Fehlertoleranz mobiler Agenten

Von der Fakultät Informatik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Markus Straßer

aus Urach, jetzt Bad Urach

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Tag der mündlichen Prüfung:	19. 11. 2002

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2003

Danksagung

Mein besonderer Dank gilt meinem Doktorvater, Prof. Dr. Kurt Rothermel, der mich durch seine Bereitschaft zu Diskussionen und seine Denk- und Motivationsanstöße sehr konstruktiv bei der Durchführung meines Forschungsvorhabens unterstützt und betreut hat.

Ebenfalls sehr herzlich bedanken möchte ich mich bei meinem Zweitbetreuer, Prof. Dr. Bernhard Mitschang, für die Bereitschaft, als Mitberichter zur Verfügung zu stehen und für alle geführten Diskussionen und Tips, die er mir gegeben hat.

Einen sehr positiven Einfluß auf das Gelingen dieser Arbeit hatten neben meinem langjährigen Büro- und Teamkollegen Wolfgang Theilmann insbesondere auch die anderen Mitglieder des Mole-Teams, Joachim Baumann, Fritz Hohl und Markus Schwehm. Ohne die vielen, konstruktiven Diskussionen und die Motivation, die sich aus der Gruppe heraus entwickelte, wäre der Weg zum Ziel wesentlich steiniger gewesen. Ein herzliches Dankeschön, auch für das Korrekturlesen dieser Arbeit!

Ebenfalls sehr hilfreich, interessant und angenehm war die Zusammenarbeit mit vielen Studenten, die in unserem Team ihre Studien- und Diplomarbeiten durchführten.

Schließlich will ich mich auch bei den restlichen Kollegen der Abteilung Verteilte Systeme für die gute, hilfreiche und sehr humorvolle Zusammenarbeit und für viele nützliche Diskussionen bedanken.

Meinem derzeitigen Kollegen Carl Mayer danke ich für das Korrekturlesen der englischen Zusammenfassung.

Zu guter letzt möchte ich mich bei denen bedanken, ohne die diese Arbeit gar nicht möglich gewesen wäre: meinen Eltern. Sie waren immer für mich da, haben mir den Rücken gestärkt und haben trotz des für sie sehr trockenen Themas diese Arbeit Korrektur gelesen.

Markus Straßer

Kurzfassung

Mobile Agenten sind Programme, die sich zur Ausführung ihrer Aufgabe autonom zwischen Ausführungsumgebungen in einem Rechnernetz bewegen können, um die jeweiligen lokalen Ressourcen der Rechner zu nutzen. Neben Lösungen im Bereich der Sicherheit und der Kontrolle mobiler Agenten ist die Zuverlässigkeit der Ausführung von mobilen Agenten eine der Grundvoraussetzungen für die breite Anwendung dieser Technologie. Die vorliegende Arbeit erarbeitet Lösungen für die zuverlässige Ausführung mobiler Agenten und das zuverlässige, partielle Rücksetzen der Agentenausführung.

Um den Ausfall von Rechnern, auf denen ein Agent einen Teil seiner Aufgabe ausführen möchte, einfacher tolerieren zu können ist es vorteilhaft, wenn dem Agenten Alternativen zur Auswahl stehen. Als Basisbaustein für die fehlertolerante Ausführung wird daher ein flexibles Reiseroutenkonzept entwickelt. Dieses Konzept erlaubt nicht nur die Spezifikation alternativer Ausführungsrechner sondern erlaubt auch, die einer Anwendung inhärenten Alternativen der Ausführungsreihenfolge der einzelnen Aufgabenteile offenzulegen.

Für die fehlertolerante Ausführung werden zwei Mechanismen entwickelt. Der Basismechanismus stellt die genau-einmal Ausführung eines Agenten durch transaktionale Ausführung des Agenten sicher. Die blockierungsfreie Ausführung von Agenten wird durch eine Erweiterung dieses Basismechanismus sichergestellt, bei der die Ausführung des Agenten durch mehrere Rechner überwacht und im Fehlerfalle weitergeführt wird. Die Korrektheit der Mechanismen wird in einem informalen Beweis nachgewiesen und eine analytische Bewertung der Mechanismen durchgeführt.

Da für das partielle Rücksetzen der Agentenausführung ein einfaches Rücksetzen auf einen alten Zustand nicht ausreicht, kommen hierfür Kompensationsoperationen zum Einsatz. Die zuverlässige Ausführung der Kompensation wird durch transaktionale Ausführung sichergestellt. Um den Anwendungsentwickler möglichst stark zu entlasten, werden die Daten des Agenten in zwei unterschiedliche Klassen aufgeteilt. Ein Teil der Daten des Agenten kann von der Ausführungsumgebung durch eine Kopie des alten Zustandes zurückgesetzt werden. Für den anderen Teil der Agentendaten und für die Ressourcen müssen Kompensationsoperationen zur Verfügung gestellt werden. Eine Klassifizierung der Kompensationsoperationen erlaubt Optimierungen bei der Kompensation.

Fault-Tolerance of Mobile Agents

Extended english abstract of the dissertation “Fehlertoleranz mobiler Agenten”

1 Introduction

The current, rapid development of the information society comes along with the opening of the Internet to the broad masses. The increase of the Internet size and capacity during the last decade has been extraordinary and has been seconded only by the even vaster increase of Internet users. New developments in the area of consumer electronics provide easy access to the Internet (e.g. by using web pads) or allow the mobile access to the net. Additionally, the number and variety of services provided via Internet increases permanently. This development results in several problems and new challenges, for example the permanent overload of the Internet or the mobile Internet access using unstable GSM connections with low bandwidth.

A technology to solve several of these problems has been introduced in the early 1990s: mobile agents. Mobile agents are programs, which autonomously execute a task on behalf of their owner. While executing their task, mobile agents are able to move within a network of computers, i.e. they are able to shift their program execution from one computer to another (migration). This enables an agent to efficiently access resources locally instead of accessing these resources using global communication. An example illustrates this: An agent has to search some specific information in a huge database using a sophisticated information retrieval algorithm. Since the database allows only simple searches, the agent has to read most of the database to finally find the required information. If the database is not local to the agents execution environment, all the information has to be transferred to the agent. However, if the agent is able to migrate to the database, only the agent (i.e. its program code and execution state) and the result has to be transferred over the network. Depending on the agent size and the size of the result, this may be a considerable smaller amount of data which has to be transferred over the network.

Although the technology of mobile agents seems to be promising for several application areas, it is not used wide-spread up to now. One of the main reasons is, that some of the topics which are very important for the commercial use of mobile agents – control mechanisms, security and fault-tolerance – are still subject to research. This thesis deals with the fault-tolerance of mobile agents. The following scenario from the area of electronic commerce outlines the importance of fault-tolerance in the area of mobile agents: A concierge agent has the task to organize a business trip. It first travels to the server of the airline and books the flight. Afterwards, it reserves a car and the hotel on the respective servers and sends the result to the user. Since the agent moves autonomously, the user can not directly monitor the agents actions. Therefore, it is impossible for the user to detect that the agent has been lost or blocked due to a system failure. Furthermore, it is not acceptable that the agent executes only a part of its task or that it executes

some parts more than once (e.g. books two flights). For reliable execution, it has to be ensured that the agent executes its task exactly-once as fast as possible without being blocked even in the presence of system failures. Currently available mobile agent platforms do not cover these requirements.

Based on a realistic system and error model, this thesis develops an algorithm which ensures exactly-once execution of an agents task and prevents the agent from being blocked by system failures. As a prerequisite to this algorithm, an itinerary concept is developed which allows to specify the application-inherent alternatives regarding the execution order of the sub-tasks of an agent as well as the possible alternatives regarding the computers where the sub-tasks have to be executed. Finally, based on the mechanism for the exactly-once execution of mobile agents, a mechanism is developed which allows the partial rollback of an agents execution.

2 Mobile agents

As defined by WHITE (1997), mobile agents are programs autonomously executing a task on behalf of their owner. In order to do this, they can move between execution environments (or hosts, also called places), use the resources offered by these environments and communicate with other agents.

For the mechanisms in the following sections, a more detailed model of mobile agents is necessary: The task of an agent consists of several sub-tasks which are executed sequential. A sub-task is also called a *step*. A step can be executed several times during an agents lifetime, e.g. the step *queryPriceList* has to be executed in several shops before the shopping agent buys some goods at the cheapest shop. Each step is executed entirely inside one execution environment. Several (not necessarily consecutive) steps may be executed in the same execution environment (place). During the execution of a step, the agent can access the resources and services offered by the place. On each physical host exists at most one execution environment for mobile agents. Migration between places is performed using weak migration, i.e. only the agents program code and data state (global variables) but not the execution state (stack, local variables) are transferred to the target place (see CUGOLA ET AL. (1996)). Agents have a globally unique name which is non-mutable during the agents execution. Mobile agents can not communicate to other mobile agents and are not able to start new mobile agents (these last two restrictions are necessary for the mechanisms developed in the next sections).

3 Itineraries

While performing a job, a mobile agent often has to visit several places to access local services. In many cases, some (or all) of these places are either known before agent initialization or can be determined by the agent several steps in advance. However, as in real life, there is no strict order in which the places have to be visited. For example, an agent having to order a CD and a theatre ticket, may perform these tasks in any sequence. Furthermore, if there are several branches of a music shop, the agent needs to visit only one of these branches. If this freedom in

an agents travel plan is made visible to the mobile agent system, this can be exploited in different ways. One possibility is, that the agent system may calculate the shortest possible path for the agent. Another, more important possibility is that this information may be used to increase the level of fault-tolerance in the system. If the system can choose between several places which the agent can visit and one of these places is currently unavailable, it is able to postpone the visit of this place automatically or may choose an alternative place instead.

To clarify this, let us consider the following scenario: Paul, planning to spend a romantic evening with his wife, instructs his personal concierge agent to order some flowers, to buy a ticket for the theatre and to make a reservation for a table in a nice restaurant close to the theatre. The play for which the agent has to buy tickets is currently enacted in two different theatres. To fulfil the job, the agent has to visit the place of the flower service, one of the two places offering the ticket service for the respective theatre (unfortunately, there is no central ticket service for both theatres), and, depending on the chosen theatre, the place of the restaurant. An itinerary for this example has to specify, that, besides ordering the flowers in a flower shop, the agent has to visit one of the two places offering the ticket service and, afterwards, the respective place of the restaurant.

The tree in figure 3-1 shows all possible paths that can be taken by Paul's agent. The path, where the agent first orders the flowers, then buys a ticket in the ModernArts theatre and finally makes a reservation for a table at the BeefHouse is marked bold. The tree shows, that in each step except the last one there is more than one possible target for the agent to be transferred to. If, for example, the agent has started by ordering the flowers, the system may chose to transfer the agent to the CentralTheatre or the ModernArts place alternatively. Therefore, the agent may proceed even if one of those two places is unavailable. In the thesis, two itinerary concepts providing the necessary flexibility are developed. The more flexible of the developed concepts allows to define itineraries by specifying preconditions which have to be fulfilled in order to execute a step and it supports the nesting of itineraries. A detailed description of those concepts can be found in STRASSER UND ROTHERMEL (1998).

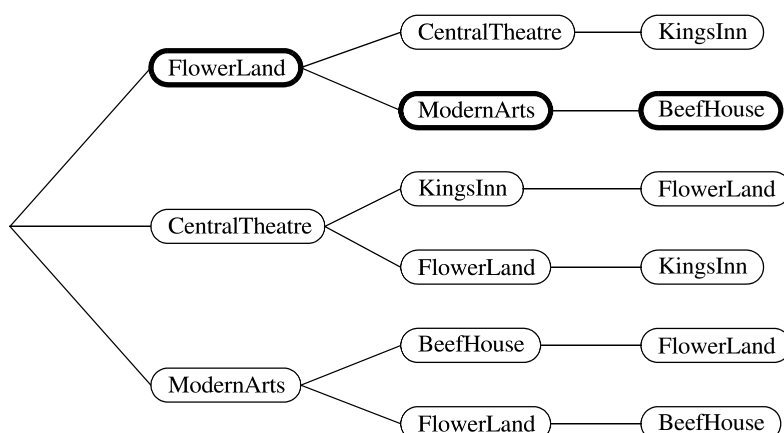


Figure 3-1. Tree of possible paths of the concierge agent.

4 Exactly-once execution

In this section, two algorithms for the exactly-once execution are developed based on the system model, the failure model and the definition for exactly-one execution of mobile agents given at the beginning of the section.

The algorithms described in this chapter have already been published in ROTHERMEL UND STRASSER (1998), STRASSER UND ROTHERMEL (1998) and STRASSER, ROTHERMEL UND MAIHÖFER (1998).

4.1 System model and failure model

This section defines the system and failure model used in the thesis. The system model describes the relevant parts of a system. A distributed system consists of several places (execution environments) which are interconnected by a network. A place consists of a processor and private volatile and stable storage. A program is executed within a process, concurrent threads within a process are possible. To manage timeouts, each place has a correct clock. Communication between places is performed by message exchange using communication channels. There is a communication channel between each pair of processes. The system is asynchronous (no upper time limit for the execution of commands or for message transmission).

The failure model defines the possible failures in the system. Places only suffer from crash failures. If a place crashes, all programs on the place stop executing – the crash of a (real) subset of the processes is not possible. All information in the volatile storage (execution state of the processes) is lost in case of a crash, information stored on stable storage is not lost. The stable storage and communication between processes in a place are error-free. The network also suffers from crash failures, failures always result in network partitions. Places within a network partition can communicate, places in different partitions can not. Communication within a partition is error-free. Since the system is asynchronous, place failures and network failures can not be distinguished. Crashes are only temporary and the crashed component (place/network) is available again after restart/repair.

4.2 Definition "exactly-once execution"

The exactly-once property has already been defined for RPC systems by SPECTOR (1982), where he defines the failure semantics of a single remote procedure. However, in the context of mobile agents, a sequence of agent steps is to be considered rather than a single procedure. Therefore, the definition of the exactly-once property of mobile agents is based on the information contained in the agent's itinerary and on the steps to be performed at the visited places.

Let $P=\{P_1,...,P_n\}$ be the set of all possible paths the agent may take for a given itinerary, let $L(P_i)$ be the number of places in path $P_i=[N_{i,1},N_{i,2},...,N_{i,L(Pi)}]$ and let $S_{i,j}$ be the step to be performed

on the j-th place $N_{i,j}$ of path P_i ($1 \le i \le n$, $1 \le j \le L(P_i)$). Then the execution of an agent is defined to be exactly-once iff

- only places $N_{i,1},...,N_{i,L(Pi)}$ belonging to one path $P_i \in P$ are visited,
- the agent executes step $S_{i,j}$ before step $S_{i,j+1}$, $1 \le j < L(P_i)$, and
- each step $S_{i,j}$ $1 \le j \le L(P_i)$ is executed exactly once.

