Flexible Fehlerbehandlung für mobile Ad-hoc-Prozesse*

Klaus Haller[1] Heiko Schuldt[1,2] Can Türker[1]

[1] Eidgenössische Technische Hochschule Zürich
Institut für Informationssysteme
CH-8092 Zürich, Switzerland

[2] Private Universität für Medizinische Informatik und Technik Tirol (UMIT)
Abteilung Information and Software Engineering
A-6020 Innsbruck, Österreich

{haller,schuldt,tuerker}@inf.ethz.ch

Abstract: Durch die inhärente Mobilität und Dynamik moderner Dienstanbieter und -nutzer werden Informationssysteme zunehmend dezentral, auf dem Peer-to-Peer-Paradigma basierend, organisiert. Gleichzeitig werden die Benutzeranforderungen immer individueller. Daher verwenden wir Ad-hoc-Prozesse, die Dienste verschiedener Anbieter bündeln, um damit individuelle Benutzerbedürfnisse zu erfüllen. Solche Ad-hoc-Prozesse zeichnen sich dadurch aus, dass sie – im Gegensatz zu Geschäftsprozessen mit hoher Wiederholfrequenz – jeweils vom Benutzer individuell zusammengestellt beziehungsweise konfiguriert werden und daher nur einmal oder wenige Male zur Ausführung kommen. Mobile Agenten werden dabei zur Prozessausführung eingesetzt, um auch ein temporäres Entkoppeln der Benutzer während der Ausführung ihrer Prozesse zu unterstützen. In diesem Beitrag bieten wir einen Überblick über unsere Ausführungsinfrastruktur und zeigen mögliche Fehlerfälle auf, die bei der Ausführung von Prozessen durch mobile Agenten zu beachten sind. Ferner diskutieren wir, wie Recovery bzw. flexible Fehlerbehandlung und Mobilität von Anwendungen zusammengeführt werden können.

1 Einleitung

Mit der stetig zunehmenden Mobilität von Anwendern (Dienstnutzern) und Datenquellen (Dienstanbietern) und dem zunehmenden Einsatz mobiler Sensoren zeichnet sich ein Wandel in der Informationsverarbeitung ab. Anwendungsentwicklung erfolgt durch das Zusammensetzen bestehender Dienste zu *Prozessen* (Workflows), die in einer verteilten Umgebung angeboten werden. Anwendungen beschränken sich jedoch nicht mehr ausschließlich auf vordefinierte Prozessmuster. Sie bestehen vielmehr aus individuellen, benutzerspezifischen Kombinationen von Dienstaufrufen, die häufig sogar ad hoc zusammengestellt werden. Als Folge dieser Individualisierung werden solche Anwendungsprozesse nur einmal (oder einige wenige Male) instanziert. Wir sprechen hier daher von *Ad-hoc-Prozessen*. Solche finden sich in verschiedenen Bereichen, so unter anderem auch im Gesundheitswesen (beispielsweise in Krankenhausinformationssystemen [Rei00]). Sie

*Teile der Arbeit wurden durch den Schweizer Nationalfond im Rahmen des Projektes MAGIC gefördert.

zeichnen sich dadurch aus, dass es zwar standardisierte Prozessmuster gibt, trotzdem aber die Möglichkeit, diese Muster individuell anzupassen, entscheidend ist. Dabei kann die Systemkonfiguration einschließlich der Dienstanbietern (oft) als statisch angesehen werden. Mit der zunehmenden Verknüpfung von Informationssystemen muss aber immer öfter auch die Dynamik der zugrunde liegenden Infrastruktur berücksichtigt werden. Neue Dienstanbieter können hinzukommen, existierende Anbieter weitere Dienste anbieten beziehungsweise existierende modifizieren. Solche Veränderungen erfordern eine flexible Prozessausführung, welche die aktuell verfügbaren Dienste berücksichtigt.

Traditionelle Infrastrukturen zur Prozessausführung bestehen aus einer zentralen Workflow-Engine. Bei dieser Engine sind die Prozessbeschreibungen hinterlegt, so dass sie die Prozessausführung komplett steuern kann, wie dies zum Beispiel bei MQSeries [MQS03] geschieht. Aufgrund der Verteilung und Heterogenität der Dienstanbieter und der Dynamik von Ad-hoc-Prozessen sind solche zentralisierten Ansätze nicht optimal. Neuere Entwicklungen hin zu verteilten Infrastrukturen zur Prozessverwaltung sind darauf ausgerichtet, standardisierte Prozesse mit hoher Wiederholfrequenz – nicht jedoch individuelle Ad-hoc Prozesse – korrekt und möglichst effizient auszuführen. In diesen Systemen werden die jeweiligen Komponenten (Peers) des Informationssystems, die Teile eines Prozesses ausführen (können), mit vordefinierten Prozessbeschreibungen und sonstigen Metadaten (optimal) konfiguriert [SSSW02]. Insbesondere müssen diese Peers die nachfolgenden Aktivitäten eines Prozesses kennen. Kommen neue Anwendungsprozesse in das System, erfolgt eine dynamische Rekonfiguration, indem die Prozessbeschreibung mit den zugehörigen Metadaten auf den betroffenen Peers verteilt bzw. repliziert wird. Der mit der Replikation verbundene Aufwand zahlt sich dann nicht mehr aus, wenn ein Prozess eingerichtet wird, der nur einmal bzw. einige wenige Male instanziert wird. Daher bietet es sich bei Ad-hoc-Prozessen an, die Prozessbeschreibung mit der Prozessinstanz durch das Netzwerk wandern zu lassen. Dienstaufrufe bzw. Prozessschritte werden dazu direkt lokal ausgeführt. Da in diesem Fall die jeweiligen Peers nicht vorkonfiguriert werden können (da a priori noch nicht bekannt ist, wie ein Ad-hoc-Prozess spezifiziert ist), müssen alle für den Dienstaufruf benötigten Informationen zusammen mit der Prozessbeschreibung zum Dienstanbieter geschickt werden. Die Infrastruktur muss lediglich die Funktionalität für die verlässliche Ausführung von Ad-hoc-Prozessen bereitstellen. Dazu gehören Basisroutinen zur Koordination von Ad-hoc-Prozessen ebenso wie spezielle Fehlerbehandlungsmechanismen. Diese Mechanismen müssen berücksichtigen, dass sich Dienstanbieter temporär bewusst abmelden oder unbewusst ausfallen können.

