entspricht dabei einem Teilauftrag. Zu Beginn werden die übergebenen Parameter ausgewertet, dann die eigentlichen Aktionen durchgeführt und schließlich, wenn das Protokoll noch nicht beendet ist, mit diesem fortgefahren, indem die berechneten Ergebnisse als neuer Teilauftrag an den betreffenden Server geschickt werden. Das Protokoll zu Migrationen ist in Abschnitt 5.5.2 vollständig beschrieben.

Darüber hinaus sind hier Methoden enthalten, mit denen Prozeß- und Aktivitätenvorlagen zwischen WF-Servern übertragen werden können.

B.3 ServerResolver

In ServerResolver ist die Funktionalität gebündelt, die mit der Verarbeitung von Serverzuordnungsausdrücken und Servernamen zu tun hat.

Mit *evaluateExpression* wird ein Serverzuordnungsausdruck ausgewertet. Handelt es sich bei dem Ausdruck um einen einfachen Servernamen, ist keine Auswertung notwendig. Es wird lediglich überprüft, ob es sich um einen gültigen Servernamen handelt. Bei einem variablen Ausdruck wird versucht, diesen auszuwerten. Dabei wird auch überprüft, ob nicht andere Vorgängerknoten des Zielknotens den Ausdruck nicht auswerten könne, so daß der Wert des berechneten Servers bei einem Koordinator hinterlegt werden muß. In jedem Fall wird ein Migrationsdeskriptor zurückgeliefert. Er enthält die Information, ob an der betreffenden Kante migriert werden muß oder nicht, ob beim Koordinator ein Wert hinterlegt werden muß oder ob noch nicht migriert werden kann, da erst noch auf den Wert für den Zielserver gewartet werden muß.

Die Methoden *appendServerValue*, *getServerValue* und *removeServerValue* dienen dazu, das Attribut NODE_EXECUTIONSERVER eines Knotens um dem Wert des tatsächlichen Servers zu ergänzen. An einen variablen Zuordnungsausdruck wird bei Aktivierung eines Knotens oder beim Einspielen von Historie beispielsweise „->S1“ angehängt, wenn dies das Ergebnis der Auswertung ist. Nach dem Zurücksetzen der Knotenzustände durch eine neue Schleifeniteration muß der Wert entfernt werden, da der Serverzuordnungsausdruck anschließend wieder neu berechnet wird.

Da im WF-Editor bei der Bearbeitung von Prozeßvorlagen unter Umständen noch keine Datenbank-IDs zu den Knoten verfügbar sind, wird dort mit eindeutigen temporären IDs gearbeitet, welche innerhalb der bearbeiteten Prozeßvorlage gültig sind. Die dort eingegebenen Serverzuordnungsausdrücke enthalten daher auch diese IDs. Bei der Instanziierung einer Prozeßinstanz aus einer Vorlage müssen deshalb in den Attributen NODE_EXECUTIONSERVER der Knoteninstanzen mit *replaceTempIDsInNodeExServer* die temporären IDs der Knotenvorlagen durch die tatsächlichen Datenbank-IDs der korrespondierenden Knoteninstanzen ersetzt werden.

getCoordinatorForPI dient dazu, den Koordinator-Server für eine Prozeßinstanz herauszufinden. In der derzeitigen Implementierung wird immer der Startserver zurückgeliefert.

getHomeServerForUser gibt den Server im Teilnetz eines Benutzers zurück.

getHostname gibt für einen symbolischen Servernamen dessen Hostname bzw. die IP-Adresse zurück. Im nicht verteilten Fall wird immer der Hostname des momentanen Servers zurückgegeben.

Mit *readADEPTini* wird die Konfigurationsdatei gelesen, die Tabelle von Serverzuordnungen zu IP-Adressen aufgebaut und eine einfache Konsistenzprüfung durchgeführt. Hier wird auch der für den Server passende Datenbank-Connectstring gesetzt.

B.4 ServerTracker

Die Komponente ServerTracker stellt die Koordinator-Funktionalität bereit, mit der eine Nachfrage nach dem Zielserver einer Migration möglich wird und Benachrichtigungen über einen berechneten Zielserver verschickt werden. Hier sind sowohl das Protokoll (vgl. Abschnitt 4.6, Algorithmus 4-2) als auch die dafür benötigten Datenbankzugriffe implementiert.

Mit *demandForServer* fragt ein Server nach dem Wert für einen Migrationszielserver, wenn er einen Serverzuordnungsausdruck nicht auswerten konnte. Ist der gesuchte Wert bereits gespeichert, erhält der Anfragende sofort eine Antwort. Anderenfalls wird er registriert und bekommt die Antwort zugestellt, sobald dem Koordinator das Ergebnis bekannt ist. Der mit dem Fortfahren in der Ausführung wartende Server kann daraufhin berechnen, ob eine Migration notwendig ist oder der Knoten auf dem gleichen Server bleiben wird und ggf. aktiviert werden kann.

Durch den Aufruf von *storeServer* wird das von einem Server, der einen variablen Serverzuordnungsausdruck auswerten kann, berechnete Ergebnis dem Koordinator mitgeteilt, so daß dieser den Wert zur Verfügung stellen kann.