In the scenario of Section 3, an electronic commerce system providing the exactly-once property for mobile agents guarantees, that the agent visits the flower shop, only one of the two theatres and the restaurant associated with that theatre. The steps which have to be executed on these places are performed in one of the orders defined by the tree of possible paths in figure 3-1. Each of these steps is executed exactly once.

4.3 The basic protocol

The exactly-once property of mobile agents can be achieved in a simple way by using transactional message queues. Transactional message queues provide for persistent messages and ensure the exactly-once delivery (see e.g. GRAY UND REUTER (1993)). Moreover, the *Put* and *Get* operations, which put a message in a queue or get a message out of a queue, can be performed within ACID transactions (see e.g. HÄRDER UND REUTER (1983)).

Figure 4-1 depicts how transactional message queues can be used to implement exactly-once agents. At each migration and at the start of an agent, the run-time system chooses one of the possible destinations of the agent. The place can be chosen either randomly or by performing optimizations, e.g. calculating a shortest path, using the information contained in the *itinerary.* The possible destinations an agent may visit next according to the itinerary are contained in the *NextSet*, which is provided by the *itinerary.* At the start, the agent is put in the input queue of the first place in the path. Once the agent has been stored in this initial queue (Q_1 in our example), the owner of the agent can be informed that this agent - provided that a crashed place recovers eventually - will eventually be performed exactly once.

All places (with the exception of the last one, $N_{i,k}$) perform the following sequence of operations: *Begin_Transaction*; *Get*(*Agent*); *Execute*(*Agent*); *Put*(*Agent*); *Commit*. *Get* removes an agent from the place's input queue, *Execute* performs the received agent locally, and *Put* places it in the input queue of the next place. All three operations are performed within a transaction and hence build an atomic unit of work. So, if for instance transaction T_j aborts due to a place

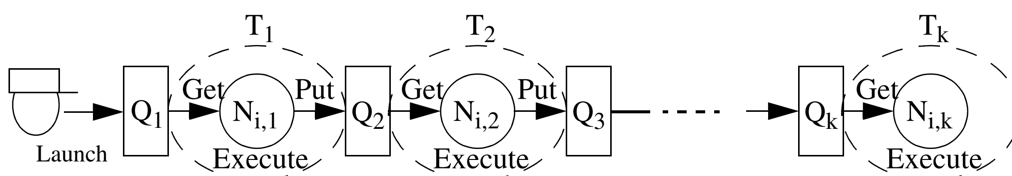


Figure 4-1. Exactly-once execution of mobile agents with the base protocol.

or transaction failure, recovery undoes all of the agent's effects at $N_{i,j}$ and restores the agent in its original state in Q_j. Any effects in Q_{j+1} are also undone. After recovery is finished, $N_{i,j}$ continues normal processing and will eventually execute this agent and then hand it over to its successor.

Although this protocol guarantees the exactly-once property of mobile agents, it is possible that agents are caught in the (local) input queue of a place that crashes after the agents have been put into the queue. A partitioning of the underlying network may have similar effects. In contrast to client/server processing, where a client calling the operations of a server monitors the availability of this server, there is no "natural" instance to monitor the progress of an agent because of the autonomy of mobile agents. Therefore, a novel protocol is presented in the following section that enables the system to monitor the agents execution and, if necessary, allows it to react on failures by executing the agent on alternative places.

4.4 The blocking-free protocol

To allow for fault-tolerance, the execution model described above is extended by the concept of *stages* (see also SCHNEIDER (1997A)): for each step there is a non-empty set of places, called a stage, which can perform the next step alternatively. One place of the stage, holding the role of the *worker*, executes the agent while the other places of the stage, the *observers*, monitor the availability of the stage's worker. When the worker becomes unavailable (caused by either the failure of the worker place or a network partition), this will be detected by the observer places, which will then elect a new worker from the set of available stage places. Each stage place is associated with a priority (which may be specified in the itinerary or determined by the system) that defines a total ordering between the places belonging to the same stage (which is required for the voting and selection process). The initial worker of a stage will become the place with the highest priority. Figure 4-2 shows a three-stage execution of an agent. For example, stage S_1 is associated with one worker and four observers. In S_2, the place with the highest priority (1) fails and the place with priority 2 is selected to be the new worker.

The execution of an agent on the worker place is performed inside an ACID transaction. To start the agent, the worker place begins a new transaction, reads the agent from its transactional input

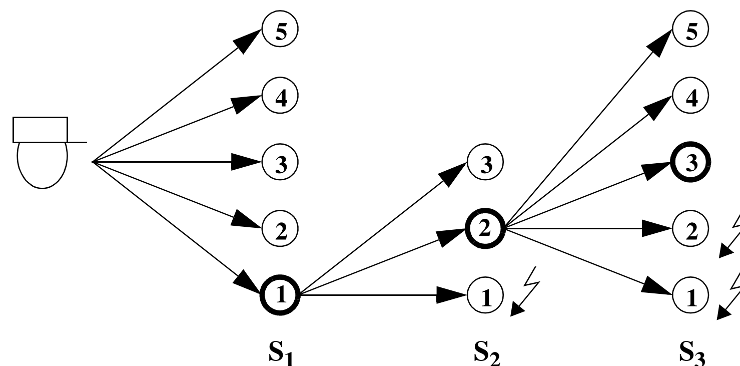


Figure 4-2. Execution of an agent in three stages

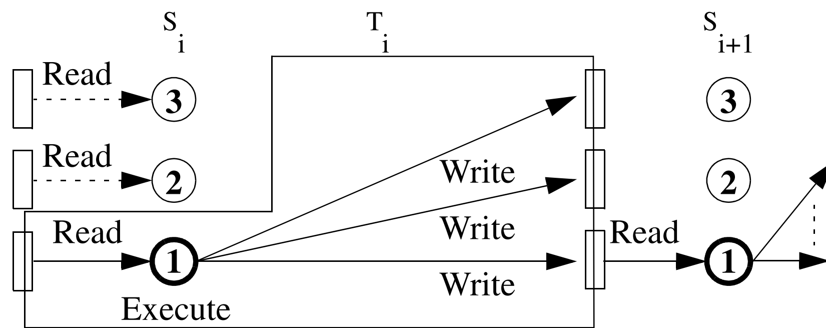


Figure 4-3. Transactional processing of an agent in a stage

queue and executes the agent. All actions of the agent are performed inside this transaction. After the agent issues the command to move to the next place, the worker puts the agent into the input queues of the places of the next stage and commits the transaction. Figure 4-3 shows the transactional execution of an agent in stage S_i. Please note that the *Read* operations of the observers are excluded from the transactions, because including them would require all stage places to be available to commit the stage.

The fault-tolerance of the protocol is provided by incorporating three different protocols into stage processing. The *monitoring protocol* monitors the availability of a stage's worker. The worker of a stage periodically sends *I_Am_Alive messages* to the observers of the stage. If an observer times out while waiting for an *I_Am_Alive message*, it assumes the worker to be unavailable (either due to a place crash or network partitioning) and initiates the selection protocol.

The selection of a new worker place is performed by the *selection protocol*. The selection protocol is a variant of the bully algorithm (see GARCIA-MOLINA (1982)). An observer detecting the failure of the worker sends an *Are_You_There* message to all places in the stage with a higher priority. Available places (observers as well as workers) reply to this message with an *I_Am_There* message. If no reply arrives within a reasonable time, the initiator decides to be the new worker and informs the other places of the stage about it. If the initiator receives a reply instead, it cancels its selection procedure and starts monitoring the new worker. This protocol results in the place with the highest priority being selected as the new worker.

In the presence of network partitioning, the protocol presented so far selects a worker in each partition. If two partitions are rejoined, two workers remain in the resulting partition. The voting protocol is responsible to ensure that only one worker may commit its transaction. This voting protocol has to be integrated into the two-phase commit protocol (2PC) of the transaction. When the transaction manager issues the prepare request, voting requests are sent to all stage places. A stage place receiving a voting request responds depending on the fact if it has already voted for another place in the stage or not. Only if a worker place gets a majority of votes from the other stage places, the transaction on this place commits and the other stage places are notified about the commit. Otherwise, the transaction is aborted on this place.

The thesis contains an informal proof of correctness of the protocol. Some considerations on communication overhead and an algorithm for stage construction contained in the thesis have been published in STRASSER, ROTHERMEL UND MAIHÖFER (1998).

4.5 Analytical evaluation

An in-depth analytical evaluation of the protocols has been performed. A comparison of the blocking probability shows, that the blocking-free protocol extremely reduces the probability that an agent gets blocked (however, the probability is not 0). Furthermore, the average time an agent stays in a stage has been calculated using a Markov model. The calculation shows, that the average time in a stage with several places is only moderately higher than in a stage with only one place (i.e. in the basic protocol). Therefore, the blocking-free protocol is well suited for any application that can not afford longer interruptions of the agent execution due to system errors.

5 Partial rollback

In this section, a protocol for the partial rollback of mobile agents executed using one of the exactly-once execution protocols of the previous section is presented. The basic ideas of this protocol are shown in figure 5-1.

The figure shows the execution of the steps S_i to S_{i+3} of the Agent A. During each step, the state of the agent as well as the state of the resources of the executing place are changed. A_i describes the agents data state before the execution of step S_i, R_i^1 and R_i^2 describe the state of the resources R_i of place P_i before and after the execution of the agent's *i-th* step. During the execution of step S_{i+3}, the agent (or the execution environment) decides, that the agent has to be rolled back to its last savepoint (savepoints have to be established by the agent). The step transaction is aborted and leaves the agent in the input queue of place P_{i+3}. Now, the steps S_{i+2}, S_{i+1} and S_i must be rolled back. Since those steps have been executed within committed transactions, it is

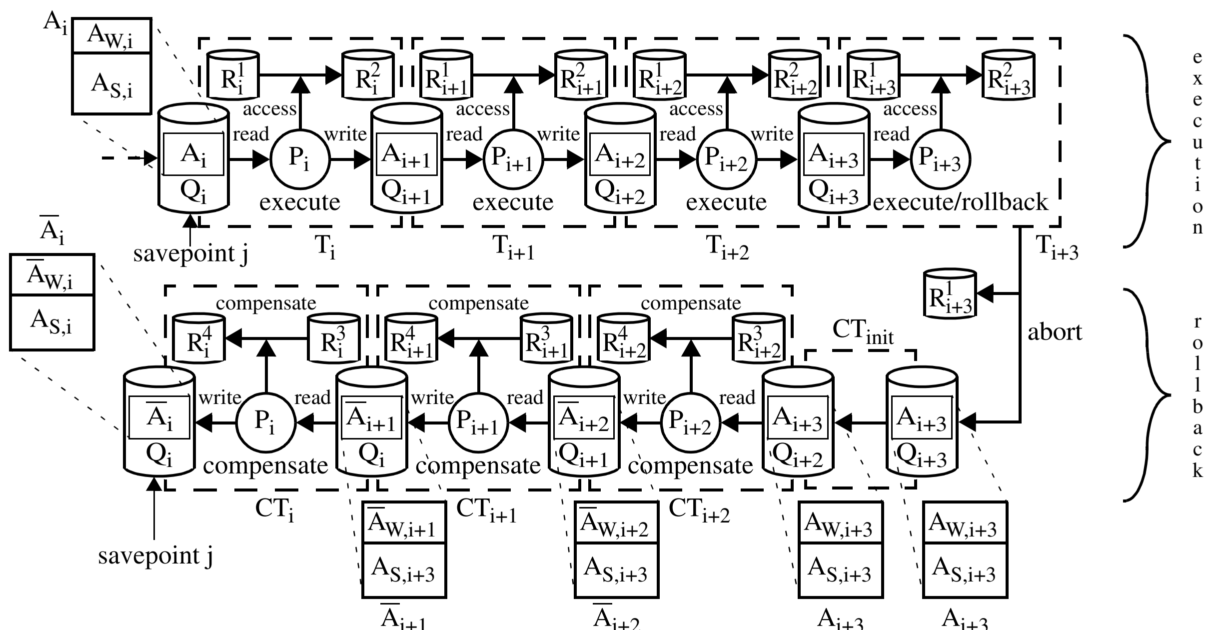


Figure 5-1. Partial rollback

not possible to simply restore the previous state of the resources of the places the agent visited in these three steps – another agent or program possibly already made changes to the resources. Therefore, the resources of the places have to be rolled back using *compensation operations*. As KORTH, LEVY UND SILBERSCHATZ (1990) have shown, the use of compensation operations for resources also requires the use of compensation operations for the program state. Therefore, the agents data state is rolled back also using compensation operations. To provide the exactly-once property for the agent rollback, the principles of the basic mechanism from the previous section are used for the compensation: the compensation of each step is executed within a compensation transaction *CT* on the same place on which the step has been executed. In figure 5-1, $\overline{A}_i$ symbolises the agents data state after compensation of step S_i, R_i^3 and R_i^4 describe the state of the resources R_i of place P_i before and after the compensation of that step. Since compensation is used, $\overline{A}_i$ and R_i^4 are in general not the same states as A_i and R_i^1 but only semantically equivalent states.

The compensation operations necessary for the rollback of the steps have to be provided by the agent developer. To simplify this task, the data of the agent is classified into two different types. *Strongly reversible data* is data that can be rolled back using a copy of the previous state. If, for example, an agent collects data and stores this data in a vector, this data can simply be rolled back by restoring the original state of the vector. $A_{S,i}$ symbolises the strongly reversible data part of the agents data in figure 5-1. *Weakly reversible data* (symbolised with $A_{W,i}$ and $\overline{A}_{W,i}$) is the (remaining) data of the agent which has to be rolled back using compensation operations. An example for such data is an electronic purse containing electronic coins. If a transaction where the agent paid goods using electronic coins has to be compensated, the original coins have already been transferred to the sellers bank and therefore the seller can only provide coins with the same value but different serial numbers.

The strongly reversible data (which has to be declared by the developer as strongly reversible) is automatically stored at savepoints. Therefore, no additional effort of the developer is necessary for the rollback of this data. Only for the compensation of the weakly reversible data and the resources data, compensation operations have to be provided.

The data necessary for the rollback is stored in the *agent rollback log*. It contains the data stored in savepoints and the compensation operations as well as additional data necessary for the compensation operations. For the strongly reversible data, *physical logging* (see HÄRDER UND REUTER (1983)) is used: an image of the strongly data is written into the log (*state logging*). For the weakly reversible data, logical logging is used: the compensation operations as well as their parameters are written into the log.