In diesem Papier skizzieren wir die Architektur einer Infrastruktur, wie sie im AMOR-Projekt (Agents, MObility, tRansactions) für die Ausführung und Fehlerbehandlung von Ad-hoc-Prozessen eingesetzt wird. Technisch gesehen werden Ad-hoc-Prozesse als mobile Agenten realisiert, die von Peer zu Peer migrieren, um einen Ad-hoc-Prozess auszuführen. Die AMOR-Infrastruktur unterstützt die Intra-Prozessparallelität durch das Klonen von Agenten. Damit können mehrere Klone gleichzeitig auf verschiedenen Peers Dienste in Anspruch nehmen und somit den Gesamtprozess parallel und unabhängig vorantreiben. Alle Agenten werden hierzu mit einer zusätzlichen Funktionalität ausgestattet. Diese ermöglicht es den Agenten, eine korrekte Synchronisation ohne einen zentralen Koordinator durchzuführen. Mit anderen Worten realisieren wir eine *vollständig dezentrali-*

sierte Prozessverwaltung, welche trotzdem eine korrekte Ausführung sicher stellt, indem sie transaktionale Ausführungsgarantien gewährleistet.

Dieser Beitrag ist wie folgt gegliedert: Kapitel 2 skizziert die Spezifikation von Ad-hoc-Prozessen und mögliche Fehler bei der Ausführung solcher Prozesse. Kapitel 3 erläutert die Ausführung von Ad-hoc-Prozessen einschließlich der benötigten Infrastruktur. Die Fehlerbehandlung wird separat in Kapitel 4 behandelt. Verwandte Arbeiten werden in Kapitel 5 diskutiert. Kapitel 6 beschließt dieses Papier mit einer Zusammenfassung und geht auf aktuelle und geplante Erweiterungen der präsentierten Infrastruktur ein.

2 Ad-hoc-Prozesse

2.1 Spezifikation

Ad-hoc-Prozesse bestehen aus einer Menge von Schritten. Wie Abbildung 1 zeigt, gibt es verschiedene Arten von Schritten: Aktivitäten, Verzweigungen und Synchronisationsschritte. Auf diesen Schritten ist eine partielle Ordnung definiert, das heißt verschiedene Schritte bzw. Prozesspfade können als parallel ausführbar spezifiziert werden, was durch Parallelitätsschritte explizit modelliert wird. Dies setzt natürlich voraus, dass kein Informationsfluss zwischen den Pfaden stattfindet.

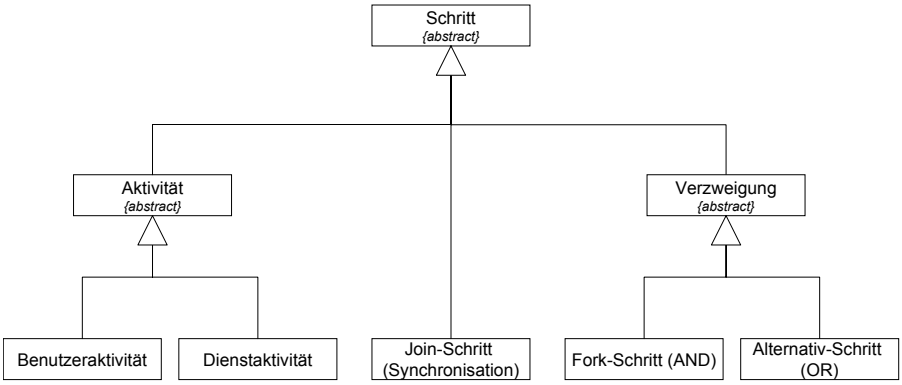

Abbildung 1: Arten von Prozessschritten

Abbildung 2 enthält einen einfachen Beispielprozess, welcher mittels weiteren Aktivitäten zu einem Ad-hoc-Prozess erweitert werden kann. Die Prozessdefinition legt fest, wie auf einen ankommenden Notfallpatienten im Krankenhaus reagiert wird. Im ersten Schritt werden die Patientendaten aufgenommen. Für solche Benutzerinteraktionen steht der Typ *Benutzeraktivität* zur Verfügung. Aktivitäten dienen dazu, den System- oder Prozesszustand zu verändern. Die zweite Aktivitätsart stellen die *Dienstaktivitäten* dar. Diese bieten Anwendungsdienste an, beispielsweise das Speichern der eingegebenen Patientendaten im Krankenhausinformationssystem. Neben solchen einfachen Diensten werden auch komplexere unterstützt, die mehrere Kommunikationsschritte/-phasen enthalten.

Ein spezieller Dienst muss aber nicht der Einzige seiner Art im System sein. Es können