Außerdem sind in ServerTracker die Datenbankzugriffe zur Bestimmung und Aktualisierung der Menge für *ActiveServers* implementiert.

B.5 LockManager

Vom LockManager werden verschiedene nicht persistente Sperren bereitgestellt (vgl. Abschnitt 6.3.2). Damit verbunden sind Funktionen zur Überprüfung, ob noch Sperren gesetzt sind, und zur Ausgabe eines Debug-Dump über bestehende Sperren. So wird verhindert, daß ein Server heruntergefahren werden kann, wenn noch Protokolle durchgeführt werden.

B.6 History

Hier sind alle Methoden zum Umgang mit der Ausführungs- und Änderungshistorie angesiedelt. Ein History-Objekt repräsentiert einen Ausschnitt aus der Ausführungshistorie einer Prozeßinstanz. Es enthält eine Referenz auf einen Prozeßinstanzgraphen vom Typ WFMS.Server.Kernel.SupportLayer.processinst.ProcessInstance und dessen kompletten PI_ExecutionHistoryEntry Objekten. Im OrderedSet myHistory wird die Teilmenge an verkürzten Historieneinträgen der privaten Klasse exHistoryEntry gehalten, für die das History-Objekt steht.

Auf einer Historie sind Operationen wie *union*, *intersection* und *difference* möglich, die das Objekt selbst unverändert lassen und im Ergebnis ein neues History-Objekt zurückgeben.

latestHistoryEntry gibt den aktuellsten Historieneintrag zurück. Dieser wird beschrieben durch Knoten-ID n, Iteration i und der Nummer des Historieneintrags, d.h. um den wievielten Eintrag zum Knoten n in Iteration i es sich handelt.

Über die Methode *pred_i* lassen sich die Vorgänger eines Knotens unter Berücksichtigung der Ausführungshistorie berechnen (vgl. Algorithmus 2-1). Bei *pred_i* nur mit einem Knoten als Parameter bezieht sich die Berechnung auf den letzten Eintrag zu diesem Knoten. Wenn als Parameter zu *pred_i* ein Migrationspunkt angegeben wird, werden für die Berechnung nur die Historieneinträge bis zu diesem Punkt berücksichtigt. Das Ergebnis ist jeweils wieder ein History-Objekt.

readFromDB liest das ProcessInstance-Objekt und die zugehörigen PI_ExecutionHistoryEntry-Objekte aus der Datenbank. In der umgekehrten Richtung gibt *getPI_ExecHisEntries* die Teilmenge der ExecutionHistoryEntries zurück, für die das History-Objekt steht. *getWrittenSmallDataelements* liefert alle Datenelemente, die während des Historienabschnittes geschrieben wurden. Mit *integrateHEsIntoDB* können schließlich auf einem Migrationszielserver Änderungs- und Ausführungshistorieneinträge in die Datenbank geschrieben und der Graph und dessen Zustand entsprechend modifiziert werden.

Die statischen Methoden *createPICHEFromRequest* und *createRequestFromPICHE* dienen dazu, aus dem Request-Objekt einer Änderungsoperation einen Änderungshistorieneintrag zu erzeugen und umgekehrt.

getActNameForNode ist eine Hilfsfunktion für Demonstrationszwecke, mit der anstelle der Knoten-ID der zu einem Knoten passende Aktivitätenname ausgegeben werden kann.

B.7 ChangeCoordinator

Die Klasse ChangeCoordinator enthält die Methoden, die im Rahmen des Protokolls zur Durchführung von dynamischen Änderungen benötigt werden. Jede Methode entspricht dabei einem Teilauftrag. Zu Beginn werden die übergebenen Parameter ausgewertet, dann die eigentlichen Aktionen durchgeführt und schließlich, wenn das Protokoll noch nicht beendet ist, mit diesem fortgefahren, indem die berechneten Ergebnisse als neuer Teilauftrag an den betreffenden Server geschickt werden. Das Protokoll zu dynamischen Änderungen ist in Abschnitt 5.5.5 vollständig beschrieben.

Darüber hinaus ist in ChangeCoordinator die Methode *doDynamicModification* implementiert. Hierüber wird die eigentliche Durchführung einer dynamischen Änderung veranlaßt.

C Wesentliche Veränderungen an bereits vorhandenen Klassen

Im bestehenden System mußten an einigen Stellen Aufrufe an die Verteilungsschicht eingefügt werden. Daneben waren noch an manchen Stellen Ergänzungen notwendig, vor allem durch die Berücksichtigung derjenigen Attribute von Objekten einer Prozeßinstanz, die bei der verteilten Ausführung eine Rolle spielen. Bisher konnte beispielsweise der Ausführungsserver eines Knotens vernachlässigt werden, da keine Migrationen möglich waren.