The agent rollback log is appended to the agent and migrates with the agent. Appending the log to the agent has two advantages. First, there is no need for any distributed actions to delete the log data when the agent finishes its execution. Secondly, the log data is always available whenever the agent is available and therefore, rollback is always possible as long as the resources which have to be rolled back are available. This second advantage is not important for the algo-

rithm described above, but an optimization described below benefits from it. A drawback of appending the log to the agent is, that the agent's size grows with the log size.

The thesis presents two optimizations of the rollback mechanism. The first optimization takes advantage of the fact that the rollback log is appended to the agent. If the agent did not alter any resources during its execution on a place, it is not necessary to migrate the agent to this place for the rollback – the operations performed on the agent's state on that place can be compensated wherever the agent resides. If e.g. the agent in figure 5-1 did not alter any data of place P_{i+1} during its step S_{i+1}, it is not necessary to migrate the agent to P_{i+1} during the rollback since the compensation operations on the agent state necessary to compensate S_{i+1} can be executed on place P_{i+2} as well as on P_i. Further optimizations allow to send compensation operations to the place where a step has been executed instead of migrating the agent.

The second optimization deals with the size of the rollback log. The main possibility to reduce the size of the rollback log is to reduce the number of savepoints to which the agent execution can be rolled back. The thesis proposes the integration of savepoint generation into an itinerary concept which allows the automatic deletion of log data which is not needed any more.

An early version of the algorithms has been presented in STRASSER UND ROTHERMEL (2000).

6 Conclusions

Mobile agents are a new and promising technology to develop distributed applications which currently suffers from the lack of mechanisms for reliable agent execution. This thesis closes the gap by presenting novel mechanisms for the exactly-once execution and rollback of mobile agents. The mechanisms for exactly-once execution guarantee that all sub-tasks of an agent are executed eventually exactly-once in a correct order despite the occurrence of system failures. The blocking-free variant additionally considerably reduces the probability that an agent is blocked due to system failures. The mechanism for partial rollback allows the partial rollback of an agent executed with one of the exactly-once execution mechanisms. The rollback is also performed exactly-once.

The exactly-once execution mechanisms can also be applied to non-agent applications. The prerequisite for their use is, that the application which has to be executed exactly-once must be able to write a savepoint on stable storage from which the execution can be resumed. If the blocking-free mechanism should be applied, it is necessary that the data written at a savepoint allows the resumption of the execution on any system contained in a stage. The general use of the presented rollback mechanisms is much more difficult and therefore subject to research.

Further investigations are also necessary to allow a single step to span the agent execution on several places and to incorporate the ability of agents which are executed exactly-once to directly communicate with other agents.

However, the mechanisms presented in this thesis provide support for reliable mobile agent execution for the most important application classes, allowing the use of mobile agent technology even in critical application areas like electronic commerce.

Inhaltsverzeichnis

Danksagung ... 3
Kurzfassung ... 5
Extended english abstract of the dissertation ... 7
Inhaltsverzeichnis ... 19
Abkürzungsverzeichnis ... 23

1 Einführung ... 25
1.1 Motivation ... 25
1.2 Wissenschaftlicher Beitrag ... 27
1.3 Aufbau der Arbeit ... 29

2 Mobile Agenten ... 31
2.1 Einführung in mobile Agenten ... 31
2.1.1 Mobile-Agenten-Technologie ... 31
2.1.1.1 Der Platz - eine Abstraktion für Rechner ... 32
2.1.1.2 Agenten-Programmiersprache und -Code ... 33
2.1.1.3 Migration und Reiserouten ... 33
2.1.1.4 Kommunikation und Ressourcen-/Dienstzugriff ... 34
2.1.1.5 Kontrollmechanismen ... 34
2.1.1.6 Sicherheit ... 35
2.1.1.7 Fehlertoleranz ... 35
2.1.2 Stärken ... 36
2.1.3 Herausforderungen ... 37
2.2 Agentenmodell ... 38

3 Reiserouten ... 41
3.1 Ein einfaches Reiseroutenkonzept ... 42
3.2 Ein flexibleres Reiseroutenkonzept ... 44
3.2.1 Die flache Reiseroute ... 45
3.2.2 Geschachtelte Reiserouten ... 50
3.2.3 Beispiel einer komplexen Reiseroute ... 56
3.2.4 Diskussion ... 59

4 Genau-einmal Ausführung ... 61
4.1 System- und Fehlermodell ... 62
4.1.1 Fehlerklassifikation ... 62
4.1.2 Verwendetes Systemmodell ... 63
4.1.3 Verwendetes Fehlermodell ... 64
4.2 Definition “Genau-einmal Ausführung” ... 64

4.3 Basisprotokoll 65
4.4 Blockierungsfreies Protokoll 69
4.4.1 Überblick über das Protokoll 69
4.4.2 Votierprotokoll 75
4.4.2.1 Integration in das 2-Phasen-Commit-Protokoll 76
4.4.2.2 Stabile Zustände des Protokolles 77
4.4.2.3 Phasen des Votierprotokolles 78
4.4.2.4 Fehlerbehandlung 85
4.4.2.5 Adaption an Fehlermodell mit Nachrichtenüberholung 91
4.4.2.6 Alternativen zum Mehrheitsentscheid 91
4.4.3 Beobachtungs- und Auswahlprotokoll 91
4.4.3.1 Beobachtungsprotokoll 92
4.4.3.2 Auswahlprotokoll 95
4.4.4 Korrektheit 97
4.4.4.1 Korrektheit eines Protokolles 98
4.4.4.2 Informaler Korrektheitsbeweis 98
4.5 Kommunikationsaufwand und Stufenkonstruktion 109
4.5.1 Kommunikationsaufwand der Protokolle 109
4.5.2 Möglichkeiten zur Reduktion des Kommunikationsaufwandes 115
4.5.3 Algorithmus zur Stufenkonstruktion 117
4.6 Analytische Bewertung der Fehlertoleranz 122
4.6.1 Markov-Modelle 123
4.6.2 Einschränkung des Fehlermodells 128
4.6.3 Verfügbarkeit eines Knotens 128
4.6.4 Systemverfügbarkeit und Blockierwahrscheinlichkeit 129
4.6.4.1 Basisprotokoll 130
4.6.4.2 Blockierungsfreies Protokoll 130
4.6.5 Verweildauer in einer Stufe 133
4.6.5.1 Basisprotokoll 134
4.6.5.2 Blockierungsfreies Protokoll 141
4.6.5.3 Vergleich der Protokolle 152
4.7 Leistungsmessungen 155
4.7.1 Protokollimplementation 155
4.7.2 Messungen 157
4.8 Verwandte Arbeiten 160
4.8.1 Bereiche Transaktionsverarbeitung und Fehlertoleranz 160
4.8.2 Mobile Agenten 164
4.9 Diskussion 167

5 Partielles Rücksetzen 171
5.1 Problemstellung 172
5.2 Kompensation 174
5.3 Erweiterung des Agentenmodells 176
5.4 Basismechanismus 184
5.4.1 Überblick 184
5.4.2 Logging 186
5.4.3 Algorithmus 189
5.4.4 Integration in den blockierungsfreien Mechanismus 193
5.5 Optimierungen 194
5.5.1 Vermeidung unnötiger Agententransporte 195
5.5.1.1 Typen von Operationseinträgen 195
5.5.1.2 Möglichkeiten der Optimierung 199
5.5.1.3 Algorithmus 205
5.5.2 Reduzierung der Größe des Rücksetz-Logs 210
5.6 Verwaltung von Rücksetzpunkten 211
5.7 Verwandte Arbeiten 216
5.8 Diskussion 217

6 Resümee 219
6.1 Zusammenfassung 219
6.2 Allgemeinheit der Ergebnisse 221
6.3 Ausblick 221

Literaturverzeichnis 223

Abkürzungsverzeichnis

2PC-Protokoll	2-Phasen-Commit-Protokoll
ACID	Eigenschaften von Transaktionen: Atomizität, Konsistenz (Consistency) Isolation und Dauerhaftigkeit
bzw.	beziehungsweise
CD	Compact Disc
d.h.	das heißt
engl.	englisch
GSM	Global System for Mobile communications
i.a.	im allgemeinen
MASIF	OMG Mobile Agent System Interoperability Facility
MTTF	Mean Time To Failure - mittlere Zeit bis zum Ausfall
MTTR	Mean Time To Repair - mittlere Zeit bis zur Reparatur
OMG	Object Management Group
PDA	Personal Digital Assistant
RPC	Remote Procedure Call (Entfernter Funktionsaufruf)
RM	Ressourcenmanager
TM	Transaktionsmanager
URL	Uniform Resource Locator
vgl.	vergleiche
Wdh.	Wiederholung
WWW	World Wide Web
z.B.	zum Beispiel

Kapitel 1

Einführung

1.1 Motivation

Die momentane, sehr stürmisch verlaufende Entwicklung der Informationsgesellschaft geht einher mit einer explosionsartigen Entwicklung im Bereich der Rechnernetze. Die Anzahl der Nutzer von Netzwerken – vor allem des Internets – und damit gekoppelt die Größe und der Durchsatz der Netzwerke nehmen rasant zu. Neue Entwicklungen im Bereich der Unterhaltungselektronik ermöglichen den einfachen Zugang zum Internet (z.B. mittels Webpads) oder bieten die Möglichkeit des mobilen, momentan vor allem GSM-gestützten Zugangs zum Netz. Zudem nimmt die Vielfalt der angebotenen Dienste ständig zu.

Durch diese Entwicklung ergeben sich eine ganze Menge an Problemen und neuen Herausforderungen, von denen hier nur einige exemplarisch aufgelistet sind: Trotz des stetigen Wachstums der Netzwerkkapazitäten sind die Netzwerke durch die weitaus stärker ansteigende Zahl an Benutzern chronisch überlastet. Die Verwendung mobiler Endgeräte ist durch die zur Zeit vorherrschende synchrone Online-Nutzung von Diensten aufgrund der geringen Übertragungsraten und der instabilen Verbindungen nur sehr eingeschränkt möglich. Obwohl das Internet gerne als das Netz der unbegrenzten Möglichkeiten dargestellt wird, ist es für den durchschnittlichen Internet-Benutzer mangels permanenter Anbindung seines Rechners jedoch nur sehr eingeschränkt möglich, selbst neue, eigene Dienste im Internet bereitzustellen.

Eine in den letzten Jahren aufkommende Technologie zur Lösung dieser Probleme und Herausforderungen ist die der mobilen Agenten. Mobile Agenten sind Programme, die autonom im Auftrag ihres Besitzers einen Auftrag ausführen. Charakteristisch für mobile Agenten ist hierbei, daß sie sich zur Erfüllung ihres Auftrages in einem Netzwerk von Rechnern bewegen, d.h. die Programmausführung von einem Rechner auf einen anderen verlagern können (Migration). Dies ermöglicht ihnen einerseits, auf einfache Weise auf Ressourcen aller Rechner im Netzwerk (lokal) zuzugreifen, und andererseits, mittels kooperativer Bearbeitung eines Problems durch mehrere mobile Agenten eine Aufgabe effizienter zu erfüllen. Hat beispielsweise

ein mobiler Agent die Aufgabe, auf beliebigen Rechnern liegende Datenquellen (z.B. WWW-Server) nach bestimmten Kriterien zu durchsuchen, so kann er zu diesen Rechnern migrieren, die Datenquellen lokal untersuchen und die Ergebnisse zum Benutzer zurücksenden, anstatt sich die Daten, z.B. die einzelnen Seiten der WWW-Servers, über das Netzwerk zu laden, um sie zu untersuchen. Eine Untersuchung von STRASSER UND SCHWEHM (1997) und Ergebnisse von THEILMANN (2000) zeigen, daß sich abhängig von der zu untersuchenden Datenmenge, der Größe des zu übermittelnden Resultats und dem für die Migration des Agenten notwendigen Aufwandes durch dieses Vorgehen erhebliche Einsparungen sowohl in der Netzwerkbelastung als auch in der zur Ausführung notwendigen Zeit ergeben können. Sowohl diese mögliche Reduzierung der Netzwerkbelastung als auch die Möglichkeit der mobilen Agenten, ihre Aufgabe ohne Kontakt zum Benutzer asynchron weiter zu bearbeiten, machen mobile Agenten auch für Anwendungen im Bereich mobiler Endgeräte äußerst attraktiv.

Neben der schon erwähnten globalen Suche und Filterung von Datenbeständen gibt es für mobile Agenten noch viele andere Anwendungsgebiete. Beispiele sind Anwendungen in den Bereichen des elektronischen Handels, des Netzwerk-Managements, der Informationsverteilung oder des Parallelrechnens. Dies sind nur einige der Anwendungsbereiche, welche von vielen Autoren in Veröffentlichungen genannt werden, z.B. in MINSKY ET AL. (1996), FÜNFROCKEN UND MATTERN (1999), LANGE UND OSHIMA (1999) und WONG ET AL. (1999).

Obwohl die Verwendung mobiler Agenten in den genannten Gebieten sehr attraktiv erscheint, wird die Technologie bisher jedoch nur sehr wenig eingesetzt. Zwar sind eine ganze Anzahl von Plattformen für mobile Agenten sowohl als Forschungsprototypen (z.B. Ara (PEINE UND STOLPMANN (1997)), D'Agents (GRAY ET AL. (1998)), Mole (BAUMANN ET AL. (1998A))) als auch als kommerzielle Produkte (z.B. Aglets (LANGE UND OSHIMA (1998)), Grashopper (GRASSHOPPER (2000)), Concordia (WALSH, PACIOREK UND WONG (1999))) verfügbar, die Anzahl der darauf realisierten und tatsächlich verwendeten Anwendungen ist jedoch sehr gering. Dies liegt unter anderem daran, daß einige der für die kommerzielle Verwendung mobiler Agenten essentiellen Problemstellungen – Kontrollmechanismen, Sicherheit und Fehlertoleranz – zum gegenwärtigen Zeitpunkt noch Gegenstand der Forschung sind. Der Bereich der Kontrollmechanismen beschäftigt sich vor allem mit den Problemen des Findens und Terminierens von mobilen Agenten und der Waisenerkennung. BAUMANN (2000) gibt einen Überblick über existierende Ansätze und schlägt weitere Lösungen für diese Problemstellungen vor. Der Bereich der Sicherheit in Agentensystemen beschäftigt sich vor allem mit dem Schutz des Rechners und mobiler Agenten gegen den Angriff durch böswillige mobile Agenten und dem Schutz mobiler Agenten gegen den Angriff durch böswillige Ausführungsumgebungen. Hierbei unterscheidet sich der Schutz des Rechners und mobiler Agenten gegen Angriffe durch böswillige Agenten nur marginal von der allgemeinen Sicherheitsproblematik in Verteilten Systemen. Der Schutz eines mobilen Agenten und damit letztendlich eines Programmes gegen eine böswillige Ausführungsumgebung ist Neuland und wird unter anderem in SANDER UND TSCHUDIN (1998) und HOHL (2001) untersucht.