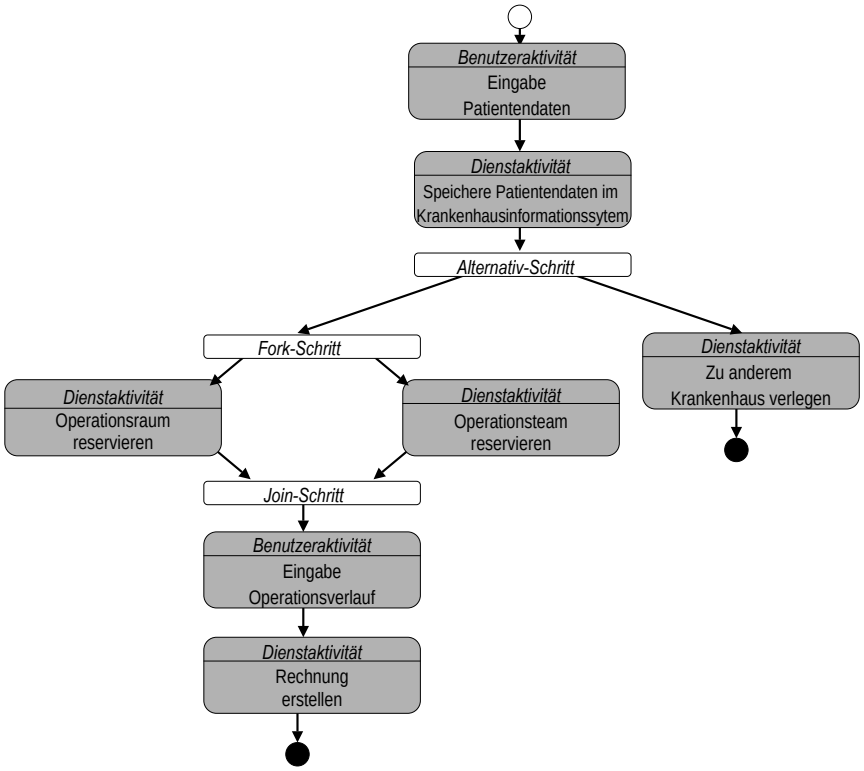

Abbildung 2: Spezifikation eines Ad-hoc-Prozesses

auch weitere, semantisch äquivalente Dienste existieren (von der Ontologienproblematik wollen wir hier abstrahieren). So bieten zum Beispiel alle Krankenhäuser einen äquivalenten Dienst 'Operationsteam reservieren' an. Da verschiedene Krankenhäuser unterschiedliche Spezialisten haben, kann das Ergebnis bei diesen Dienstaufrufen auch unterschiedlich sein. Daher kann in der Prozessspezifikation angegeben werden, ob und gegebenenfalls wie viele verschiedene Dienste gleichzeitig (durch verschiedene Agentenklone) kontaktiert werden sollen, um die Anwendungsbedürfnisse optimal zu erfüllen. Die dadurch entstehende Parallelität bezeichnen wir als *Dienstparallelität*.

Eine zweite Art von Parallelität ist die *Prozessparallelität*, die durch *Fork-* und *Join-Schritte* spezifiziert wird. Nur Fork-Schritte dürfen mehrere ausgehende, parallel auszuführende Aktionen haben. Sie ermöglichen es in unserem Beispiel, parallel nach einem Operationsraum und einem Operationsteam zu suchen. Bei der Prozessparallelität müssen alle Pfade erfolgreich ausgeführt werden, während bei Dienstparallelität nur mindestens einer der Pfade erfolgreich sein muss.

Verzweigungspunkte erlauben es, alternative Ausführungspfade zu spezifizieren. Diese können verwendet werden, falls die Ausführung eines Pfades fehlschlägt. Falls in unserem Beispiel der Notfallpatient mangels Verfügbarkeit von Team und Räumlichkeiten nicht

behandelt werden kann, wird er in ein anderes Krankenhaus verwiesen. Die Reihenfolge, in welcher diese Varianten versucht werden, kann dabei statisch definiert sein oder aber der Prozess entscheidet bei der Ausführung des Verzweigungspunktes aufgrund des Prozesszustandes, welche Variante er wählt.

2.2 Fehlerfälle

Bei der Ausführung von Prozessinstanzen können zur Laufzeit verschiedene Fehler auftreten. Im Folgenden klassifizieren wir die Fehler danach, ob sie durch das Laufzeitmodell des AMOR-Systems abgefangen werden können/sollen oder ob sie Wissen über die Anwendungssemantik erfordern und daher auf der Ebene der Prozessspezifikation zu behandeln sind. Einen ersten Überblick hierzu liefert Tabelle 1.

Fehlerart	**Behandlung**
Dienstfehler	... durch die Prozessspezifikation
Providerfehler	
Ausführungsmodellbedingte Fehler	... transparent durch das System
Isolationsverletzung	
Infrastrukturfehler	

Tabelle 1: Fehlerklassifikation und Zuständigkeiten für Fehlerbehandlung

Infrastrukturfehler sind transparent vom System zu behandeln. Beispiele für solche Fehler sind ungeplante Unterbrechungen (nicht aber geplante, da diese im Ausführungsmodell berücksichtigt werden können) in Folge von Netzwerkfehlern. Auch der Absturz eines Agenten gehört zu dieser Fehlerklasse. Daneben gibt es Fehler, die auftreten können, wenn mehrere Prozesse gleichzeitig aktiv sind. Auch wenn die einzelnen Prozesse erfolgreich ihre Aktivitäten ausführen, können sie derart interferieren, dass sie das gewählte Concurrency-Control-Korrektheitskriterium verletzen (Isolationsverletzung). In unserem System verwenden wir das Korrektheitskriterium *Konfliktserialisierbarkeit* [BHG87].[1] Eine Isolationsverletzung liegt vor, wenn der Serialisierungsgraph einen Zyklus enthält. In diesem Fall muss das System verhindern, dass die in den Zyklus involvierten Prozesse (scheinbar) erfolgreich terminieren. Hierzu signalisiert das System eine Isolationsverletzung.

Ferner gibt es Fehler, die auf das Ausführungsmodell zurückzuführen sind. Bei zentralisierten Workflow-Engines reduziert sich diese Frage darauf, wie das Recovery bei einem Absturz der Workflow-Engine durchgeführt wird. Falls man aber Workflows dezentral ausführt, so wie in unserem Fall, kann ein Ausführungspfad (ein Agent) 'verschwinden'. Hier muss das Ausführungsmodell garantieren, dass das ganze System wieder in einen konsistenten Zustand überführt wird.

Allen bisher diskutierten Fehlern ist gemeinsam, dass ihr Auftreten in keinem direkten

[1] Genauer handelt es sich um Konfliktserialisierbarkeit im Zusammenhang mit semantisch reichen Operationen [VHBS98].

Zusammenhang mit der Anwendungssemantik steht. Weder der Anwendungsentwickler bei der Programmierung noch der Anwender zur Laufzeit sollten sich daher mit solchen Fehlern auseinander setzen müssen. Außerdem muss das Ausführungsmodell sicherstellen, dass es mit diesen Fehlern umgehen kann – und wenn möglich nicht die Prozessausführung abgebrochen wird. Da das Ausführungsmodell unabhängig von der genauen Anwendungssemantik ist, kann es keine Fehler lösen, die im Zusammenhang mit der Anwendungssemantik stehen. Stattdessen muss die Prozessspezifikation festlegen, wie mit Fehlern umzugehen ist. Diese Fehler können wir in zwei Klassen einteilen: *Dienst-* und *Providerfehler*.