C.1 WFMS.Server.Application.ADEPTServer

Die Server-Hauptschleife wurde so erweitert, daß auch Server-Server-Kommunikation möglich wurde. Durch den orthogonalen OP-Code-Bereich wird die Kommunikation von der Client-Server-Aufrufen unterschieden. Server-Server-Requests werden zur Bearbeitung an den DistributionManager übergeben. An den folgenden weiteren Stellen wurden wesentliche, die Ausführung betreffende, Änderungen vorgenommen:

- Nachdem ein Schritt mittels workitemTerminate beendet wurde und der Zustand neu berechnet wurde, muß mit checkForMigrations überprüft werden, ob Migrationen durchzuführen sind.
- In processClientRequest wurden Sperren eingebaut, die das Starten und Beenden von Schritten kapseln.
- Bevor eine dynamische Änderung begonnen wird, wird in einer ersten Phase Zustandsinformation von anderen aktiven Servern eingeholt.
- Die endgültige Änderungsdurchführung findet jetzt über die Methode *doDynamicModification* des ChangeCoordinator statt und nicht mehr im ADEPTServer selbst.
- Es wurden zusätzliche Kommandozeilenparameter in Bezug auf die Verteilung und Möglichkeiten zum Aufruf von Administratorbefehlen an einen laufenden Server über die Konsole eingeführt. Dazu siehe auch die Anleitung zur Handhabung in Abschnitt E.1.

C.2 WFMS.Server.Kernel.SupportLayer.basics.processinst.ProcessInstance

Beim dynamischen Einfügen bekommen die Knoten n1, n2 und der neue Knoten Server zugeordnet. Des weiteren wurde der Parameter erweitert, um Knoten mit bereits gegebener ID erstellen zu können.

C.3 WFMS.Server.Kernel.SupportLayer.basics.processinst.ProcInstInterface

Beim dynamischen Einfügen wurde der Parameter erweitert, um Knoten mit bereits gegebener ID erstellen zu können.

C.4 WFMS.Server.Kernel.SupportLayer.basics.processinst.PIResult

Die Werte n2Id und newNodeId wurden dazu gefügt, da sie später benötigt werden, um Knoten mit gegebener ID zu erstellen. Außerdem wurde die Fehlermeldung um die Möglichkeit für Verteilungsfehler, wenn z.B. ein anderer Server nicht erreichbar ist, erweitert.

C.5 WFMS.Server.Kernel.DBAccess.PInstance.Instance

Beim Erzeugen einer Prozeßinstanz mit *build* wird noch die Methode *replaceTempIDsInNodeExServer* des ServerResolver aufgerufen, die temporäre IDs in den Serverzuordnungsausdrücken ersetzt.

C.6 WFMS.Server.Kernel.SupportLayer.PIService

Die umfangreichsten Änderungen waren in PIService nötig. Dies sind im wesentlichen:

- Beim Schreiben der Ausführungshistorie in *changeNodeStatus* werden, wenn eine Aktivität beendet wurde, deren Loop- und Decisionparameter in die Historie geschrieben. Diese Werte

werden später auf einem Migrationszielserver zum Einspielen der Historie unter Vermeidung zusätzlicher Zugriffe auf Datenelemente benötigt.

- Beim Aktivieren des Startknotens wird das Attribut NODE_EXECUTIONSERVER des Startknotens auf den momentanen Server gesetzt.
- Beim Aktivieren eines Knotens mit variablem Serverzuordnungsausdruck wird an das Attribut NODE_EXECUTIONSERVER der tatsächliche Server angehängt. Beim Rücksetzen des Status durch eine neue Schleifeniteration wird der Wert wieder entfernt.
- In *markoutedges* und *changeBranchState*, welche durch *calcState* aufgerufen werden, wird mit *ServerResolver.evaluateExpression* der Server des Zielknotens ausgewertet. Der dabei berechnete Migrationsdeskriptor wird von *calcState* an den Aufrufenden weitergegeben. Im Anschluß können so die erforderlichen Migrationen ausgelöst werden.

C.7 WFMS.Server.Kernel.DBAccess.PInstance.InstWL

Es werden nur noch Aktivitäten des eigenen Servers zur Aufnahme in die Arbeitsliste berücksichtigt