Die vorliegende Arbeit beschäftigt sich mit der Fehlertoleranz der Ausführung mobiler Agenten. Für viele der oben genannten Anwendungsbereiche ist eine fehlertolerante Ausführung mobiler Agenten eine notwendige Voraussetzung für den Einsatz dieser Technologie. Betrachten wir dazu das folgende Szenario aus dem Bereich des elektronischen Handels: Unser persönlicher Hilfsassistent, welcher als mobiler Agent realisiert ist, hat die Aufgabe, eine Geschäftsreise zu organisieren. Hierfür muß er einen Flug buchen, ein Zimmer reservieren und bei einer Autovermietung einen Mietwagen ordern. Nachdem er alle notwendigen Informationen vom Benutzer erhalten hat, migriert er zuerst vom PDA (Personal Digital Assistant) des Benutzers zum Server der Fluggesellschaft und bucht dort den Flug. Mit den nun feststehenden genauen Reisedaten kann er das Zimmer und den Mietwagen in einer beliebigen Reihenfolge auf den jeweils zuständigen Servern buchen. Das Ergebnis legt er entweder im Briefkasten des Benutzers ab oder präsentiert es, nachdem er auf den PDA des Benutzers zurückmigriert ist. Ein Problem hierbei ist, daß der Benutzer den Fortschritt des Agenten bei der Ausführung dieser Aufgabe nicht direkt beobachten kann und deshalb selbst nicht bemerkt, wenn der Agent durch Systemfehler verloren geht oder in seiner Ausführung blockiert wird. Es ist beispielsweise nicht akzeptabel, wenn zwar ein Flug und ein Auto gebucht werden, die Hotelreservierung jedoch unterbleibt. Ebenso inakzeptabel ist jedoch, daß eine der Teilaufgaben durch Systemfehler der beteiligten Rechner oder des Netzwerkes mehrfach ausgeführt wird, z.B. daß zwei Flüge gebucht werden. Es muß also sichergestellt sein, daß alle (Teil-)Aufgaben des Agenten *genau einmal* (engl.: *exactly once*) ausgeführt werden. Zudem muß sichergestellt werden, daß der Agent seine Aufgabe so schnell wie möglich ausführt und nicht durch Systemfehler unnötig lange blockiert werden kann. Fällt wegen eines Systemfehlers ein Rechner aus, dann sollte jeder zum Zeitpunkt des Absturzes auf diesem Rechner ausgeführte Agent möglichst schnell automatisch auf einem anderen Rechner – für den Agenten möglichst transparent – neu gestartet werden. Auch in diesem Falle sollte sichergestellt werden, daß die Aufgaben des Agent genau einmal ausgeführt werden. Bisher erhältliche Ausführungsplattformen für mobile Agenten bieten in diesem Sinne nur wenig oder keine Fehlertoleranz.

1.2 Wissenschaftlicher Beitrag

In Veröffentlichungen vorgeschlagene Ansätze zur fehlertoleranten Ausführung von mobilen Agenten behandeln meist nur Teilaspekte. Während alle Ansätze gewährleisten, daß ein Agent durch Rechner- oder Netzwerkausfälle nicht verloren gehen kann und viele sicherstellen, daß ein Agent bei Ausfall eines Rechners nicht bis zum Neustart des Rechners blockiert wird, stellen die wenigsten Ansätze sicher, daß die Aufgaben des Agenten unter allen Umständen genau einmal ausgeführt werden. Darüberhinaus basieren die vorgeschlagenen Ansätze überwiegend auf Fehlermodellen, in denen keine Netzwerkpartitionierungen auftreten können. Der einzige Ansatz, der alle diese Aspekte abdeckt stammt von SCHNEIDER (1997A). Dieser Ansatz bietet nicht nur Schutz gegen Ausfälle von Rechnern oder dem Netzwerk, sondern darüber hinaus auch

Schutz gegen byzantinische Fehler, d.h. gegen Fehler, bei denen sich die defekten Komponenten zufällig verhalten. Hierdurch bedingt ist der Mehraufwand für diesen Ansatz sehr hoch und die Voraussetzungen, um diesen Ansatz nutzen zu können, nur in seltenen Anwendungsfällen gegeben. Existierende Ansätze aus dem Gebiet der Verteilten Systeme sind aufgrund der Mobilität und Autonomie der mobilen Agenten nicht ohne jedes weitere übertragbar und behandeln darüber hinaus auch nicht alle der oben aufgeführten Teilaspekte. Einen Überblick über diese Arbeiten gibt Abschnitt 4.8.

Die vorliegende Arbeit entwickelt einen Mechanismus, der sicherstellt, daß die Aufgaben eines Agenten genau einmal ausgeführt werden, d.h. der Agent geht unter keinen Umständen durch Rechner- oder Netzwerkausfall verloren und führt jede seiner Aufgaben einmal (und nicht öfter) durch. Darüberhinaus wird sichergestellt, daß der Agent durch Rechner- oder Netzwerkausfälle nicht blockiert werden kann. Im Fehlermodell des Netzwerkes sind hierbei Netzwerkpartitionierungen enthalten. Wesentliche Ziele bei der Entwicklung des Mechanismus sind einerseits eine möglichst breite Schicht von Anwendungsszenarien durch den Mechanismus zu unterstützen, aber auch andererseits möglichst wenig Mehraufwand bei der fehlertoleranten Ausführung der Agenten zu verursachen und den Mechanismus für den Agentenprogrammierer möglichst transparent zu gestalten.

Voraussetzung der effizienten Anwendung dieses Mechanismus ist es, bei jeder Migration des Agenten mehrere mögliche Migrationsziele – z.B. alternative Rechner für dieselbe Teilaufgabe oder alternative Teilaufgaben auf verschiedenen Rechnern – zur Auswahl zu haben. Die wenigen, für mobile Agenten existierenden Reiseroutenkonzepte werden diesem Anspruch nur sehr eingeschränkt gerecht. Daher wird in dieser Arbeit zuerst ein Reiseroutenkonzept entwickelt, welches es mit einfachen Mitteln erlaubt, die einer Anwendung inhärenten Alternativen zu spezifizieren, welche in der Auswahl eines Rechners zur Ausführung einer Teilaufgabe und in der Ausführungsreihenfolge der Teilaufgaben bestehen.

Schließlich wird aufbauend auf den Mechanismus zur fehlertoleranten Ausführung mobiler Agenten ein Mechanismus entwickelt, welcher das *partielle Rücksetzen* (*partial rollback*) der Ausführung von Agenten ermöglicht. Da der fehlertolerante Ausführungsmechanismus auf einer Aneinanderreihung von Transaktionen zur Ausführung der Teilaufgaben eines Agenten beruht, ist es notwendig, diese Transaktionen beim partiellen Rücksetzen des Agenten zu kompensieren. Im Gegensatz zu gängigen Mechanismen, bei denen der Programmierer einer Anwendung für jede Transaktion eine Kompensationstransaktion zur Verfügung stellen muß, ermöglicht es der in dieser Arbeit vorgestellte Ansatz, daß ein Anwendungsprogrammierer für bestimmte Anwendungsklassen keine Kompensationstransaktionen zur Verfügung stellen muß.

1.3 Aufbau der Arbeit

Kapitel 2 führt detailliert in die Thematik der mobilen Agenten ein. Hierbei wird der Raum der Interpretationsmöglichkeiten des Begriffes "mobiler Agent" aufgespannt, die Anwendungsmöglichkeiten mobiler Agenten skizziert und die in dieser Arbeit verwendete Definition des Begriffes "mobiler Agent" eingeführt.

Das darauffolgende Kapitel beschäftigt sich eingehend mit Mechanismen zur Spezifikation flexibler Reiserouten, deren Flexibilität von den in den folgenden Kapiteln entwickelten Mechanismen zu Optimierungszwecken verwendet werden kann.

In Kapitel 4 werden Mechanismen zur fehlertoleranten Ausführung mobiler Agenten entwickelt. Die Korrektheit der entwickelten Mechanismen wird nachgewiesen und der Nachrichtenmehraufwand durch Optimierungen verringert. Anschließend werden noch eine analytische Bewertung der durch die entwickelten Mechanismen eingeführten Verfügbarkeit und Messungen der durch die Mechanismen eingeführten Zeitkomplexität durchgeführt.

Kapitel 5 beschäftigt sich mit Systemmechanismen zum partiellen Rücksetzen der Ausführung eines Agenten. Nach einer Analyse der Problemstellung werden Kompensationsoperationen und deren Einschränkungen diskutiert, bevor das in Kapitel 2 eingeführte Agentenmodell um das partielle Rücksetzen erweitert wird. Dieses Modell dient dann als Grundlage zur Entwicklung eines Basismechanismus zum partiellen Rücksetzen und von Optimierungen dieses Mechanismus.

Kapitel 6 schließt die Arbeit mit einer Zusammenfassung, einer Diskussion der Übertragbarkeit der erzielten Ergebnisse auf andere Gebiete und einem Ausblick.

Kapitel 2

Mobile Agenten

Bedingt durch die sich exponentiell entwickelnde weltweite Vernetzung, den dabei vermehrt auftretenden nicht oder nur unbefriedigend gelösten Problemen, erfährt die Technologie der mobilen Agenten dank ihres Potentials zur Lösung zumindest einiger dieser drängenden Probleme in den letzten Jahren zunehmende Aufmerksamkeit – sowohl durch die akademische als auch die industrielle Forschung. Im Mittelpunkt steht hierbei vor allem die Entwicklung und die Standardisierung jener Infrastruktur (eine Middleware), welche für den Einsatz der Technologie mobiler Agenten notwendig ist.

Im ersten Teil dieses Kapitel wird der Frage nachgegangen, was mobile Agenten überhaupt sind und welche Funktionalität eine Infrastruktur für mobile Agenten zur Verfügung stellen muß. Die Antwort hierauf wird keine eindeutige, scharfe Definition sein. Es wird vielmehr ein Raum aufspannt innerhalb dessen die meisten der heute existierenden Ansätze eingeordnet werden können. Ferner wird darauf eingegangen, welche Probleme durch die Verwendung mobiler Agenten gelöst – aber auch aufgeworfen – werden. Bei der Diskussion der Stärken der Mobile-Agenten-Technologie ergeben sich exemplarisch einige der möglichen Anwendungsbereiche dieser Technologie. Der zweite Teil des Kapitels führt das in dieser Arbeit verwendete Mobile-Agenten-Modell ein.

2.1 Einführung in mobile Agenten

2.1.1 Mobile-Agenten-Technologie

Mobile Agenten sind Programme, die im Auftrag ihres Besitzers autonom einen Auftrag ausführen. Hierzu können sie sich frei in einem Netzwerk von Rechnern bewegen, die auf den Rechnern verfügbaren Ressourcen und Dienste (lokal) in Anspruch nehmen und mit anderen (mobilen) Agenten kommunizieren (WHITE (1997)). Die Idee, Programme inklusive Daten über ein Netzwerk auf andere Rechner zu versenden und dort auszuführen, ist als solche nicht sehr

neu – sie läßt sich bis in die Anfänge des Internets zurückverfolgen (vgl. BAUMANN (1999) für einen Überblick). Die Idee der mobilen Agenten unterscheidet sich hiervon jedoch vor allem dadurch, daß die autonome Fortbewegung eines Agenten über potentiell viele Rechner ohne die Notwendigkeit des Kontaktes zum Ursprungsrechner des Agenten grundlegender Bestandteil des Programmierparadigmas ist.

Zur Realisierung der genannten Eigenschaften mobiler Agenten wird eine Infrastruktur benötigt. Neben einer Ausführungsumgebung für Agenten stellt sie unter anderem die zur Fortbewegung – auch *Migration* genannt – und zur Kommunikation notwendigen Mechanismen zur Verfügung. Diese Infrastruktur bezeichnet man als *Mobile-Agenten-Plattform* (engl.: *mobile agent platform*), oft auch als *Mobile-Agenten-System* (engl.: *mobile agent system*). Da der Begriff Mobile-Agenten-System allerdings auch häufig zur Bezeichnung einer Menge auf Rechnern installierter Mobile-Agenten-Plattformen inklusive der sich in dem System aufhaltenden mobilen Agenten verwendet wird, wird in dieser Arbeit der Begriff Agentenplattform zur Bezeichnung der auf einem Rechner vorhandenen Infrastruktur verwendet. Da jede im Bereich der mobilen Agenten arbeitende Forschergruppe ihre eigenen Schwerpunkte setzt, existieren annähernd soviel verschiedene inkompatible Implementierungen (bzw. Typen) von Agentenplattformen wie Forschergruppen. Bei den in den nächsten Abschnitten folgenden Beschreibung der Eigenschaften und Funktionalitäten von Agentenplattformen werden einzelne Implementierungen von Plattformen exemplarisch aufgelistet. Einen umfassenderen Überblick über existierende Plattformen bietet HOHL (2000).

2.1.1.1 Der Platz - eine Abstraktion für Rechner

Eine in Agentenplattformen häufig anzutreffende Abstraktion ist die des *Platzes* (engl.: *place*), manchmal auch mit anderen Namen wie z.B. *Ort* (engl.: *location*) bezeichnet. Ein Platz befindet sich auf einem (physikalischen) Rechner und bietet eine Ausführungsumgebung für mobile Agenten. Ein Agent bewegt sich in diesem Fall nicht mehr von Rechner zu Rechner, sondern von Platz zu Platz. Auf einem Rechner können sich mehrere Plätze befinden. Dies ermöglicht beispielsweise, daß verschiedene Autoritäten (Benutzer, Organisationen, ...) voneinander unabhängig auf einem Rechner Ausführungsumgebungen anbieten, welche den Agenten sehr unterschiedliche Ressourcen und Dienste zur Verfügung stellen. Stellt eine Agentenplattform eine Platz-Abstraktion zur Verfügung, so muß sie ein (eindeutiges) Adressierungskonzept für Plätze zur Verfügung stellen. Agentenplattformen, die eine Platz-Abstraktion anbieten, sind unter anderem Ara (PEINE UND STOLPMANN (1997)), Mole (BAUMANN ET AL. (1998B)) und Telescript (BRADSHAW (1997)), hingegen stellt zum Beispiel Tacoma (JOHANSEN, VAN RENESSE UND SCHNEIDER (1995)) diese Abstraktion nicht zur Verfügung.