Beide treten nur im Zusammenhang mit der Ausführung von Dienstaktivitäten auf. Bei einem Providerfehler kann kein Anbieter für einen bestimmten Dienst gefunden werden. In unserem Krankenhausbeispiel könnte es im Zusammenhang mit der Verlegung eines Kranken sein, dass – zum Beispiel wegen einer Netzwerkpartitionierung – kein anderes Krankenhaus gefunden wird. Bei Dienstfehlern wird zwar ein Dienstanbieter gefunden, allerdings kann dabei das durch die Anwendungssemantik erforderliche Ziel nicht erreicht werden. So würde bei unserem Beispielprozess zwar ein Anbieter für den Dienst 'Operationsteam reservieren' gefunden, allerdings sind bereits alle passenden Teams ausgebucht. An diesem Beispiel wird auch deutlich, weshalb die Unterscheidung zwischen Provider- und Dienstfehlern sinnvoll ist: Im letztgenannten Fall könnten die Kriterien aufgeweicht werden ('lieber ein unerfahrenes Ärzteteam nehmen als eine dringend notwendige Operation zu verschieben oder ausfallen zu lassen'), während eine solche Aufweichung der Kriterien nicht sinnvoll ist, wenn man überhaupt keine Dienstanbieter findet.

3 Ausführung von Ad-hoc-Prozessen im AMOR-System

3.1 Systemarchitektur

Unsere Systemarchitektur (siehe Abbildung 3) basiert auf einem Peer-to-Peer-Netzwerk. Die Peers stellen die eigentlichen Dienstanbieter, wie zum Beispiel medizinische Informationssysteme oder Patientendatenbanken niedergelassener Ärzte, dar. Die Dienste der einzelnen Peers werden von Ad-hoc-Prozessen genutzt. Eine Aufgabe der Peers ist es, eine einheitliche Schnittstelle anzubieten, um von der Komplexität und Heterogenität der darunter liegenden Datenquellen zu abstrahieren. Dies geschieht mit der Dienstorientierung. Die Peers verbergen mit Hilfe von Diensten beispielsweise, ob die Abrechnung in einem Krankenhaus mittels SAP oder mit einem anderen Produkt realisiert wird.

Auf jedem Peer befindet sich eine lokale Koordinationsschicht, die unter anderem Metadaten zur aktuellen Systemkonfiguration speichert. Das Dienstrepository verwaltet Informationen über die lokal verfügbaren Dienste. Das Netzwerkrepository hingegen notiert die Verbindungen zu anderen Peers im Netzwerk. Beide Repositories zusammen erlauben es den Prozessen, Dienste sowohl lokal als auch auf anderen Peers zu finden [HS01]. Zur Beschleunigung solcher Suchvorgänge besteht die Möglichkeit, Informationen über Dienste auf anderen Peers zwischenzuspeichern und damit bei sich oft wiederholenden Anfragen einen deutlichen Geschwindigkeitsgewinn zu erzielen. In statischen Konfigurationen wäre dieser Ansatz identisch mit einem (replizierten) Yellow-Pages-Ansatz.

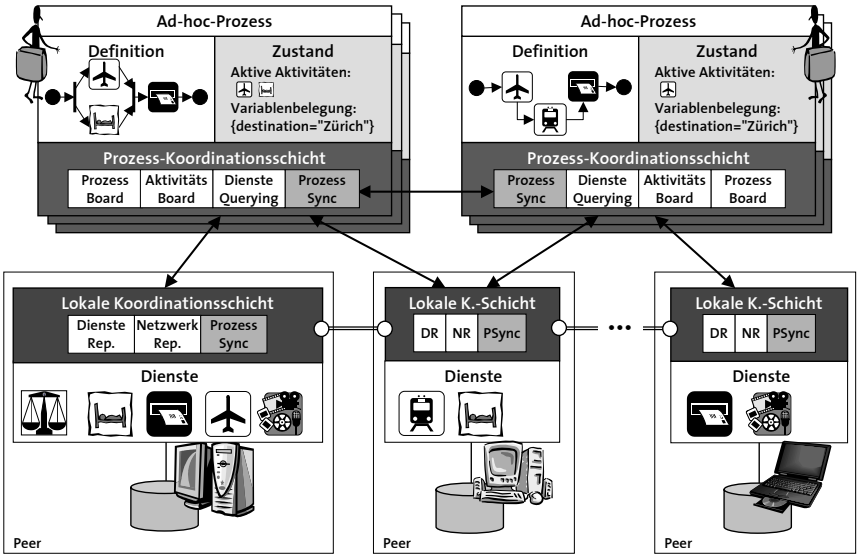

Abbildung 3: Aufbau des AMOR-Systems

Das Dienstrepository wird also für die Realisierung der Querying-Funktionalität durch die *Dienste-Querying*-Komponente der Ad-hoc-Prozessinstanzen verwendet. Diese Komponente kontaktiert die lokale Koordinationsschicht, wenn ein Dienstanbieter benötigt wird. Damit muss der Ad-hoc-Prozess über die Prozessbeschreibungen verfügen, um einen passenden Dienstanbieter suchen lassen zu können.

Weiterhin verwaltet jeder Prozess seinen aktuellen Ausführungszustand. Dies ermöglicht der Prozesskoordinationsschicht, die eigentliche Prozessausführung zu steuern. Hierzu verwendet die Koordinationsschicht noch weitere, spezialisierte Komponenten, wie beispielsweise ein *Prozessboard*. Ein Prozessboard verwaltet eine Menge an Objekten, die über ihre Namen adressiert werden. Damit realisiert das Prozessboard den Informationsfluss zwischen verschiedenen – unter Umständen auch parallelen – Schritten eines Prozesses, da jeder Schritt neue Einträge (Instanzvariablen) in das Prozessboard veranlassen bzw. lesend oder schreibend auf diese zugreifen kann. Sichtbar sind diese Prozessboardeinträge allerdings nur innerhalb einer Prozessinstanz.

Im Falle der Dienstparallelität klonen sich die mobilen Agenten. Danach können die jeweiligen Klone semantisch äquivalente Dienste verschiedener Peers aufrufen. Später synchronisieren sich diese Klone über ihre Aktivitätsboards. Der Aufbau eines Aktivitätsboards ist äquivalent zu dem eines Prozessboards. Ein Aktivitätsboard enthält eine Menge an Objekten, deren Sichtbarkeit allerdings auf die Klone genau einer Aktivität beschränkt ist. Innerhalb der Aktivitäten gleichen die verschiedenen Klone ihre Einträge an Synchronisationspunkten ab. Damit wird es zum Beispiel möglich, das bestqualifizierte Operationsteam zu reservieren. Hierzu würden im ersten Teil der Dienstaktivität die verschiedenen

Peers kontaktiert. Anschließend würden dann – sofort oder bei Netzwerkunterbrechungen verzögert – die einzelnen Klone ihre Informationen austauschen, so dass anschließend die Reservierung des am besten geeigneten Operationsteams erfolgen kann.