D Aufbau der WfMS-Architektur

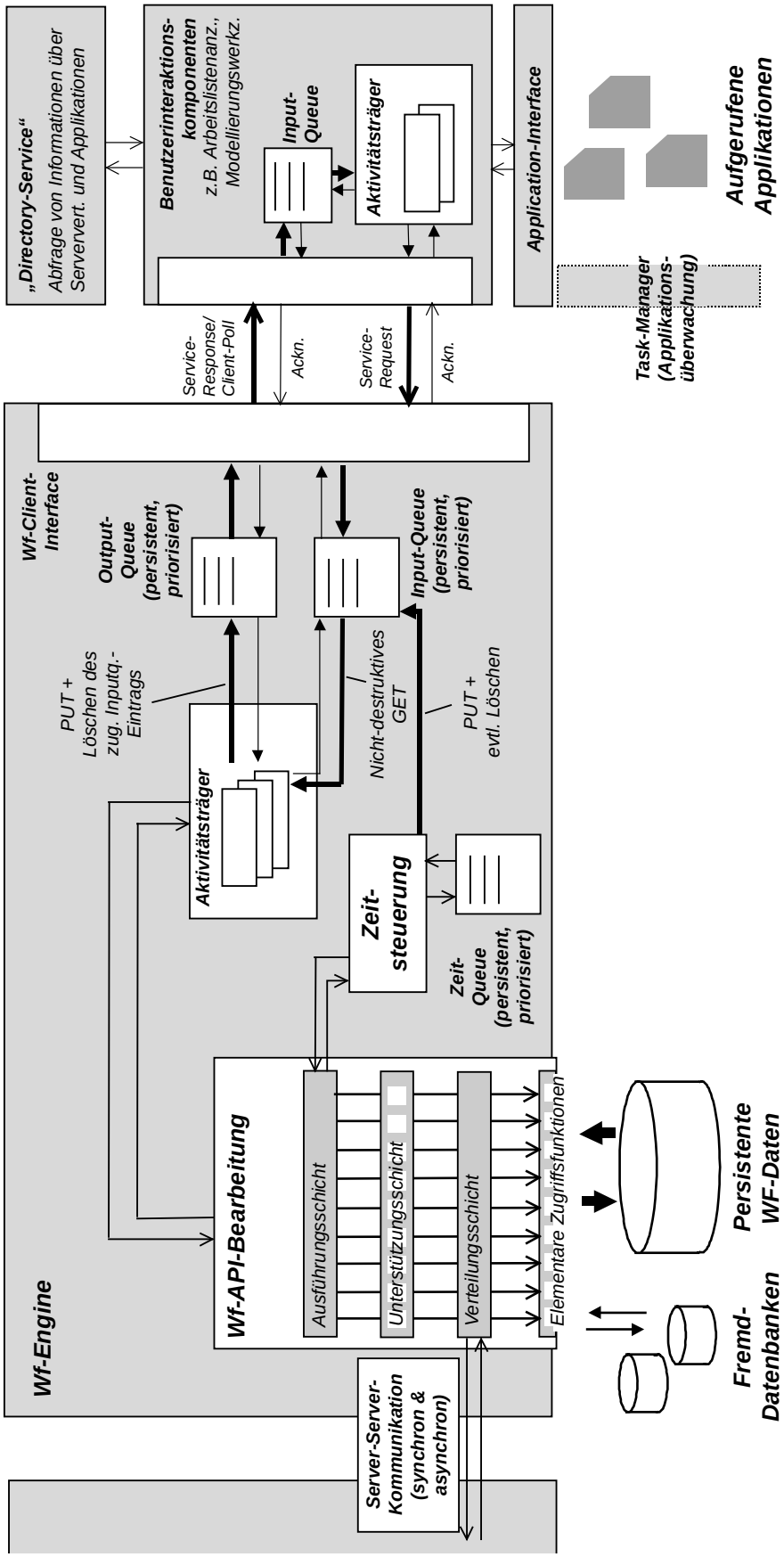


Abbildung D-1: WfMS-Architektur (entnommen aus [HSB99]).