2.1.1.2 Agenten-Programmiersprache und -Code

Weitere Merkmale, in denen sich die verschiedenen Agentenplattformen unterscheiden, sind einerseits die zur Programmierung der Agenten unterstützten Programmiersprachen und andererseits die Art, in welcher Form der Code des Agenten beim Transport über das Netzwerk vorliegt und wie der Code ausgeführt wird. Generell kann ein Agent entweder von einem Interpreter oder direkt als Maschinencode ausgeführt werden. Im ersten Fall liegt der Code des Agenten entweder im Quellcode (z.B. TCL (OUSTERHOUT (1994))) oder aber in einem rechnerunabhängigen Zwischencode (z.B. Java Byte-Code, vgl. auch ARNOLD UND GOSLING (1997)) vor. Wird der Agent direkt als Maschinencode ausgeführt, dann wird der Agent entweder schon als übersetztes Programm verschickt oder vor der Ausführung vom ausführenden Rechner übersetzt. Unter anderem aus Gründen der Portabilität verwenden die meisten der heute existierenden Agentenplattformen eine interpretierte Sprache als Programmiersprache für Agenten. Hier wird vor allem Java verwendet (Mole (STRASSER, BAUMANN UND HOHL (1996)) und Aglets (LANGE UND OSHIMA (1998))), aber auch TCL (D'Agents (GRAY ET AL. (1998)) und Ara (PEINE UND STOLPMANN (1997))) und Python (HYLTON ET AL. (1996)) finden Verwendung. Nur wenige Plattformen bieten die Möglichkeit, unter mehreren Programmiersprachen zu wählen (z.B. TACOMA (JOHANSEN, VAN RENESSE UND SCHNEIDER (1995)), Ara und D'Agents).

2.1.1.3 Migration und Reiserouten

Der Begriff der Migration beschreibt die Fortbewegung eines Agenten von einer Ausgangsausführungsumgebung zu einer Zielausführungsumgebung. Der Agent wird bei der Migration aus der Ausgangsumgebung entfernt, zur Zielumgebung transportiert und dort gestartet. Hierbei muß von der Agenten-Plattform neben dem Code des Agenten auch dessen Zustand zur Zielumgebung übertragen werden. CUGOLA ET AL. (1996) unterscheiden hier zwischen *schwacher Migration* (engl.: *weak migration*) und *starker Migration* (engl.: *strong migration*). Bei der schwachen Migration wird zusätzlich zum Code nur der Zustand der globalen Variablen übertragen; lokale Variablen, der Stapel (engl.: stack) und Programmzähler werden nicht übertragen. Der Agent wird dann in der Zielumgebung entweder mit einer ausgezeichneten Start-Prozedur (bzw. Methode) gestartet oder es wird beim Migrationsbefehl eine Prozedur/Methode angegeben, mit welcher der Agent gestartet werden soll. Bei der starken Migration wird der komplette Zustand des Agenten übertragen. Nach der Ankunft bei der Zielumgebung setzt der Agent dann seine Ausführung direkt nach dem Migrationsbefehl fort. Agentenplattformen mit starker Migration sind unter anderem Ara und Telescript. Aufgrund der schwierigeren Implementation und semantischer Ungereimtheiten bei starker Migration (z.B. Zugriff auf lokale Ressourcen, offene Netzwerkverbindungen) stellen die meisten der heute verfügbaren Agentenplattformen nur schwache Migration zur Verfügung.

Um das Ziel einer Migration anzugeben, existieren in den heutigen Systemen zwei unterschiedliche generelle Ansätze. Beim ersten Ansatz wird das Ziel beim Migrationsbefehl explizit als

Parameter im Programmcode angegeben. Da dieser Parameter eine Variable sein kann, ist hiermit die dynamische Wahl des nächsten Zieles während der Ausführung des Agenten gewährleistet. Der zweite Ansatz wird der Tatsache gerecht, daß es bei den meisten Mobile-Agenten-Anwendungen einen oder auch mehrere Zeitpunkte gibt, zu denen zumindest für die nähere Zukunft festgelegt wird, welche Plätze vom Agent besucht werden sollen. Diese Planung kann dann in eine *Reiseroute* eingetragen werden, welche in Agentenplattformen mit diesem Ansatz essentieller Bestandteil eines Agenten ist. Der Migrationsbefehl hat in diesem Fall keinen Ziel-Parameter sondern entnimmt das Ziel der Reiseroute des Agenten. Die Reiseroute kann bei den meisten Systemen dynamisch beliebig geändert und erweitert werden. Die mögliche Komplexität der Reiseroute schwankt, abhängig von der Agentenplattform, zwischen einer einfachen linearen Liste von Plätzen, die in der angegebenen Reihenfolge besucht werden (zum Beispiel in Concordia, vgl. WONG ET AL. (1997)), bis zu sehr komplexen Spezifikationen. Kapitel 3 stellt ein solches komplexes Reiseroutenkonzept vor. Vorteil des Reiseroutenansatzes ist, daß die Reiseroute für das System sichtbar wird. Läßt die Reiseroute Spielräume hinsichtlich der Abfolge der zu besuchenden Plätze, kann das System dies beispielweise dazu nutzen, um die Abfolge der besuchten Plätze im Rahmen des gegebenen Spielraumes (z.B. nach der Zeit) zu optimieren.

2.1.1.4 Kommunikation und Ressourcen-/Dienstzugriff

Essentielle Bestandteile des Mobile-Agenten-Paradigmas, und daher von allen Agentenplattformen angeboten, sind die lokale Kommunikation zwischen Agenten und die Nutzung lokaler Ressourcen bzw. Dienste. Mit dem Argument, daß globale Kommunikation durch Migration (mit anschließender lokaler Kommunikation) ersetzt werden kann, verzichten einige Plattformen (z.B. Ara, Messenger Environment M0 (TSCHUDIN (1997))) komplett auf globale Kommunikation. Die anderen Plattformen bieten aus Effizienzgründen globale Kommunikation zwischen Agenten und teilweise auch globalen Ressourcenzugriff an.

Gängige Kommunikationsmechanismen für lokale und globale Kommunikation zwischen Agenten sind Nachrichtenaustausch und Prozedur- bzw. Methodenaufrufe (ähnlich einem RPC). Damit bei diesen Kommunikationsmechanismen der Empfänger einer Nachricht oder eines Methodenaufrufes spezifiziert werden kann, muß die Agentenplattform ein Namens- und Adressierungskonzept bereitstellen. Seltener, und dann meist auch nur im Falle der lokalen Kommunikation wird die anonyme Kommunikation über gemeinsamen Speicher verwendet. CABRI, LEONARDI UND ZAMBONELLI (1998) beschreiben mit den TupleSpaces ein solches Konzept der anonymen Kommunikation. Einen allgemeineren Überblick über Kommunikationsmechanismen bieten BAUMANN ET AL. (1997).

2.1.1.5 Kontrollmechanismen

Neben den in den meisten Agentenplattformen vorhandenen Mechanismen zum Starten von Agenten (entweder durch einen Benutzer oder einen Agent) und zum Klonen von Agenten gibt

es Bedarf an einer ganzen Reihe weiterer Kontrollmechanismen, die aber nur in sehr wenigen Systemen angeboten werden. Funktioniert das Beenden (die *Terminierung*) eines lokal auf einem Platz sich befindlichen Agenten noch bei den meisten Systemen, wird nur bei sehr wenigen Systemen die Terminierung eines Agenten durch seinen Besitzer auf einem beliebigem (evtl. unbekanntem) Platz unterstützt. Beispiele für Systeme mit globaler Terminierung sind Mole, die Aglets Workbench und den MASIF-Standard (MILOJICIC ET AL. (1998)) erfüllende Systeme. Mechanismen zum Auffinden des momentanen Aufenthaltsortes eines Agenten bieten auch nur die wenigsten Plattformen (Beispiele hierfür sind ebenfalls die bei der Terminierung genannten Systeme). Folge hiervon ist, daß bei den meisten Systemen, die globale Kommunikation anbieten, stets der momentane Aufenthaltsort des Agenten, mit dem man kommunizieren möchte, angegeben werden muß (die Anwendung hat dafür zu sorgen, daß der Aufenthaltsort bekannt ist). Einen Mechanismus zur *Waisenerkennung* (engl.: *orphan detection*) stellt momentan nur Mole zur Verfügung. Einen ausführlichen Überblick über den Stand im Gebiet der Kontrollmechanismen für mobile Agenten und die Vorstellung der in Mole implementierten Kontrollmechanismen bietet BAUMANN (1999).

2.1.1.6 Sicherheit

Dem Thema Sicherheit kommt im Bereich der mobilen Agenten besondere Bedeutung zu. Da ein Agent seinen Code bei der Migration mitbringt, muß sichergestellt sein, daß dieser anderen Agenten und dem ausführenden System (und von dort aus anderen Systemen) keinen Schaden zufügt. Diese Problematik unterscheidet sich jedoch nicht wesentlich von der Sicherheit in verteilten Systemen und *Mobile-Code-Systemen* (z.B. Java Applets), wodurch entsprechende Lösungen angepaßt werden können. Ein völlig neuer Aspekt besteht jedoch darin, daß bei mobilen Agenten die Ausführungsumgebung (bzw. deren Besitzer) eines Agenten Interesse daran haben könnte, die Ausführung des Agenten in irgendeiner Weise zu sabotieren oder die im Agent vorhandenen Informationen zu lesen. Sucht zum Beispiel ein Einkaufsagent ein möglichst günstiges Angebot für einen einzukaufenden Artikel, könnte ein “virtueller Laden” den Agent durch Manipulation seiner Ausführung davon überzeugen, daß dieser Laden der billigste ist. Trägt der Agent elektronisches Geld (CHAUM (1985)) mit sich, dann kann einfach eine Kopie der in ihm enthaltenen elektronischen Münzen gemacht werden. Sicherheit in Agentenplattformen im allgemeinen und verschiedene Ansätze zum Schutz des Agenten vor feindlichen Ausführungsumgebungen findet man in HOHL (2001).

2.1.1.7 Fehlertoleranz

Die fehlertolerante Ausführung der mobilen Agenten ist für viele potentielle Anwendungen die Voraussetzung, um überhaupt Mobile-Agenten-Technologie einsetzen zu können. Bewegt sich ein Agent durch ein Netzwerk, muß sichergestellt sein, daß auf keinen Fall die Ausführung des Agenten (unbemerkt) durch einen Rechnerausfall bzw. -fehler oder einen Netzwerkfehler abge-

brochen wird und dabei der Agent, d.h. sein kompletter Zustand, zerstört wird. Zusätzlich sollte verhindert werden, daß die Ausführung eines Agenten durch einen Systemfehler bis zur Behebung des Fehlers blockiert werden kann. Mechanismen zur Sicherung der fehlertoleranten Ausführung mobiler Agenten gibt es bisher nur ansatzweise in einigen wenigen Agentenplattformen. Einen Überblick über die bisher existierenden Ansätze bietet Abschnitt 4.8. Der Rest dieser Arbeit wird sich intensiv mit dieser Problemstellung auseinandersetzen.

2.1.2 Stärken

Die Stärken der Technologie der mobilen Agenten sind sehr vielfältig. CHESS ET AL. (1997) bieten eine ausführliche Diskussion der positiven Eigenschaften der mobilen Agenten und stellen fest, daß zwar jede einzelne dieser Eigenschaften auch mittels anderer Technologien erreicht werden kann, die Synergieeffekte durch die Vereinigung aller dieser Eigenschaften in der Technologie der mobilen Agenten den Einsatz dieser Technologie allerdings sehr attraktiv machen. Hier an dieser Stelle seien nur die wichtigsten Vorteile kurz erwähnt:

Optimierung von Kommunikationskosten. Unter gewissen Voraussetzungen erlaubt die Anwendung mobiler Agenten eine wesentliche Reduktion von Kommunikationskosten. Empfängt ein Agent sehr viele Daten von einer entfernten Ressource und produziert daraus ein vergleichsweise kleines Ergebnis, dann kann es im Sinne der über das Netzwerk zu übertragenden Datenmenge günstiger sein, daß der Agent zur Ressource migriert, die Daten dort lokal verarbeitet und das Ergebnis zurückschickt. Dies ist dann der Fall, wenn die Summe der für die Migration und das Ergebnis zu übertragenden Datenmenge kleiner ist als die von der Ressource zum Agent übertragene Menge an Daten. Hat ein Agent sehr viele Kommunikationsschritte mit einer entfernten Ressource oder einem anderen Agenten durchzuführen, dann kann es im Sinne der für die Kommunikation benötigten Zeit günstiger sein, zu der Ressource oder zu dem Agenten zu migrieren um die Kommunikationsverzögerungen einzusparen, obwohl dadurch bei einem sehr großen Agent eventuell mehr Daten über das Netzwerk transportiert werden. Eine von vielen Autoren zitierte Anwendung, welche von diesen Optimierungen profitieren kann, ist die verteilte Informationssuche. THEILMANN UND ROTHERMEL (1999) und THEILMANN (2000) skizzieren den Einsatz mobiler Agenten zur Realisierung effizienter, intelligenter Suchmaschinen. Vorteile zeitlicher Natur ergeben sich natürlich auch, wenn der Agent schnell auf Ereignisse reagieren soll. Soll beispielsweise ein Agent Aktienkurse überwachen und bei Kursschwankungen entweder neue Aktien kaufen oder alte Aktien abstoßen, dann kann er auf dem Rechner des Brokers, der sowohl die aktuellen Kurse liefert als auch Kauf- und Verkaufverträge entgegennimmt, wesentlich schneller reagieren als wenn er von einem entfernten Rechner aus arbeitet. Diskussionen dieses Aspektes findet man unter anderem in CARZANIGA, PICCO UND VIGNA (1997), STRASSER UND SCHWEHM (1997), VOIGT (1996), PINDONIS (1996) und PULIAFITO, RICCOBENE UND SCARPA (1999).

Verteilung aufwendiger Berechnungen. Wie STRASSER, BAUMANN UND SCHWEHM (1999) zeigen, kann die Verteilung aufwendiger Berechnungen mittels mobiler Agenten sehr einfach bewerkstelligt werden. Um jedoch auf Agenten-Plattformen mit Platzkonzept zu verhindern, daß einem Rechner mit mehreren Plätzen überproportional viel Arbeit zugeteilt wird, müssen Mechanismen existieren, welche der Anwendung die Zuordnung von Plätzen zu Rechnern erlauben.

Unterstützung asynchroner Operationen und mobiler Endgeräte. Nachdem ein mobiler Agent seinen Heimatrechner (den Rechner, auf dem er gestartet wurde) verlassen hat, benötigt er normalerweise keine Verbindung mehr zu diesem Rechner. Nur für eventuelle Rückfragen bzw. das Zurückmelden des Ergebnisses (falls notwendig) ist eine kurze Verbindung zum Heimatrechner notwendig. Es ist also möglich, sowohl die Kommunikationsverbindung des Heimatrechners zu unterbrechen als auch den Heimatrechner auszuschalten. Beides ist für Mobilgeräte der heutigen Generation essentiell, da einerseits die Kommunikation normalerweise über teure und unstabile GSM-Wählverbindungen realisiert wird und andererseits die Kapazität der Stromversorgung mobiler Endgeräte nur für eine relativ kurze Betriebsdauer ausreicht. Da außerdem die Anbindung mobiler Endgeräte an stationäre Netze zum momentanen Zeitpunkt nur über sehr geringe Bandbreiten verfügt (GSM zur Zeit ca. 10kBit/s) kommt hier der weiter oben erwähnte Effekt des beschleunigten Zugriffes auf entfernte Daten besonders zum Tragen.