Sowohl die Peers als auch die Ad-hoc-Prozessinstanzen sind mit *Prozesssynchronisationskomponenten* ausgestattet. Diese realisieren die Isolationseigenschaft mittels eines verteilten Transaktionsprotokolls[2], das korrekte parallele Abläufe garantiert. Sobald eine Isolationsverletzung erkannt wird, wird den beteiligten Ad-hoc-Prozessinstanzen signalisiert, welche Schritte zur Behandlung dieses Fehlers zurückzusetzen sind.

3.2 Ausführung von Schritten

Wie bereits erklärt, können *Aktivitäten* auf das Prozessboard und auf die Dienste verschiedener Peers zugreifen. Ist in der Prozessspezifikation angegeben, dass Dienstparallelität gewünscht ist – also gleichzeitig mehrere semantisch äquivalente Dienste genutzt werden sollen – repliziert sich die Prozessinstanz. Hierzu klont sich der entsprechende Agent. Jeder dieser Klone migriert dann zu dem von ihm zu kontaktierenden Peer. Da in einem dynamischen Peer-to-Peer-Netzwerk die aktuelle Verfügbarkeit der einzelnen Peers nicht zum Programmierzeitpunkt bekannt ist, müssen die Peers (genauer die Dienste) zur Laufzeit gesucht werden. Hierzu kontaktiert die Prozessinstanz die lokale Koordinationsschicht des jeweiligen Peers, auf dem sie sich gerade befindet. Die lokale Schicht überprüft daraufhin ihr Dienstrepository nach den auf diesem Peer verfügbaren Diensten und leitet die Anfrage gegebenenfalls an andere Peers weiter.

Die bei Aktivitäten angewandte Technik, Parallelität mittels Klonen zu realisieren, wird ebenfalls bei der Ausführung von *Fork-Schritten* – also bei Prozessparallelität – angewendet. Bei der Ausführung eines solchen Fork-Schrittes wird für jeden ausgehenden Prozesspfad ein Klon erzeugt, so dass die Prozesspfade parallel und unabhängig voneinander ausgeführt werden können. Jeder Klon verfügt hierbei über eine Kopie des Aktivitäts- und des Prozessboards. Damit können sie im Falle von vorübergehenden Netzwerkunterbrechungen weiterarbeiten. Das System stellt für den Programmierer transparent sicher, dass trotzdem keine Inkonsistenzen auftreten. Falls diese Klone dann den korrespondierenden *Join-Schritt* erreichen, synchronisieren sie sich wieder und wählen genau einen Klon aus, der ihren Ausführungspfad weiter ausführen soll. Alle anderen Klone hingegen terminieren.

Ein *Benutzerinteraktionsschritt* ermöglicht die Interaktion zwischen Anwender und mobilen Agenten während der Prozessausführung. Um dem Benutzer Zwischenergebnisse der bisherigen Prozessausführung präsentieren zu können, greifen Benutzerinteraktionsschritte auf die Prozessboardeinträge zu. Daneben ist auch ein schreibender Zugriff möglich. Damit kann eine Prozessinstanz die Ergebnisse der bisherigen Prozessausführung dem Anwender mitteilen. Andererseits kann ein Benutzerinteraktionsschritt auch schreibend auf das Prozessboard zugreifen. Dies ermöglicht dem Benutzer, Daten dem Prozess mitzuteilen, unter anderem um unter verschiedenen Optionen zu wählen. Falls beispielsweise verschiedene Operationsteams zur Verfügung stehen, könnte der Anwender den operie-

[2] Genauere Informationen hierzu finden sich in [HS03].

renden Arzt seines Vertrauens auswählen. Ein weiterer Aspekt ist bei der Interaktion mit einem Benutzer zu beachten. Ein Benutzer verwendet nicht immer den gleichen Rechner, sondern verfügt beispielsweise über einen Rechner im Büro und einen PDA, mit dem er auf dem Arbeitsweg zu erreichen ist. Jeder dieser Rechner verfügt über die Information, welcher Benutzer gerade eingeloggt ist. Daher kann ein Benutzerinteraktionsschritt zum Ausführungszeitpunkt ermitteln, wo sich der Anwender befindet, der informiert werden soll beziehungsweise von dem Angaben benötigt werden. Daraufhin migriert der Agent zu dem entsprechenden Peer.

4 Fehlerbehandlung

Die Fehlerbehandlung des AMOR-Systems zeichnet aus, dass sie dezentral realisiert ist und flexibel auf die verschiedenen Fehlersituationen eingeht. In Fehlersituationen wird nur *partiell* zurückgesetzt, also nur soweit, wie es notwendig ist, um den Fehler zu beheben. Anschließend wird die Ausführung mit einem alternativen Pfad (falls vorhanden) fortgesetzt. Anders als bei klassischen Datenbanktransaktionen üblich, wird somit der Prozess nur in Ausnahmefällen vollständig zurückgesetzt.