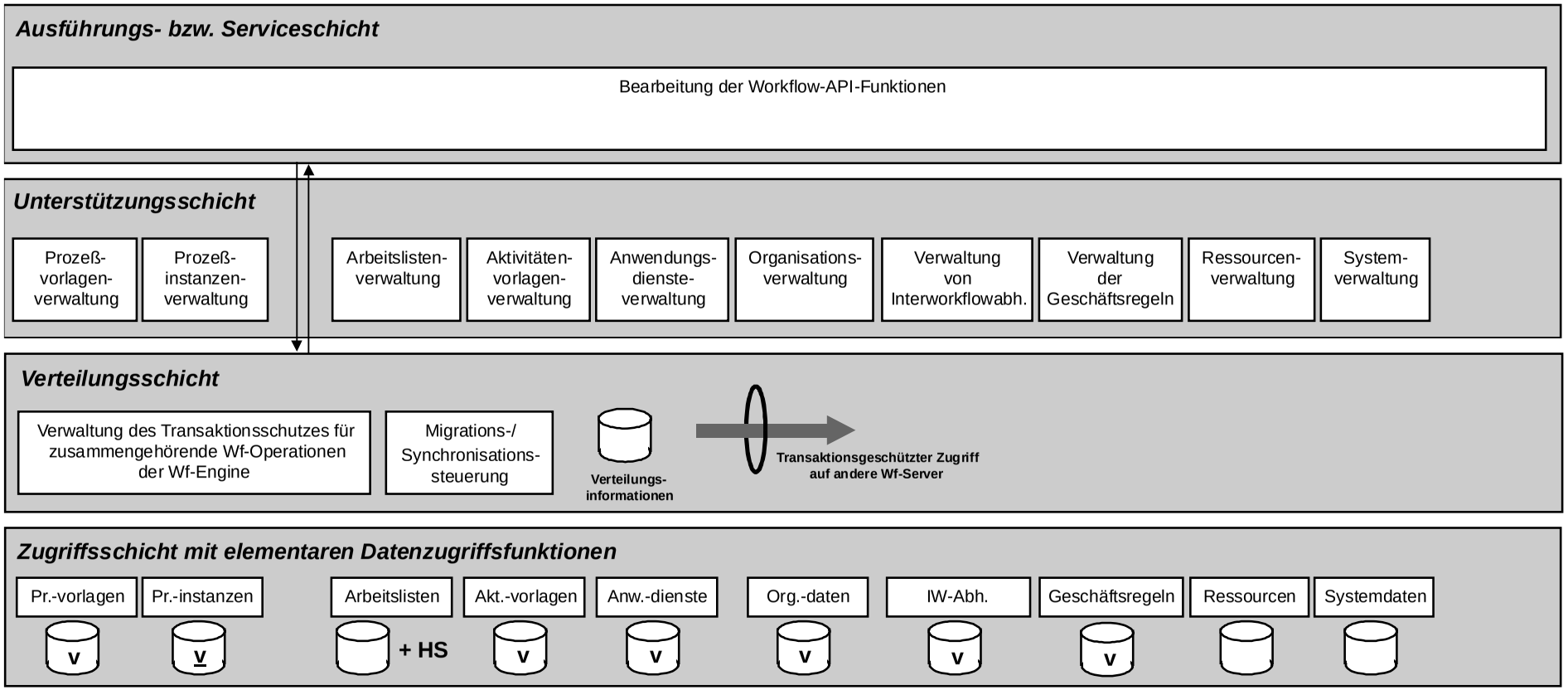


Abbildung D-2: Bearbeitung der WF-API-Funktionen (v steht für „Verteilte Speicherung“, aus: [HSB99]).

E Bedienung des Systems

Um einen schnellen Start eines verteilten ADEPT-Systems, beispielsweise für Demonstrationszwecke, zu ermöglichen, wird hier eine knappe Anleitung gegeben. Sie dient als Referenz zu allen im Rahmen der Verteilung vorgenommenen Erweiterungen.

Vorausgesetzt wird, daß auf dem ausgewählten Rechner Java 1.2 bereits installiert ist und alle Klassendateien zu ADEPT und die zusätzlich benötigten Klassen vorhanden sind. Die Batchdateien sind so eingerichtet, daß sich ADEPT-Server und Klienten starten lassen. Es besteht eine IP-Verbindung zum Rechner, auf dem die verwendete Datenbank läuft.

E.1 Konfiguration

Zunächst müssen die beteiligten Server in der Datei `ADEPT.INI` eingetragen werden. Ein Beispiel dafür kann Abbildung E-1 entnommen werden. Im Prototyp sind derzeit bis zu fünfzig Server mit beliebigen Bezeichnern möglich.

Ein Eintrag hat das Format „Sn = ConnectString, DB-Username, Adresse, Bezeichner“, wobei $n \in 1..50$. Der ConnectString gibt an, welche Datenbank ein Server benutzt. Er besteht aus JDBC-Treibername, IP-Adresse, Port und DB-Name. DB-Username ist der Benutzername für die Datenbank (das zugehörige Paßwort wird nicht in der Datei gespeichert). Wenn sich ein Server in der lokalen Domain befindet, kann als Adresse sein DNS-Hostname angegeben werden. Für Server außerhalb der lokalen Domain muß die komplette IP-Adresse numerisch angegeben werden (nicht die DNS-Adresse). Der Bezeichner ist ein beliebiger Ausdruck, darf jedoch keine „(“ und „)“ enthalten. Über ihn wird im WfMS ein Server referenziert.

Die Datei `ADEPT.INI` muß sich im Java-Klassenverzeichnis befinden, und zwar im Unterverzeichnis `WFMS\Server\Kernel\DBAccess.`

```
S1=jdbc:oracle:thin:@134.60.171.80:1521:orcl, wfsys, dbis-p46, S1
S2=jdbc:oracle:thin:@134.60.171.80:1521:orcl, wfsys, dbis-p48, S2
S3=jdbc:oracle:thin:@duke.informatik.uni-ulm.de:1521:wfsys, wfsys, dbis-p47, S3
```

Abbildung E-1: Beispielhafte Datei ADEPT.INI.

Wenn beim Starten des ADEPT-Servers kein Kommandozeilenparameter angegeben wird, arbeitet dieser per Default als Server im Einzelbetrieb. Mit dem Parameter „-server Bezeichner“ übernimmt er in einem verteilten System die Rolle des Servers, der mit dem Bezeichner angegeben ist und der natürlich in der Datei eingetragen sein muß. Es wird eine Konsistenzprüfung mit der Datei `ADEPT.INI` durchgeführt. Sollte die der Rolle des Servers entsprechende IP-Adresse, welche in der Datei angegeben ist, nicht mit der tatsächlichen IP-Adresse des Servers übereinstimmen, wird das Starten abgebrochen. Während noch WF-Server laufen, dürfen keine Änderungen an der Datei vorgenommen werden. Mit dem Parameter „-?“ lassen sich alle möglichen Kommandozeilenparameter ausgeben.

Am einfachsten lassen sich ADEPT-Server starten, wenn für jeden Server eine eigene Batch-Datei angelegt wird, in der der Java-Aufruf getätigt wird, nachdem zuvor die Systemumgebung (z.B. Pfade) je nach Betriebssystem passend für Java eingerichtet wurde. In der Batch-Datei müssen auch die Klassenpfade zu den von ADEPT benötigten Klassen angegeben sein.