Dynamische Installation von Funktionalität. Durch Migration auf einen Platz kann ein Agent beliebige Funktionalität auf diesen Platz bringen, die dann von Dritten genutzt werden kann. Beispiele hierfür sind Mehrwertdienste und Protokolle. Ein Agent kann zum Beispiel auf einen Rechner migrieren und dort zu einem bereits angebotenen, einfachen Dienst eine Schnittstelle mit wesentlich ausgefeilteren Zugriffsmethoden zur Verfügung stellen. Dieser Agent nutzt dann die einfache Schnittstelle des schon vorhandenen Dienstes, um darauf aufbauend den höherwertigen Dienst (Mehrwertdienst) anzubieten. Möchte ein Benutzer auf entfernte Daten zugreifen, welche aber nur mittels eines nicht auf dem Benutzerrechner installierten Protokolls zugreifbar sind, kann ein Agent auf den Rechner des Benutzers migrieren, der die benötigte Protokollfunktionalität mitbringt.

2.1.3 Herausforderungen

Zum Entstehungszeitpunkt dieser Arbeit finden mobile Agenten nur in sehr wenigen Anwendung praktischen Einsatz. Der Grund hierfür sind einige Probleme bei der Anwendung mobiler Agenten, die vor einem breiten Einsatz mobiler Agenten gelöst werden müssen:

Sicherheit. Wie schon weiter oben erwähnt gibt es vor allem im Bereich des Schutzes mobiler Agenten vor bösartigen Ausführungsumgebungen noch Forschungsbedarf. Momentan in Entwicklung befindliche Ansätze beschreibt HOHL (2001).

Kontrolle der Agenten. Kontrollmechanismen wie das Auffinden von Agenten, das Terminieren von Agenten und die Waisenerkennung existieren zwar (vgl. BAUMANN (1999)), sind aber in den meisten Agentenplattformen nicht realisiert.

Fehlertoleranz. Mechanismen zur fehlertoleranten Ausführung mobiler Agenten existieren nur in Ansätzen. Die vorliegende Arbeit wird ihren Beitrag zur Lösung dieses Problems leisten.

"Henne-Ei-Problematik". Die Entwicklung agentenbasierter Anwendungen lohnt sich nur dann, wenn weitverbreitet Installationen von Agentenplattformen zur Verfügung stehen. Der weitverbreiteten Installation stehen jedoch zwei Gründe entgegen. Einerseits gibt es (auch aufgrund der nicht-existierenden Anwendungen) nur sehr wenige Argumente, warum ein Betreiber eines Rechners auf diesem eine Agentenplattform installieren sollte. Andererseits gibt es so viele verschiedene Agentenplattformen, daß selbst bei vielen Installationen auf jeden einzelnen Plattformtyp nur wenige Installationen entfallen (eine Besserung dieses Problems erhoffen sich die Standardisierungsversuche in FIPA (1999) und in MILOJICIC ET AL. (1998)).

2.2 Agentenmodell

Um eine möglichst breite Anwendbarkeit der in dieser Arbeit entwickelten Mechanismen in den existierenden Agentenplattformen zu gewährleisten, ist das in dieser Arbeit verwendete Agenten-Modell, das in diesem Unterkapitel vorgestellt wird, bewußt generisch gehalten.

Ein mobiler Agent bewegt sich zur Erfüllung seiner Aufgabe durch ein Netzwerk von Rechnern und nutzt die lokal auf den Rechnern vorhandenen *Ressourcen*. Um die Beschreibung der in dieser Arbeit entwickelten Mechanismen zu vereinfachen, existiert auf jedem Rechner – in Übereinstimmung mit der im Bereich der Rechnernetze gebräuchlichen Nomenklatur hier auch (Rechen-)*Knoten* (engl.: *node*) genannt – genau eine Ausführungsplattform für Agenten; ein Platz-Konzept existiert nicht. Die Mechanismen sind in leicht abgewandelter Form jedoch auch anwendbar, wenn mehrere Ausführungsplattformen auf einem Rechner existieren können. Mobile Agenten können nicht miteinander kommunizieren und können auch keine neuen Agenten starten.

Agenten sind mittels eines global eindeutigen Namens identifizierbar, der den Agenten bei der Erzeugung zugewiesen wird und der sich während der Laufzeit des Agenten nicht ändert. Ein Mechanismus zum Auffinden von Agenten wird nicht vorausgesetzt.

Die Aufgabe eines Agenten ist in Teilaufgaben unterteilt, welche vom Agent sequentiell ausgeführt werden. Die Ausführung einer Teilaufgabe wird als *Schritt* (engl.: *step*) bezeichnet. Eine Teilaufgabe kann während der Ausführungszeit eines Agenten mehrmals ausgeführt werden. Hat ein Einkaufsagent beispielsweise die Aufgabe, zuerst nach dem günstigsten Preis zu suchen und erst dann die gewünschten Artikel zu kaufen, wird die Teilaufgabe *DurchsuchePreisliste* bei jedem Händler durchgeführt.

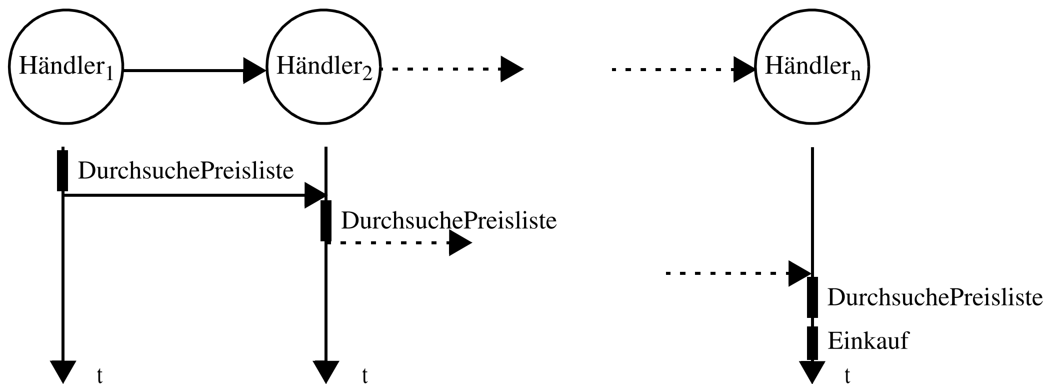


Abbildung 2-1. Ausführung eines Einkaufsagenten in Schritten

Ein Schritt wird jeweils vollständig auf einem Knoten abgearbeitet. Auf einem Knoten können mehrere Schritte ausgeführt werden, wobei diese Schritte nicht direkt aufeinanderfolgen müssen (aber können). Abbildung 2-1 zeigt die Ausführung eines Einkaufsagenten. Der Agent konsultiert zuerst die Preislisten der Händler $Händler_1$, $Händler_2$,..., $Händler_n$, indem er auf jedem Händler-Rechner *DurchsuchePreisliste* ausführt. Beim zuletzt besuchten Händler $Händler_n$ beschließt er (noch in *DurchsuchePreisliste*), daß der Einkauf hier am günstigsten ist und führt die Teilaufgabe *Einkauf* aus.

Die letztendliche Zuordnung von einzelnen Schritten zu Knoten läßt sich durch die Zuordnungsfunktion **z** beschreiben:

$$\text{z:}\ S \rightarrow K \qquad (2\text{-}1)$$

Hierbei ist S die Menge $\{S_1, S_2, ..., S_n\}$ der Schritte eines Agenten und K die Menge der verfügbaren Knoten. Die Schritte des Agenten werden in der Reihenfolge S_1, S_2, ..., S_n ausgeführt. Im obigen Beispiel haben die Schritte S_1 bis S_n dieselbe Teilaufgabe *DurchsuchePreisliste*, Schritt S_{n+1} tätigt den Einkauf. Es ist zu beachten, daß diese Funktion nur die tatsächliche Ausführungsreihenfolge und die tatsächlichen Ausführungsknoten der Schritte eines Agenten widerspiegelt. Die Funktion muß bei Beginn der Ausführung eines Agenten noch nicht bekannt sein. Im folgenden Kapitel wird eine Möglichkeit der Definition einer flexiblen Reiseroute eingeführt, welche Alternativen in der Definition der Ausführungsreihenfolge und der Ausführungsknoten von Schritten ermöglicht. Die Zuordnungsfunktion **z** beschreibt die bei der Ausführung des Agenten tatsächlich gewählte Reiseroute.

Die Fortbewegung des Agenten geschieht mittels schwacher Migration, d.h. es werden der Programm-Code und der Datenzustand des Agenten (z.B. globale Variablen) migriert. Der Ausführungszustand (Stack, Programmzähler etc.) wird nicht migriert. Ein auf einem Knoten ankommender Agent wird gestartet, indem der Datenzustand des Agenten auf dem Knoten instantiiert und der den nächsten auszuführenden Schritt realisierende Code des Agenten ausgeführt wird.

Während der Ausführung eines Schrittes auf einem Knoten ändern sich sowohl der Datenzustand des Agenten als auch der Zustand der Ressourcen, auf die der Agent (direkt oder indirekt) zugreift; der Code des Agenten ändert sich nicht. Um dies in einer etwas formaleren Notation fassen zu können, werden zuerst einige Abkürzungen eingeführt:

Definition 2-1: Knoten K_i

bezeichnet den Knoten, auf dem der i-te Schritt S_i eines Agenten A ausgeführt wird. Die Zuordnungsfunktion **z** bildet also S_i auf K_i ab.

Definition 2-2: Agentenzustand A_i

beschreibt den Datenzustand des Agenten A vor der Ausführung des i-ten Schrittes. A_1 ist der Startzustand des Agenten.

Definition 2-3: Ressourcenzustände R_i^1, R_i^2

beschreiben die Zustände der Ressourcen von Knoten K_i vor (R_i^1) bzw. nach (R_i^2) der Ausführung des i-ten Schrittes des Agenten.

Die Schrittfunktion $\boldsymbol{s_i}$ beschreibt die Änderung des Agenten- und Ressourcenzustandes durch einen Schritt i:

$$s_i\colon (A_i, R_i^1) \mapsto (A_{i+1}, R_i^2) \qquad (2\text{-}2)$$

Diskussion. Das hier vorgestellte Agentenmodell ermöglicht ein breites Spektrum von Anwendungen mobiler Agenten, deckt jedoch durch seine Einschränkungen hinsichtlich der Kommunikation zwischen Agenten und der Erzeugung neuer Agenten nicht das gesamte Spektrum der möglichen Anwendungen mobiler Agenten ab. Bei der Beschreibung der in dieser Arbeit entwickelten Mechanismen wird ausführlich darauf eingegangen, weshalb die Einschränkungen im Modell gemacht wurden und welche Implikationen sich durch die Aufhebung der Einschränkungen ergeben würden.

Kapitel 3

Reiserouten

Die Ausführung der Teilaufgaben eines mobilen Agenten geschieht im allgemeinen auf unterschiedlichen Rechnern um dort auf lokale Dienste und Ressourcen zuzugreifen. Oftmals sind einige (oder alle) dieser vom Agent zu besuchenden Knoten schon beim Start des Agenten bekannt oder können durch den Agent viele Schritte im voraus bestimmt werden. Wie im realen Leben existiert häufig jedoch keine strikte Reihenfolge, in der die Knoten besucht werden müssen. Durchsucht beispielsweise ein Agent zuerst die Artikellisten verschiedener Händler auf verschiedenen Knoten, um sich dann für den Kauf bei einem Händler zu entscheiden, dann ist aus Anwendungssicht die Besuchsreihenfolge der Händler-Knoten zum Durchsuchen der Listen irrelevant. Muß ein Agent eine Karte für das Theater und eine CD bestellen, dann ist die Reihenfolge dieser beiden Aktionen ebenfalls beliebig. Darüberhinaus besteht oftmals die Möglichkeit, frei zwischen mehreren Knoten zu wählen, falls alle diese Knoten denselben Service (zu denselben Konditionen) anbieten. Sowohl durch die flexible Besuchsreihenfolge als auch durch alternativ zu besuchende Knoten können dem Agent also zum Zeitpunkt einer Migration potentiell mehrere Migrationsziele zur Verfügung stehen.

Wenn man diese Freiheiten im Reiseplan der Agentenplattform in geeigneter Form sichtbar macht, kann man diese Information in vielerlei Hinsicht benutzen. Eine mögliche Verwendung ist die Optimierung der Reiseroute durch die Berechnung des kürzest möglichen Pfades für den Agenten. Eine andere, im Kontext dieser Arbeit wesentlich wichtigere Verwendung dieser Information ist die Erhöhung der Fehlertoleranz im Agentensystem. Stehen zum Zeitpunkt der Migration laut Reiseplan mehrere alternative Migrationsziele zur Verfügung, dann führt die (kurzfristige) Nichtverfügbarkeit eines dieser Knoten nicht zur Blockierung der Ausführung des Agenten. Vielmehr besteht in diesem Falle die Möglichkeit, den ausgefallenen Knoten zu einem späteren Zeitpunkt zu besuchen oder, falls der Knoten einer von mehreren alternativ zu besuchenden Knoten ist, einen der Alternativknoten zu besuchen.

Um einerseits die Agentenplattform mit den notwendigen Informationen zu versorgen und andererseits aber auch dem Entwickler von Mobile-Agenten-Anwendungen ein mächtiges Werk-

zeug in die Hand zu geben, werden in diesem Kapitel zwei *Reiseroutenkonzepte* (engl.: *itinerary concepts*) vorgestellt. Beide Konzepte erlauben sowohl die Spezifikation eines flexiblen Reiseplans für mobile Agenten als auch die dynamische Anpassung und Erweiterung des Reiseplans während der Ausführung des Agenten. Das zuerst dargestellte, einfache Reiseroutenkonzept ist sehr einfach in die Agentenplattform integrierbar und intuitiv verständlich, jedoch ist es in seiner Flexibilität eingeschränkt. Das zweite, komplexere Reiseroutenkonzept hingegen erlaubt die Spezifikation sehr flexibler Reiserouten. Zur einfachen Verwendung können bei diesem komplexeren Konzept ähnliche Abstraktionen wie beim einfachen Reiseroutenkonzept angeboten werden, die vollständige Ausnutzung der Flexibilität dieses Konzeptes erfordert jedoch die Spezifikation der Reiseroute mittels boolescher Ausdrücke. Beide Konzepte wurden erstmals in STRASSER UND ROTHERMEL (1998) veröffentlicht.