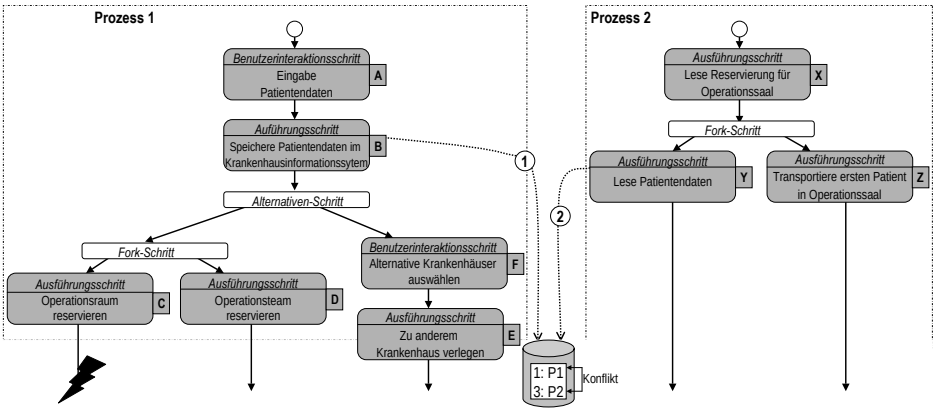

Abbildung 4: Beispiel für das Rücksetzen

Abbildung 4 dient als Illustration für das partielle Rücksetzen. Das erweiterte Beispielszenario besteht aus einem Peer und zwei Prozessen. Prozess 1 entspricht dem Beispielprozess in Abbildung 2. Prozess 1 hat gerade die Schritte C und D erfolgreich ausführt, während Prozess 2 die Schritte Y und Z ebenso erfolgreich beendet hat. Beide Prozesse haben dabei jeweils einmal einen Dienst auf dem Peer aufgerufen, wobei beide Dienstaufrufe in Konflikt zueinander stehen. Wir nehmen nun an, dass die Ausführung des entsprechenden Klons nach Schritt C scheitert. Bei einem einfachen Transaktionsmodell würde dies bedeuten, dass Prozess 1 ganz zurückzusetzen ist (was natürlich in dem Anwendungskontext keinen Sinn macht). Da aber Prozess 2 nach Prozess 1 eine konfligierende Aktion auf diesem Peer ausgeführt hat, wäre auch Prozess 2 (vollständig) zurückzusetzen.

Durch die Einführung von Alternativpfaden [SABS02] verhindern wir einen Teil der Pro-

bleme. So wird in unserem Fall ausgeschlossen, dass Prozess 1 nach den Schritten C und D auch die Schritte A und B zurücksetzt. Stattdessen kann Prozess 1 die Ausführung mit Schritt F fortsetzen. Trotzdem wäre es immer noch erforderlich, Prozess 2 vollständig zurückzusetzen. Um dies zu vermeiden, könnte Schritt Y zurückgesetzt und später wieder neu ausgeführt werden.

Während dieser Unterschied auf den ersten Blick vernachlässigbar erscheint, ändert sich die Situation, wenn man folgende Aspekte berücksichtigt:

1. Die Prozesse können komplex sein und gleichzeitig viele Schritte enthalten und damit viele Peers in Anspruch nehmen.
2. Einzelne Schritte können längere Zeit zur Ausführung benötigen, entweder weil sie Benutzerinteraktionen beinhalten oder rechenintensiv sind.
3. Ein Benutzer sollte möglichst nur dann eine Eingabemaske ein weiteres Mal präsentiert bekommen, wenn sich die dort dargestellte Information verändert hat, etwa wenn ein von ihm gewähltes Operationsteam auf einmal doch nicht verfügbar ist.

4.1 Ermitteln der Rücksetzmenge

Die theoretische Grundlage für die Ermittlung der Rücksetzmenge liefert die *Unified Theory for Concurrency Control and Recovery* [VHBS98], welche das Korrekheitkriterium *Serializability with ordered termination (SOT)* definiert. Dieses Kriterium berücksichtigt gleichzeitig Concurrency Control und Recovery – gerade auch für semantisch reiche Operationen. Wir verwenden dieses Kriterium, um im Fehlerfall zu ermitteln, welche Dienstaufrufe kaskadierend aufgrund von Konflikten zwischen Dienstaufrufen zurückzusetzen und in welcher Reihenfolge hierzu inverse Dienste aufzurufen sind. Wie bereits erläutert, beschränken wir hier die Diskussion auf einige Fehlerarten: Provider- und Dienstfehler sowie Isolationsverletzungen.

Initiale Rücksetzgrenze. Die initiale Rücksetzgrenze gibt an, wie weit ein Prozess mindestens zurückgesetzt werden muss, um einen vorliegenden Fehler zu beheben. Diese Grenze wird zu Beginn der Recovery-Phase ermittelt und ist abhängig von der Art des aufgetretenen Fehlers. Wie bereits vorher erläutert, müssen Provider- und Dienstfehler durch die Prozessspezifikation abgedeckt werden. Eine solche Spezifikationsmöglichkeit stellen die Verzweigungspunkte dar. Falls die Ausführung scheitert, so wird so weit zurückgesetzt, bis ein solcher Verzweigungspunkt gefunden wird, der über einen noch unversuchten Alternativpfad verfügt. Nur falls kein solcher Pfad gefunden wird, scheitert die Prozessausführung endgültig. In unserem Beispiel in Abbildung 4 ist der Verzweigungspunkt der initiale Rücksetzschritt. Dieser Ansatz reflektiert, dass es in unserem Beispiel kaum sinnvoll ist, endlos nach einem Operationsteam zu suchen, sondern dass irgendwann eine andere Lösung gefunden, nämlich der Patient verlegt werden muss.

Neben den explizit im Prozess sichtbaren Alternativen können auch Benutzerinteraktionsaktivitäten Alternativen bieten, nämlich dann, wenn der Benutzer eine Auswahl getroffen hat, die letztlich für das spätere Scheitern der Ausführung verantwortlich war. Falls ein

Patient verlegt werden soll und das ausgewählte, bestgeeignete Krankenhaus ihn nicht aufnehmen kann, könnte der Benutzer weitere (weniger geeignete) Krankenhäuser zulassen. Somit sind beim Ermitteln der initialen Rücksetzgrenze Benutzerinteraktionsschritte (unter der Voraussetzung, dass der Benutzer dort eine Wahl treffen konnte) genauso wie Verzweigungspunkte zu behandeln. Zusätzlich muss dem Benutzer zur Laufzeit erklärt werden, weshalb er noch einmal eine (andere) Wahl treffen soll.

Bei einer *Isolationsverletzung* sieht die Situation anders aus. Die Ausführung der einzelnen Prozesse ist zwar isoliert gesehen erfolgreich, allerdings gibt es gesamthaft Wechselwirkungen und Abhängigkeiten zwischen den Prozessen, die zu einem nicht-serialisierbaren – und damit inkorrekten – Ablauf führen. Dies ist in unserem Ansatz immer dann der Fall, wenn als Konsequenz des optimistischen Concurrency-Control-Ansatzes ein Zyklus im Serialisierungsgraphen entsteht. In diesem Fall muss man durch Kompensation der ausgeführten Schritte so weit zurückzusetzen, bis die Isolationseigenschaft wieder hergestellt wird. Ein solches Rücksetzen kann durch einen Agenten ausgelöst werden, wenn er selbst feststellt, zurücksetzen zu müssen. Es kann aber auch sein, dass ein anderer Agent diese Notwendigkeit zuerst erkennt und Ersteren darüber informiert.