Erstmalige Inbetriebnahme eines Servers

Die Datenbank, welche unter dem ConnectString angegeben wird, muß natürlich vorhanden sein. Der angegebene Benutzer muß für die Datenbank existieren und das vereinbarte Paßwort besitzen. In einer neu eingerichteten, leeren Datenbank müssen per SQL-Skript zunächst auch alle für ADEPT benötigten Tabellen und ein Sequencer erstellt werden.

Im Anschluß kann die Datenbank mit Template-Information gefüllt werden. Dazu ist nach dem Start des ADEPT-Servers vom WF-Administrator im Server-Konsolenfenster die Taste „s“ für synchronize zu drücken und mit RETURN zu bestätigen. Hierauf werden alle in der Datenbank des aktuellen Servers noch nicht vorhandenen Prozeßvorlagen aus der Datenbank des ADEPT-Servers S1, der zu diesem Zeitpunkt laufen muß, kopiert. Die gesamte Template-Information kann auf allen Servern mit Ausnahme von S1 durch Drücken der Taste „r“ plus RETURN gelöscht werden.

In zukünftigen Versionen können auch neue Vorlagen ergänzt werden, so daß das Löschen aller vorhandenen Vorlagen vor dem Einspielen der neuen Vorlagen entfallen kann. Es kann auch ein noch aufwendigerer Replikationsmechanismus eingesetzt werden.

Variable Serverzuordnungen

Wenn variable Serverzuordnungen benutzt werden, muß darauf geachtet werden, daß sich im Organisationsmodell Daten zu dem Mitarbeiter befinden, der einen Arbeitsschritt ausgeführt hat. Es ist nämlich möglich, daß als Ausführungsserver einer Aktivität A der Server im Domain desjenigen Mitarbeiters, der den im Serverzuordnungsausdruck des Knotens angegebenen Schritt ausgeführt hat, verwendet wird. Die Information, welcher Server dies ist, wird aus der Ausführungshistorie (User-ID des Bearbeiters der letzten Iteration von A) in Verbindung mit der Tabelle OM_MITARBEITER bezogen. Welche Serverzuordnungsausdrücke verwendet werden können, ist Abschnitt E.2 zu entnehmen.

Ein SQL-Statement zum Erzeugen eines beispielhaften Datensatzes, wie er für Demonstrationszwecke verwendet werden kann, ist in Abbildung E-2 gegeben. Mögliche Werte für M_DOMAINID sind derzeit 1 bis 50, was den Servern S1 bis S50 entspricht. Beim Starten eines Klienten ist darauf zu achten, daß nur solche Benutzernamen eingegeben werden, zu denen auch Einträge in der Datenbank existieren.

```
INSERT INTO OM_MITARBEITER (M_ID, M_VORNAME, M_NAME, M_DOMAINID)
VALUES ('1234575001', 'Jochen', 'zeitler1', '1')
```

Abbildung E-2: SQL-Statement zum Anlegen eines Mitarbeiters für Demonstrationszwecke.

E.2 Modellierung der WF-Graphen

Die Modellierung der Graphen kann wie gewohnt mit dem Programm WFEdit2 durchgeführt werden. Wird eine verteilte Ausführung gewünscht, so muß im Feld „Ausführungsserver“ der gewünschte Server angegeben werden. Mögliche Werte sind die, die in der Datei `ADEPT.INI` als Bezeichner festgelegt sind. Wenn kein bzw. ein ungültiger Wert angegeben wird, dann geht die Ausführungsschicht davon aus, daß der Startserver der Instanz gemeint ist.

Mögliche Variable Serverzuordnungsausdrücke sind „S(NIABC)“ für eine Referenz auf den Server des Knotens mit der beispielhaften ID NIABC und „D(P(NIABC))“ für den Server im Domain des Benutzers, der die Aktivität NIABC zuletzt ausgeführt hat.

Vom Modellierer ist darauf zu achten, daß er einen gültigen Serverzuordnungsausdruck eingibt. Dies bedeutet insbesondere, daß der referenzierte Knoten existiert und daß er ein Vorgänger des Knotens ist, für den der Ausdruck eingegeben wird. Vorgänger bedeutet ein Vorgänger über Kontroll- oder Sync-Kanten, der garantiert ausgeführt wird. Unzulässig sind beispielsweise Vorgänger über Sync-Kanten, die in einem Teilzweig einer bedingten Verzweigung liegen (wie in Abbildung 3-5 die Aktivität Q als Vorgänger der Aktivitäten F oder G).

Nachdem eine WF-Vorlage auf dem Server S1 hinzugefügt oder verändert wurde, muß sie noch zu den anderen Servern ausgebreitet werden, falls eine verteilte Ausführung möglich sein soll. Hierfür muß auf jedem anderen Server im Server-Konsolenfenster die Taste „r“ plus RETURN zum Löschen der Vorlagen und anschließend die Taste „s“ plus RETURN zum Kopieren der Vorlagen gedrückt werden.