3.1 Ein einfaches Reiseroutenkonzept

Die *einfache Reiseroute* setzt sich aus verschiedenen Typen von *Reiserouteneinträgen* zusammen. Die einfachste, erstmals in WONG ET AL. (1997) eingeführte Form eines Eintrages ist ein einfacher (*Knoten*, *Methode*)-Eintrag, der die auszuführende Teilaufgabe (definiert durch *Methode*) und den Knoten, auf dem die Teilaufgabe auszuführen ist, angibt. Diese Form des Reiserouteneintrages wird *Schritteintrag* genannt. Andere mögliche Einträge sind die *Sequenz*, die *Menge* und die *Alternative*. Diese Einträge enthalten (rekursiv) weitere Reiserouteneinträge.

Eine *Sequenz* ist eine Liste $[e_1, ..., e_n]$ von n Einträgen ($n \geq 1$). Sie legt fest, daß die durch den Eintrag e_i ($1 \leq i < n$) spezifizierten Schritteinträge ausgeführt sein müssen bevor die durch e_{i+1} spezifizierten Schritteinträge ausgeführt werden. Ist also beispielsweise der Eintrag e_i auch eine Liste und e_{i+1} ein Schritteintrag, so müssen alle Einträge der Liste e_i ausgeführt worden sein bevor der Schritteintrag e_{i+1} ausgeführt wird.

Eine *Menge* $\{e_1, ..., e_n\}$ von Reiserouteneinträgen legt fest, daß die Einträge $e_1, ..., e_n$ in beliebiger Reihenfolge abgehandelt werden können. Da jeder Eintrag selbst wieder (rekursiv) andere Einträge enthalten kann ist zu beachten, daß die durch e_i spezifizierten Einträge entweder alle vor oder alle nach e_j auszuführen sind ($1 \leq i,j \leq n$; $i \neq j$). Sind beispielsweise $e_i = \{e_{i1}, ..., e_{ik}\}$ und $e_j = \{e_{j1}, ..., e_{jm}\}$ selbst wieder Mengen, dann ist zwar die Reihenfolge, in der e_i und e_j ausgeführt werden beliebig, eine gemischte (verschränkte) Ausführung der beiden Mengen $\{e_{i1}, ..., e_{ik}\}$ und $\{e_{j1}, ..., e_{jm}\}$ ist jedoch nicht erlaubt, d.h. die Einträge $e_{i1}, ..., e_{ik}$ müssen entweder komplett vor oder komplett nach den Einträgen $e_{j1}, ..., e_{jm}$ ausgeführt werden.

Die *Alternative* $<e1, ..., e_n>$ spezifiziert, daß genau einer der Einträge $e_1, ..., e_n$ ausgewählt werden muß.

Das folgende Szenario soll helfen, diese Definition besser zu verstehen: Paul plant, einen romantischen Abend mit seiner Freundin zu verbringen. Hierzu beauftragt er seinen persönlichen Butler-Agenten, Blumen zu bestellen, eine Eintrittskarte für den aktuell in zwei Kinos der Stadt

laufenden Liebesfilm zu kaufen und einen Tisch in einem dem Kino nahegelegenen Restaurant zu reservieren. Um den Auftrag auszuführen, muß der Agent den Fleurop-Rechner besuchen, weiterhin einen der beiden für die jeweiligen Kinos zuständigen Rechner und schließlich den Knoten jenes Restaurants, das dem tatsächlich gewählten Kino näher ist.

Die nachfolgend mit Hilfe der oben definierten Notation dargestellte Reiseroute *i* (von engl.: itinerary) spezifiziert den Reiseplan des Butler-Agenten:

```
i = {(Fleurop, kaufeBlumen),
    <[ (Luna, kaufeEintrittskarte), (Rößle, reserviereTisch) ],
      [ (Planie, kaufeEintrittskarte), (Linde, reserviereTisch) ]
    >
  }.
```

Eine graphische Repräsentation der Reiseroute zeigt Abbildung 3-1. Auf oberster Ebene spezifiziert ein *Menge*-Eintrag, daß der Agent auf dem Knoten “Fleurop” mittels der Methode “kaufeBlumen” Blumen einkaufen soll und daß der spezifizierte *Alternative*-Eintrag ausgeführt werden muß. Hierbei kann er zuerst die Blumen kaufen und dann den *Alternative*-Eintrag ausführen oder umgekehrt. Die *Alternative* spezifiziert zwei Möglichkeiten, die Eintrittskarte zu kaufen und einen Tisch zu reservieren. Jede dieser Möglichkeiten ist mittels einer *Sequenz* spezifiziert, in der festgelegt wird, daß zuerst das Ticket gekauft und anschließend im dazu passenden Restaurant ein Tisch reserviert wird. Die Sequenz ist hierbei notwendig, da der Agent die zur Reservierung notwendige Uhrzeit (“wann endet der Film?”) erst beim Kauf der Eintrittskarte erfährt.

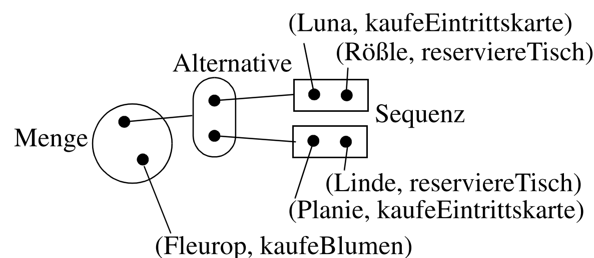


Abbildung 3-1. Reiseroute des Butler-Agenten

Anhand dieser Reiseroute kann das System bestimmen, welche Knoten der Agent als nächstes besuchen kann. Für den ersten Schritt stehen hier entweder der Blumenladen oder eines der beiden Kinos zur Auswahl. Der zweite und die darauffolgenden Schritte hängen jeweils von der Wahl der vorhergehenden Schritte ab. Anhand der in einer Reiseroute enthaltenen Informationen kann ein Baum erstellt werden, welcher alle möglichen Ausführungspfade des Agenten ent-

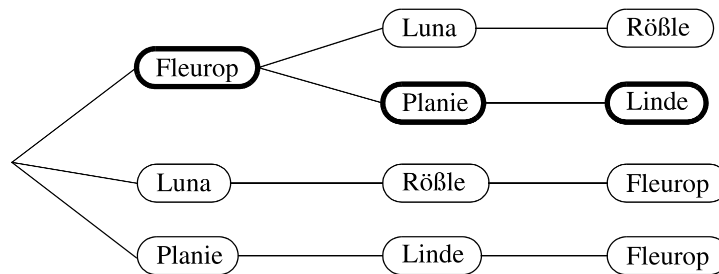


Abbildung 3-2. Baum der möglichen Pfade des Butler-Agenten

hält. Der Baum in Abbildung 3-2 zeigt alle möglichen Ausführungspfade von Pauls Agent. Der Pfad, bei dem der Agent zuerst Blumen kauft, danach Karten für die Planie kauft und einen Tisch in der Linde reserviert ist in der Abbildung hervorgehoben.

Die Reiseroute stellt Abfrage- und Änderungsoperationen zur Verfügung. Die Abfrageoperationen erlauben die Navigation durch die Einträge einer Reiseroute, ermöglichen abzufragen, welche Einträge schon abgearbeitet wurden und welche Knoten als nächstes besucht werden können. Die Änderungsoperationen erlauben das Einfügen und Löschen von Einträgen in jene Teile der Reiseroute, welche noch nicht abgearbeitet wurden. Dies erlaubt dem Agent einerseits, einen Überblick über seine bisherige Ausführung zu gewinnen und andererseits, seine Reiseroute während der Ausführung dynamisch an die jeweiligen Gegebenheiten anzupassen.

3.2 Ein flexibleres Reiseroutenkonzept

Obwohl das im vorherigen Unterkapitel vorgestellte Reiseroutenkonzept schon ein recht mächtiges Werkzeug zur Spezifikation von Reiseplänen bereitstellt, bietet es doch nur eingeschränkte Flexibilität sowohl hinsichtlich der Mächtigkeit der Definitionsmöglichkeiten bei der Erstellung der Reiseroute als auch hinsichtlich der sich aus einer Reiseroute ergebenden Wahlmöglichkeiten während der Ausführung. Die folgenden Ausführungsreihenfolgebeziehungen können beispielsweise nicht (oder nur mit sehr viel Aufwand) mittels dieses Konzeptes spezifiziert werden:

- Knoten N_i muß vor Knoten N_j besucht werden, dazwischen dürfen jedoch beliebige andere Knoten besucht werden. Im Beispiel aus dem letzten Unterkapitel ist beispielsweise die Angabe, daß der Blumenladen auch zwischen dem Kauf der Eintrittskarte und dem Reservieren des Tisches besucht werden kann, nur schwerlich zu spezifizieren.
- Prioritäten zwischen Einträgen (z.B. das Kino Planie gegenüber dem Luna bevorzugen).
- Mindestens i Knoten aus einer Menge von j Knoten ($j \geq i$) besuchen bevor man einen anderen Knoten besucht (z.B. vor einem Einkauf mindestens i Angebote einholen).
- Den Besuch eines Knotens vom Zustand des Agenten abhängig machen, z.B. einen Knoten nur dann besuchen, wenn noch nicht genügend Informationen gesammelt wurden.

Das in diesem Unterkapitel vorgestellte, sehr flexible Reiseroutenkonzept hingegen unterstützt die Spezifikation solcher Reihenfolgebeziehungen beinahe vollständig. Lediglich die Einführung von Reihenfolgebeziehungen basierend auf dem Zustand des Agenten wird aus semantischen Gründen nicht unterstützt werden.

Eine Implementierung des Reiseroutenkonzeptes und dessen Integration in das Agentensystem Mole (BAUMANN ET AL. (1998A)) findet man in BUSCHLE (1999).

3.2.1 Die flache Reiseroute

Die Grundidee für das flexible Reiseroutenkonzept basiert auf der Beobachtung, daß die beim einfachen Reiseroutenkonzept mittels *Sequenz* und *Alternative* spezifizierten Reihenfolgebeziehungen aus der Sicht eines einzelnen Schritteintrages implizit aussagen, welche Bedingungen für die Ausführung des Schrittes erfüllt sein müssen. Die Sequenzen der Reiseroute aus Abschnitt 3.1 spezifizieren beispielsweise, daß vor der Reservierung eines Tisches zuerst die Eintrittskarte gekauft werden muß (d.h. der Schritt *kaufeEintrittskarte* muß auf dem Kino-Knoten ausgeführt worden sein). Die Alternative spezifiziert, daß der Kauf einer Eintrittskarte beim Luna-Theater nur erfolgen darf, wenn beim Planie-Theater noch keine Karte gekauft wurde (und umgekehrt). In dem flexiblen Reiseroutenkonzept wird daher der Ansatz gewählt, daß für jeden Eintrag einer Reiseroute explizit eine (boolesche) Bedingung spezifiziert wird. Nur wenn diese (Vor-)Bedingung wahr wird, kann der Eintrag ausgeführt werden. Zusätzlich wird die Spezifikation von Präferenzen zwischen Einträgen ermöglicht.

Definition 3-1: Reiseroute $\mathfrak{R}$

Eine Reiseroute $\mathfrak{R}=(E, \boldsymbol{P})$ besteht aus einer Menge $E=\{e_1, e_2, ..., e_n\}$ von Reiserouteneinträgen und einer Prioritätsrelation $\boldsymbol{P} \subset E$xE zwischen diesen Einträgen.

Definition 3-2: Reiserouteneintrag *e*

Ein (Reiserouten-)Eintrag *e* ist ein Tripel (*Vorbedingung*, *Knoten*, *Methode*). Das Tripel spezifiziert, daß der Agent zu *Knoten* migrieren und die durch *Methode* implementierte Teilaufgabe ausführen kann (kurz: den Eintrag ausführen), falls die *Vorbedingung* zur Migrationszeit erfüllt ist und der Eintrag noch nicht ausgeführt wurde.

Um nach der Durchführung eines Schrittes (oder zu Beginn der Ausführung des Agenten) den nächsten auszuführenden Schritt zu bestimmen, werden die Vorbedingungen der noch nicht ausgeführten Reiserouteneinträge ausgewertet und unter Berücksichtigung der Prioritätsrelation $\boldsymbol{P}$ wird einer der Einträge ausgewählt, dessen Vorbedingung zu *wahr* ausgewertet wurde. Wird keine der ausgewerteten Vorbedingungen *wahr*, ist die Bearbeitung des Agenten beendet. Im fehlerfreien Fall wird jeder Eintrag höchstens einmal ausgeführt.

Vorbedingung(e) bezeichnet die Vorbedingung eines Eintrages e. Eine Vorbedingung, welche ein boolescher Ausdruck $p(e_1, e_2, ..., e_n)$ ist, wird als erfüllt bezeichnet, wenn sie zur Migrationszeit zu *wahr* ausgewertet werden kann. Hat ein Eintrag die triviale Vorbedingung *wahr*, dann bedeutet dies, daß er jederzeit ausgeführt werden kann. Ein Eintrag kann die triviale Vorbedingung *falsch* haben, wird dann aber nie ausgeführt (und ist deshalb sinnlos). Um sich in der Vorbedingung auf andere Einträge der Reiseroute zu beziehen, wird das Prädikat $D(e)$ (von engl.: done) verwendet. Eine vorläufige Definition von $D(e)$ ist gegeben durch

Definition 3-3: Prädikat $D(e)$

$D(e) \equiv wahr \Leftrightarrow$ Eintrag e wurde schon ausgeführt

Prioritäten zwischen Einträgen einer Reiseroute werden durch die Prioritätsrelation $\boldsymbol{P}$ spezifiziert:

Definition 3-4: Prioritätsrelation $\boldsymbol{P} \subset S \mathrm{x} S$

$(e_i, e_j) \in \boldsymbol{P} \Leftrightarrow$ Eintrag e_i hat höhere Priorität als Eintrag e_j

Die Priorität $(e_i, e_j) \in \boldsymbol{P}$ zwischen zwei Einträgen e_i und e_j spezifiziert nur, daß, falls die beiden Einträge zu einem Zeitpunkt alternativ ausgeführt werden können (d.h. die Vorbedingungen der beiden Einträge sind erfüllt und beide Einträge wurden noch nicht ausgeführt), die Ausführung des Eintrages e_i bevorzugt wird. Ist die Ausführung von e_i aus technischen Gründen nicht möglich (z.B. da der durch e_i spezifizierte Ausführungsknoten nicht erreichbar ist) oder ist die Vorbedingung von e_i nicht erfüllt, kann e_j ausgeführt werden falls dessen Vorbedingung erfüllt ist. Aus Konsistenzgründen darf die Relation $\boldsymbol{P}$ keine Zyklen (z.B. $(e_i,e_j), (e_j,e_k), ..., (e_l,e_m), (e_m,e_i)$) enthalten.