Transitive Rücksetzschritte. Zum Beheben von Fehlern kann es notwendig sein, dass transitiv mehrere Schritte kompensiert werden müssen. Diese nennen wir *transitive Rücksetzschritte*. Für ihre Ermittlung ist es unwesentlich, welcher Fehler das Zurücksetzen ausgelöst hat. Als *transitive prozessinterne Rücksetzschritte* bezeichnen wir die Schritte, die innerhalb des rücksetzenden Prozesses kompensiert werden müssen, um sicherzustellen, dass der Prozess entsprechend der Spezifikation ausgeführt wird. So sind parallele Schritte auch dann zurückzusetzen, falls die Rücksetzschranke vor dem korrespondierenden Fork-Schritt liegt. Falls in unserem Beispiel das Reservieren eines Operationssaals scheitert, reicht es nicht einfach aus, nur den Schritt 'Patient verlegen' auszuführen. Zusätzlich muss der Schritt 'Operationsteam reservieren' kompensiert werden.

Als zweiter Grund für das transitive Rücksetzen kommen andere Prozesse in Frage. Die daraufhin zurückzusetzenden Aktivitäten bezeichnen wir als *transitive prozessübergreifende Rücksetzschritte*. Sei a_R eine zurückzusetzende Aktivität. Unter der Annahme, dass genau dann, wenn Aktivität a im Konflikt zu b steht, auch die inverse Operation a^{-1} mit b konfligiert, muss eine Aktivität a' vorher zurückgesetzt werden, falls a_r und a' im Konflikt stehen und $a_R < a'$ gilt. Falls also in Abbildung 4 der Schritt B von Prozess 1 kompensiert wird, so muss auch Schritt Y von Prozess 2 zurückgesetzt werden.[3] Für die transitiven Rücksetzschritte gilt, dass die Regeln zur Ermittlung von transitiven Rücksetzschritten unter Umständen mehrfach angewendet werden müssen.

[3] Voraussetzung hierfür ist, dass alle Agenten über einen erweiterten Serialisierungsgraphen verfügen. In diesem ist jede Kante mit der/den Aktion/en annotiert, welche die Kante(n) verursacht hat/haben. In dem Beispiel in Abbildung 4 würde Prozess 1 Prozess 2 informieren, dass Letzterer eine Kante zurückzunehmen hat. Prozess 2 schaut dann in seinem Serialisierungsgraphen nach und stellt fest, dass eben Schritt Y die Kante verursacht hat und somit Y die (lokale) initiale Rücksetzgrenze ist.

4.2 Rücksetzen

Nachdem die Menge der zu kompensierenden Schritte (vorläufig) ermittelt ist, kann das eigentliche Rücksetzen beginnen. Das Rücksetzen verläuft sehr ähnlich zur Vorwärtsausführung: Falls eine *Dienstaktivität* zurückgesetzt wird, die auf verschiedenen Peers ausgeführt wurde, so werden Klone erzeugt. Diese migrieren zu den entsprechenden Peers, um dort die Kompensationsoperationen auszuführen. Dabei kann das Problem auftreten, dass eine Operation nicht direkt kompensiert werden darf. Dies ist der Fall, falls nach der ursprünglichen Operation eine weitere Operation ausgeführt wurde, die mit der Kompensationsoperation in Konflikt steht. In diesem Fall muss der zurücksetzende Prozess den anderen Prozess auffordern, partiell zurückzusetzen und abwarten, bis dies erfolgt ist. In unserem Beispiel kann Prozess 1 also erst dann Schritt B zurücksetzen, nachdem – auf seine Anforderung hin – Prozess 2 den Schritt Y kompensiert hat.

Die *Join-Schritte* und *Fork-Schritte* wechseln in der Recovery-Phase ihre Funktionen: Bei einem Join-Schritt werden nun Klone gebildet, während bei einem Fork-Schritt die Klone aufeinander warten und schließlich verschmelzen. Bei einem *Benutzerinteraktionsschritt* ist der Anwender über das Rücksetzen und den Grund dafür aufzuklären.

5 Verwandte Arbeiten

Das AMOR-Projekt ist an der Schnittstelle zwischen mobilen Agenten, Workflow-Management und Transaktionsverwaltung angesiedelt. In diesem Papier stand speziell die Systemarchitektur für die Prozessausführung mittels mobiler Agenten und die flexible Fehlerbehandlung im Vordergrund. Besonders sind wir auf unsere flexible Fehlerbehandlungsstrategie eingegangen.

Wichtige Arbeiten im Bereich Fehlerbehandlung bei Workflows sind [KR98] und [EL96]. Die letztgenannte Arbeit legt ihr Gewicht dabei weniger auf den Bereich Concurrency-Control, sondern konzentriert sich eher auf die zuverläsissige Workflowausführung. Die Stärke dieser Arbeit liegt dabei in der Kombination einer Klassifikation von Workflowarten und von Recovery-Konzepten speziell für einzelne Workflowarten, allerdings wird keine einheitliche, gemeinsame Betrachungsweise entwickelt. Ein weiterer, entscheidender Unterschied ist das zentralistische Ausführungsmodell, welches auch der zweiten Arbeit [KR98] zugrunde liegt. Letztere präsentiert ein detailliertes Konzept, bei dem der Benutzer genau angeben kann/muss, wie weit zurückzusetzen ist bzw. welche Abhängigkeiten innerhalb eines Prozesses speziell beim Rücksetzen zu beachten sind. Somit wird dort – anders als in AMOR – die Information, was zu kompensieren ist, nicht automatisch aus der Prozessbeschreibung ermittelt.

Die Ausführung von Workflows durch mobile Agenten wurde erstmalig in [CGN96] diskutiert. In dieser frühen Arbeit werden noch keine transaktionalen Garantien berücksichtigt. Diesen Aspekt berücksichtigen erst spätere Arbeiten, darunter [SP98], in welcher insbesondere mittels Replikation eine fehlertolerante Ausführung erreicht wird. Während sich diese Arbeit eher mit dem Problem der Atomarität beschäftigt, behandelt [SAE01] als weitere wichtige Arbeit auch Fragen der Isolation, wobei aber der Recovery-Ansatz nicht konsequent diskutiert wird.