E.3 Ausführung von WF-Instanzen

Bei der Ausführung von WF-Instanzen ergeben sich keine für den Benutzer unmittelbar sichtbaren Änderungen zur nicht verteilten Ausführung. Es ist lediglich möglich, daß bei manchen Operationen die Antwort länger als gewohnt dauert, wenn zuvor der Server noch umfangreiche Informationen von anderen Servern einholen mußte.

Für Demonstrationszwecke kann der Server, auf dem eine Aktivität ausgeführt wurde bzw. wird, angezeigt werden. Der Zustand des Graphen, der angezeigt wird, entspricht dem aktuell bekannten Zustand auf dem Server, bei dem der Client angemeldet ist. Somit können auch Schritte als ACTIVATED angezeigt werden, die auf einem anderen Server zur Ausführung bereit stehen. Diese Schritte befinden sich nicht in der Arbeitsliste. In der Arbeitsliste sind nur die Schritte aufgenommen, die der Client ausführen kann.

Wenn eine WF-Instanz ausgeführt wird, bei der variable Serverzuordnungen verwendet werden, muß darauf geachtet werden, daß sich nur ein in der Datenbank verzeichneter Benutzer anmeldet. Ansonsten kann die User-ID nicht aufgelöst werden, wenn der Server in seinem Domain bestimmt werden soll.

E.4 Wiederaufnahme abgebrochener Migrationen

Durch Drücken der Taste „m“ plus RETURN im Konsolenfenster des ADEPTServer (Migrationsquellserver) können abgebrochene Migrationen erneut angestossen werden. Dies kann notwendig werden, wenn zum Zeitpunkt der Beendigung eines Arbeitsschrittes, nach dem migriert werden soll, der Zielserver einer ausgehenden Migration nicht am Laufen ist.

E.5 Kurzübersicht

Die Tabelle E-1 ist als Kurzreferenz für Benutzer gedacht, deren Rechner einschließlich der verwendeten Datenbank durch den WF-Administrator bereits korrekt eingerichtet wurden. Sie enthält eine Auswahl wichtiger Funktionalität, chronologisch geordnet nach dem jeweils relevanten Zeitpunkt.

Aufgabe	**Vorgehen**
Einrichtung eines Rechners für den ADEPT-Server	siehe E.1 Konfiguration
Start eines ADEPT-Servers	Über die dem Server zugeordnete Batch-Datei.
Manuelles Verändern der Serverzuordnung von WF-Vorlagen	siehe E.2 Modellierung der WF-Graphen
Wiederholen abgebrochener Migrationen	Drücken der Taste „m“ plus RETURN im Konsolenfenster des ADEPTServer (Migrationsquellserver)
Beenden des Servers	Drücken der Taste „q“ plus RETURN im Konsolenfenster des ADEPTServer

Tabelle E-1: Kurzreferenz zu ADEPT$_{distribution}$.

Literaturverzeichnis

[AGK+96] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. El Abbadi und C. Mohan: *Exotica/FMDC: A Workflow Management System for Mobile and Disconnected Clients*. Distributed and Parallel Databases, 4(3):229-247, Juli 1996.

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör und C. Mohan: *Failure Handling in Large Scale Workflow Management Systems*. Technical Report RJ9913, IBM Almaden Research Center, 1994.

[AMG95] G. Alonso, C. Mohan und R. Günthör: *Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management*. In: *Proceedings of the IFIP Working Conference on Information Systems for Decentralized Organizations*, 1995.

[BD97] T. Bauer und P. Dadam: *A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration*. In: *Second IFCIS Conference on Cooperative Information Systems*, S. 99-108, Kiawah Island, South Carolina, Juni 1997.

[BD98a] T. Bauer und P. Dadam: *Architekturen für skalierbare Workflow-Management-Systeme – Klassifikation und Analyse*. Technischer Bericht 98-02, Universität Ulm, Fakultät für Informatik, Januar 1998.

[BD98b] T. Bauer und P. Dadam: *Variable Migration von Workflows in ADEPT*. Technischer Bericht 98-09, Universität Ulm, Fakultät für Informatik, September 1998.

[BD99a] T. Bauer und P. Dadam: *Efficient Distributed Control of Enterprise-Wide and Cross-Enterprise Workflows*. In: *Proc. Workshop Enterprise-Wide and Cross-Enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI (Informatik '99)*, S. 25-32, Paderborn, 1999.

[BD99b] T. Bauer und P. Dadam: *Verteilungsmodelle für Workflow-Management-Systeme - Klassifikation und Simulation.* Informatik Forschung und Entwicklung, 14(4), Dezember 1999.

[BD00] T. Bauer und P. Dadam: *Variable Serverzuordnungen und komplexe Bearbeiterzuordnungen im Workflow-Management-System ADEPT*. Technischer Bericht, Universität Ulm, Fakultät für Informatik, 2000 (erscheint demnächst).

[BMR94] D. Barbarà, S. Mehrotra und M. Rusinkiewicz: *INCAS: A Computational Model for Dynamic Workflows in Autonomous Distributed Environments*. Technischer Bericht, Matsushita Information Technology Laboratory, Princeton, New Jersey, Mai 1994.

[CDK95] G. Coulouris, J. Dollimore und T. Kindberg: *Distributed Systems: Concepts and Design*. Addison-Wesley, 2. Auflage, 1995.