Falls man nicht nur auf mobile Agenten fokussiert, sondern auch den Zugriff von nicht-mobilen Agenten auf Datenbanken berücksichtigt, so ist hier [PB95] zu nennen. Die Arbeit bietet gute, abstrakte Diskussion der Möglichkeiten des Zugriffs auf Datenbanken durch Agenten. Ein anderer Ansatz beschreibt [CD00]. Dieses Papier behandelt, wie mehrere nicht-mobile Agenten kooperieren können, um Transaktionen atomare auszuführen. Weiterhin ist die Arbeit von [Nag01] zu nennen. Diese Arbeit behandelt, wie die im KI-Umfeld oft eingesetzten, nicht-mobilen BDI-Agenten ausgeführt werden können. Das Schwergewicht liegt dabei, diese 'robust' gegenüber Fehlern der Umgebung zu machen.

Das AMOR-System basiert auf einem Peer-to-Peer-Netzwerk. Vorangetrieben durch die Popularität von File-Sharing-Anwendungen wie Gnutella [Gnu] wird auf dem Gebiet der Peer-to-Peer-Netzwerke in jüngster Zeit sehr intensiv geforscht. Die akademische Forschung in diesem Bereich konzentriert sich vor allem auf effiziente Zugriffsmethoden, wie zum Beispiel dem P-Grid [Abe01]. Solche Methoden sind dabei orthogonal zu der Richtung des AMOR-Projektes und könnten zur Effizienzsteigerung in den Prototypen integriert werden.

6 Zusammenfassung und Ausblick

In diesem Papier haben wir erläutert, wie Ad-hoc-Prozesse mittels mobiler Agenten in Peer-to-Peer-Netzwerken ausgeführt werden können. Hierzu haben wir die notwendige Infrastruktur vorgestellt. Wir sind ausführlich auf den Aspekt der Fehlerbehandlung eingegangen, indem wir die verschiedenen potentiellen Fehlerfälle skizziert haben. Wesentlich dabei ist die Unterscheidung, ob ein Fehler von dem Ausführungsmodell zu behandeln oder bei der Prozessspezifikation zu berücksichtigen ist. Das eigentliche Rücksetzen erfolgt partiell. Während bei Isolationsfehlern der gleiche Ausführungspfad erneut versucht wird, wählt das System bei Provider- oder Dienstfehlern einen alternativen Pfad.

Die in diesem Papier beschriebenen Strategien für zuverlässige Umgebungen sind bereits im Rahmen des AMOR-Projektes der Datenbankgruppe der ETH Zürich implementiert. In aktuellen Arbeiten, die bereits weitgehend Eingang in unseren Prototypen gefunden haben, untersuchen wir die Auswirkungen dynamischer und unzuverlässiger Umgebungen. Damit beziehen wir Abstürze, Netzwerkpartitionierungen und auch geplante Unterbrechungen in unser Ausführungsmodell ein. Dies wirft zunächst die Frage auf, wie Fehlerfälle, die aus solchen ungeplanten Verbindungsfehlern entstehen, entdeckt und behandelt werden können, während für geplante Unterbrechungen eine erweiterte Strategie notwendig ist, da nur so bei Bedarf eine geeignete Fehlerbehandlung initiiert werden kann. Sie besteht dabei aus dem Starten eines neuen Agenten, der den begonnenen Prozess fortsetzt.

Um eine potentiell doppelte Ausführung einer Prozessinstanz zu vermeiden, muss das System ausschließen, dass ein verloren geglaubter Agent plötzlich wieder auftaucht und seine Prozessausführung fortsetzt. Um trotz all dieser potentiellen Probleme unsere Prozesse nicht nur korrekt, sondern auch möglichst effizient ausführen zu können, berücksichtigen wir daher weiterhin verschiedene Quality-of-Service-Parameter bei der Ausführung.

Literatur

[Abe01] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *9th Int. Conf. on Cooperative Information Systems, Trento, Italy*, 2001.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[CD00] Q. Chen and U. Dayal. Multi-Agent Cooperative Transactions for E-Commerce. In *7th Int. Conference on Cooperative Information Systems, Eilat, Israel*, 2000.

[CGN96] T. Cai, P. Gloor, and S. Nog. DartFlow: A Workflow Management System on the Web using Transportable Agents. Technical Report TR96-283, Dartmouth College, 1996.

[EL96] J. Eder and W. Liebhart. Workflow Recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96) , Brussels, Belgium*, 1996.

[Gnu] Gnutella. http://www.gnutella.com.

[HS01] K. Haller and H. Schuldt. Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment. In *5th Int. Conf. on Mobile Agents (MA), Atlanta, GA*, 2001.

[HS03] K. Haller and H. Schuldt. Consistent Process Execution in Peer-to-Peer Information Systems. In *CAiSE'03, Velden, Austria*, 2003.

[HSS03] K. Haller, H. Schuldt, and H.-J. Schek. Transactional Peer-to-Peer Information Processing: The AMOR Approach. In *4th International Conference on Mobile Data Management, Melbourne, Australia*, 2003.

[KR98] M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *ICDE, Orlando, Florida, USA*, 1998.

[MQS03] MQSeries Workflow – Version 3, Release 3.2, 2003. IBM, International Business Machines Corporation, http://www-3.ibm.com/software/ts/mqseries/workflow/v332/index.html.

[Nag01] K. Nagi. *Transactional Agents – Towards a Robust Multi-Agent-System, LNCS 2249*. Springer, 2001.

[PB95] E. Pitoura and B. Bhargava. A framework for providing consistent and recoverable agent-based access to heterogeneous mobile databases. *SIGMOD Record*, 24(3):44–49, 1995.

[Rei00] M. Reichert. Prozessmanagement im Krankenhaus - Nutzen, Anforderungen und Visionen. *das Krankenhaus*, 11(92), 2000.

[SABS02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1), 2002. To appear.

[SAE01] R. Sher, Y. Aridor, and O. Etzion. Mobile Transactional Agents. In *21st Int. Conf. on Distributed Computing Systems, Phoenix, AZ*, 2001.

[SP98] A. Silva and R. Popescu-Zeletin. An Approach for Providing Mobile Agent Fault Tolerance. In *2d Int. Workshop on Mobile Agents, Stuttgart, Germany*, 1998.

[SSSW02] H.-J. Schek, H. Schuldt, C. Schuler, and R. Weber. Infrastructure for Information Spaces. In *Proceedings of Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002*, Bratislava, Slovakia, 2002.

[VHBS98] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, 190(2), 1998.