[CGP+96] F. Casati, P. Grefen, B. Pernici, G. Pozzi und G. Sánchez: *WIDE: Workflow model and Architecture*. Technischer Bericht CTIT 96-19, University of Twente, 1996.

[Dad96] P. Dadam: *Verteilte Datenbanken und Client/Server-Systeme*. Springer-Verlag, 1996.

[DKM+97] S. Das, K. Kochut, J. Miller, A. Sheth und D. Worah: *ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR$_2$*. Technischer Bericht #UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, Februar 1997.

[Gri97] M. Grimm: *ADEPT-TIME: Temporale Aspekte in flexiblen Workflow-Management-Systemen*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, 1997.

[Hen97] C. Hensinger: *ADEPT$_{flex}$ - Dynamische Modifikation von Workflows und Ausnahmebehandlung in WfMS*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, 1997.

[Hol98] O. Holzwarth: *Middlewaretechnologien für datenbankgestützte Multimedia-Anwendungen - Analyse und Realisierung im Rahmen einer Beispielanwendung*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, 1998.

[HSB99] C. Hensinger, B. Schultheiß und T. Bauer: *ADEPT$_{WORKFLOW}$ - Architektur und Schnittstellen.* Interner Bericht, Abteilung Datenbanken und Informationssysteme, Universität Ulm, 1999.

[HS96] P. Heinl und H. Schuster: *Towards a Highly Scalable Architecture for Workflow Management Systems*. In: *Proceedings of the 7th International Workshop on Database and Expert Systems Applications, DEXA '96*, S. 439-444, Zürich, September 1996.

[JBS97] S. Jablonski, M. Böhm und W. Schulze (Hrsg.): *Workflow-Management: Entwicklung von Anwendungen und Systemen*. Dpunkt-Verlag, 1997.

[MWW+98] P. Muth, D. Wodtke, J. Weißenfels, A. Kotz-Dittrich und G. Weikum: *From Centralized Workflow Specification to Distributed Workflow Execution*. Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, 10(2):159-184, März/April 1998.

[NSS98] J. Neeb, R. Schamburger und H. Schuster: *Using Distributed Object Middleware to Implement Scalable Workflow Management Systems*. In: *Proceedings of the International Workshop on Issues and Applications of Database Technology*, Berlin, Juli 1998.

[Par98] H. Partsch: *Requirements-Engineering systematisch.* Springer Verlag, Berlin 1998.

[RBD98] M. Reichert, T. Bauer und P. Dadam: *Operational Support for Flexible and Scalable Workflow Management in ADEPT*. Interner Bericht, Abteilung Datenbanken und Informationssysteme, Universität Ulm, 1998.

[RBD99] M. Reichert, T. Bauer und P. Dadam: *Enterprise-Wide and Cross-Enterprise Workflow-Management: Challenges and Research Issues for Adaptive Workflows*. In: *Proc. Workshop Enterprise-Wide and Cross-Enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI (Informatik '99)*, S. 56-64, Paderborn, 1999.

[RD98] M. Reichert und P. Dadam: *ADEPT$_{flex}$ - Supporting Dynamic Changes of Workflows Without Losing Control*. Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, 10(2):93-129, März/April 1998.

[Rei00] M. Reichert: *Dynamische Ablaufänderungen in prozeßorientierten Workflow-Management-Systemen*. Dissertation, Universität Ulm, Fakultät für Informatik , 2000 (erscheint demnächst).

[RHD98] M. Reichert, C. Hensinger und P. Dadam: *Supporting Adaptive Workflows in Advanced Application Environments.* In: *Proc. EDBT Workshop on Workflow Management Systems*, S. 100-109, Valencia, März 1998.

[SM96] A. Schill und C. Mittasch: *Workflow Management Systems on Top of OSF DCE and OMG CORBA*. Distributed Systems Engineering, 3(4):250-262, Dezember 1996.

[SNS99] H. Schuster, J. Neeb und R. Schamburger: *A Configuration Management Approach for Lange Workflow Management Systems*. In: *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration*, San Francisco, California, Februar 1999.

[SS97] G. Semeczko und S.Y.W. Su: *Supporting Object Migration in Distributed Systems*. In: *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, Melbourne, Australia, April 1997.

[Tra97] S. Traub: *Verteilte PC-Betriebssysteme*. B.G. Teubner, 1997.

[Wei97] P. Weilbach: *Implementierungsaspekte zur Verwaltung und Synchronisation dynamischer Änderungen in prozeßorientierten Workflow-Management-Systemen.* Diplomarbeit, Universität Ulm, Fakultät für Informatik, 1997.

[WK96] J. Wäsch und W. Klas: *History Merging as a Mechanism for Concurrency Control in Cooperative Environments.* In: *Proc. RIDE-NDS ´96*, S. 76-85, New Orleans, Louisiana, Februar 1996.

[WMC99] Workflow Management Coalition: *Terminology & Glossary.* WFMC-TC1011, Version 3.0, Brüssel, Februar 1999.

Erklärung

Name: Jochen Zeitler

Mat.-Nr.: 331 306

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

(Jochen Zeitler)