Das folgende Beispiel illustriert die durch die bisher beschriebenen Teile des Konzept gegebenen Möglichkeiten:

Beispiel 3-1. Reiseroute $\Re = (\{e_1, e_2, e_3, e_4, e_5\}, \boldsymbol{P})$ für das Szenario aus Abschnitt 3.1:

$e_1 = (wahr, Fleurop, kaufeBlumen)$
$e_2 = (\neg D(e_4), Luna, kaufeEintrittskarte)$
$e_3 = (D(e_2), Rößle, reserviereTisch)$
$e_4 = (\neg D(e_2), Planie, kaufeEintrittskarte)$
$e_5 = (D(e_4), Linde, reserviereTisch)$
$\boldsymbol{P} = \{(e_2, e_4)\}$

Die hiermit spezifizierte Reiseroute ist flexibler als die in Abschnitt 3.1 spezifizierte Reiseroute, da bei dieser Definition der Einkauf der Blumen jederzeit möglich ist, und dadurch während der Ausführung des Agenten bei der Auswahl des nächsten durchzuführenden Schrittes im Durchschnitt mehr Alternativen zur Verfügung stehen als bei der einfachen Reiseroute. Die Bevorzugung der Kombination Luna/Rößle gegenüber der

Kombination Planie/Linde stellt zumindest hinsichtlich der Fehlertoleranz keine (wesentliche) Einschränkung der Flexibilität dar, da sie nur in dem Fall eine Rolle spielt, wenn beide Kino-Knoten verfügbar sind. Der Anwender erhält damit jedoch die zusätzliche Möglichkeit, seine Präferenzen zwischen Alternativen auszudrücken. Die Möglichkeit, jederzeit Blumen zu kaufen, ergibt sich daraus, daß der Eintrag e_1, welcher das Kaufen der Blumen spezifiziert, die triviale Vorbedingung *wahr* hat und die anderen Einträge in keiner Weise von ihm abhängen. Die Einträge e_2 und e_4 beschreiben das Kaufen der Eintrittskarte, wobei die Vorbedingungen sicherstellen, daß nur einer der Einträge ausgeführt werden kann: e_2 kann nur ausgeführt werden, falls e_4 (noch) nicht ausgeführt wurde (ausgedrückt durch $\neg D(e_4)$) und umgekehrt. Die Vorbedingungen e_3 und e_5 der Schritte zum Reservieren des Tisches stellen sicher, daß in einem der Restaurants nur dann reserviert wird, wenn im entsprechenden Kino die Karte gekauft wurde: Z.B. garantiert die Vorbedingung $D(e_2)$ von e_3, daß nur dann im Rößle ein Tisch reserviert wird, falls in e_2 im Luna die Eintrittskarte gekauft wurde. Die Bevorzugung der Kombination Luna/Rößle ergibt sich aus der Spezifikation der Prioritätsrelation $\boldsymbol{P}$.

Um die Semantik der in Abschnitt 3.1 spezifizierten Reiseroute zu erhalten, muß die Spezifikation der Prioritätsrelation in $\boldsymbol{P}=\emptyset$ geändert werden und die Vorbedingung von e_1 wesentlich komplexer ausfallen:

$e_1 = ((\neg D(e_2) \wedge \neg D(e_4)) \vee (D(e_3) \vee D(e_5)), \mathit{Fleurop}, \mathit{kaufeBlumen})$

Hiermit wird spezifiziert, daß die Blumen entweder vor dem Kauf der Eintrittskarten $(\neg D(e_2) \wedge \neg D(e_4)$, d.h. in keinem der Kinos darf eine Karte gekauft worden sein) oder nach der Reservierung des Tisches $(D(e_3) \vee D(e_5)$, d.h. in einem der Restaurants muß ein Tisch reserviert worden sein) gekauft werden müssen.

Mit den bisher vorgestellten, recht einfachen Mitteln ist es sogar möglich zu spezifizieren, daß i Einträge aus einer Menge von j Einträgen $e_1, ..., e_j$ $(i \leq j)$ ausgeführt werden müssen. Hierbei werden jedoch die Vorbedingungen der Einträge sehr komplex, solange nur das Prädikat $D(e)$ verwendet werden kann:

Beispiel 3-2. Ein Agent soll von zwei Anbietern je ein Angebot für einen Artikel einholen. Es stehen insgesamt 4 Anbieter zur Verfügung. Der Eintrag e_k beschreibt das Abholen eines Angebotes von Anbieter k auf dessen Knoten n_k $(k = 1...4)$:

$e_1 = (\neg((D(e_2) \wedge (D(e_3) \vee D(e_4))) \vee (D(e_3) \wedge D(e_4))), n_1, \mathit{holeAngebot})$

$e_2 = (\neg((D(e_1) \wedge (D(e_3) \vee D(e_4))) \vee (D(e_3) \wedge D(e_4))), n_2, \mathit{holeAngebot})$

e_3 und e_4 analog

Sollen hierbei Anbieter bevorzugt werden, dann kann dies zusätzlich mittels Prioritäten geschehen.

Offensichtlich werden die Vorbedingungen für größere Mengen von Einträgen wesentlich komplexer. Aus diesem Grunde werden in den Vorbedingungen zusätzlich boolesche Terme der Form

(Zahl Vergleichsoperator Ausdruck)

zugelassen. Ein Ausdruck ist eine Summe in der Form

$$d(e_a) + d(e_b) + ... + d(e_z)$$

Die Funktion $d(e)$ ist definiert durch

Definition 3-5: Funktion $d(e)$

$d: e \rightarrow \{0,1\}$
$d(e) \equiv 1 \Leftrightarrow D(e) \equiv wahr$

Die Vergleichsoperator kann aus der Menge $\{<, \leq, =, \geq, >\}$ entnommen werden. Wie die zwei folgenden Beispiele zeigen, lassen sich hiermit einfach Beziehungen wie "*i* Einträge aus *j* Einträgen ausführen" oder "mindestens *i* Einträge aus *j* Einträge ausführen bevor ..." spezifizieren.

Beispiel 3-3. Analog zum vorherigen Beispiel soll ein Agent von *i* Anbietern Angebote einholen. Es stehen insgesamt *j* ($i \leq j$) Anbieter zur Verfügung. Der Eintrag e_k beschreibt das Abholen eines Angebotes von Anbieter *k* auf dessen Knoten n_k ($k = 1...j$):

$e_k = ((i > d(e_1)+d(e_2)+ ... +d(e_j)), n_k, holeAngebot)$ für $k = 1 ... j$

Die Vorbedingungen werden *falsch*, sobald *i* Einträge ausgeführt wurden.

Beispiel 3-4. Ein Agent soll von *j* Anbietern Angebote einholen und diese auf seinem Heimatknoten abliefern. Sind von den *j* Anbietern momentan nicht alle erreichbar (Knotenausfall o.ä.), so kann der Agent auch weniger als *j* Angebote abliefern, er muß jedoch mindestens *i* ($i \leq j$) Angebote eingeholt haben. Die Einträge e_k, $k=1...j$ beschreiben das Einholen von Angeboten, e_{j+1} das Abliefern der Angebote:

$e_k = (\neg D(e_{j+1}), n_k, holeAngebot)$ für $k = 1 ... j$
$e_{j+1} = ((i \leq d(e_1)+d(e_2)+ ... +d(e_j)), Heimatknoten, liefereAngeboteAb)$
$\boldsymbol{P} = \{(e_1, e_{j+1}), (e_2, e_{j+1}), ..., (e_j, e_{j+1}), \}$

Die Vorbedingung von e_{j+1} wird wahr, sobald mindestens *i* Einträge aus e_1, ..., e_j ausgeführt wurden. Ab diesem Zeitpunkt können die Angebote auf dem Heimatknoten abgeliefert werden. Die Prioritäten legen jedoch fest, daß das Einholen von Angeboten Vorrang vor dem Abliefern hat, so daß der Agent nur bei Nichtverfügbarkeit eines Anbieter-Knotens die Angebote vorzeitig abliefert. Die Vorbedingungen der Einträge e_1,..., e_j stellen sicher, daß nach dem Abliefern der Angebote keine neuen Angebote mehr eingeholt werden.

Abbildung 3-3 zeigt eine bis auf die Definition der Syntax von Reiserouteneinträgen vollständige Zusammenfassung der Syntax von Vorbedingungen in Erweiterter Backus-Naur-Form (EBNF):

Vorbedingung=("(" Vorbedingung ")" | "¬" Vorbedingung |
Vorbedingung "∧" Vorbedingung |
Vorbedingung "∨" Vorbedingung | Lit).

Lit = (Prädikat | Gleichung | "*wahr*" | "*falsch*").

Prädikat = "*D*(" Eintrag ")".

Gleichung= "(" Zahl Operator Ausdruck ")".

Operator = ("<" | "≤" | "=" | "≥" | ">").

Ausdruck = (Ausdruck "+" Ausdruck | Funktion).

Funktion = "*d*(" Eintrag ")".

Zahl = (“0” | ZifferON { ("0" | ZifferON) }).

ZifferON = ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9").

Abbildung 3-3. Syntax einer Vorbedingung in EBNF

Hiermit werden die am Anfang von Abschnitt 3.2 aufgeführten Anforderungen an die Reiseroute schon weitestgehend erfüllt. Ausnahme ist die Anforderung, daß der Besuch eines Knotens (im Endeffekt die Ausführung eines Reiserouteneintrages) vom Zustand des Agenten abhängig gemacht werden kann. Diese Anforderung könnte durch die Verwendung anwendungsspezifischer Prädikate in den Vorbedingungen der Reiserouteneinträge erfüllt werden. Soll ein Agent beispielsweise einen Eintrag der Reiseroute nur dann ausführen, nachdem er genügend Information gesammelt hat, dann könnte ein vom Agentenentwickler implementiertes Prädikat *hatGenugInformation*(...), welches genau dann *wahr* liefert, sobald genug Informationen im Agent enthalten sind, in der Vorbedingung zu dem betreffenden Eintrag verwendet werden. Es gibt jedoch mehrere Gründe, diesen Typ von Prädikat nicht in das Reiseroutenkonzept zu übernehmen. Einer der Gründe ist, daß das Reiseroutenkonzept durch solche Prädikate wesentlich komplexer würde: Einerseits würden Reiserouten dadurch schwerer verständlich, da hiermit Entscheidungen, die der Agent selbst aktiv treffen sollte, in die Reiseroute verschoben würden. Andererseits müßten alle diese Prädikate bei jeder Migration neu ausgewertet werden, was bei komplex zu berechnenden Prädikaten sehr viel Mehraufwand für die Ausführung eines Agenten bedeuten würde. Der Hauptgrund gegen die Einführung dieses Typs von Prädikaten wird jedoch erst im Laufe des nächsten Abschnittes klar.

3.2.2 Geschachtelte Reiserouten

Ein Nachteil der flachen Reiseroute besteht darin, daß es hiermit nicht möglich ist, eine Menge von Reiserouteneinträgen als eine logische Einheit zu behandeln:

Beispiel 3-5. Ein Agent soll von i Anbietern Angebote für ein Produkt in 2 Runden einholen. In der ersten Runde wird zuerst von jedem Anbieter ein Angebot eingeholt:

$e_k = (wahr, n_k, holeAngebot)$ für $k = 1 \dots i$

Basierend auf dem Ergebnis der ersten Runde wird dann in der zweiten Runde versucht, bessere Angebote auszuhandeln. Die Einträge für die zweite Runde dürfen erst dann ausgeführt werden, wenn die Einträge für die erste Runde komplett ausgeführt wurden:

$e_{i+k} = (e_1 \wedge e_2 \wedge \dots \wedge e_i, n_k, verhandleAngebot)$ für $k = 1 \dots i$

Die Reiseroute des Agenten ergibt sich hieraus zu

$\Re = (\{e_1, e_2, \dots, e_{2i}\}, \varnothing)$

Dies (oder noch komplexere Szenarien) zu spezifizieren ist nicht nur mühsam und fehleranfällig, es müssen darüberhinaus auch bei jeder Migration die Vorbedingungen zumindest teilweise neu berechnet werden. Kommt im Beispiel noch ein weiterer Anbieter nachträglich hinzu, müssen zudem noch alle Vorbedingungen der Einträge für die zweite Runde geändert werden.

Um diese Nachteile zu überwinden, wird das Konzept der *geschachtelten Reiserouten* eingeführt. Geschachtelte Reiserouten erweitern die flachen Reiserouten um die Möglichkeit, zusätzlich komplette Reiserouten als Einträge in einer Reiseroute zu spezifizieren. Hierdurch ergeben sich für viele komplexe Szenarien wesentliche Vereinfachungen bei der Spezifikation von Reiserouten. Für jede geschachtelte Reiseroute läßt sich jedoch auch eine flache Reiseroute angeben, deren Einträge dann wesentlich komplexere Vorbedingungen enthalten als die Einträge der geschachtelten Reiseroute.

Die folgenden Definitionen legen die Struktur geschachtelter Reiserouten und dabei verwendete Begriffe fest:

Definition 3-6: Eintrag e einer geschachtelten Reiseroute

Ein Eintrag e einer geschachtelten Reiseroute ist entweder ein Basis-Eintrag (*B-Eintrag*) oder eine Reiserouten-Eintrag (*R-Eintrag*).

Definition 3-7: Basis-Eintrag (*B-Eintrag*) einer geschachtelten Reiseroute

Ein Basis-Eintrag entspricht dem in Definition 3-2 definierten Tripel (Vorbedingung, Knoten, Methode).

Definition 3-8: Reiserouten-Eintrag (*R-Eintrag*) einer geschachtelten Reiseroute

Ein *R-Eintrag e* ist ein Quadrupel (*Vorbedingung*, *E*, ***P***, *Typ*), welches eine komplette (Teil-)Reiseroute beschreibt. $E=\{e_1, e_2, ..., e_n\}$ ist die Menge der in dem Reiserouten-Eintrag direkt enthaltenen Einträge; die Einträge $e_1, e_2, ..., e_n$ dürfen in keinem anderen R-Eintrag direkt enthalten sein. Um mit der Ausführung der in dem R-Eintrag *e* enthaltenen Einträge (d.h. mit der Ausführung der durch *e* beschriebenen Reiseroute) zu beginnen, muß die *Vorbedingung* erfüllt sein. ***P*** spezifiziert gemäß Definition 3-4 die Prioritätsrelation zwischen den Einträgen $e_1, e_2, ..., e_n$. Die Definition des Typs eines *R-Eintrages* erfolgt weiter unten.

Definition 3-9: Menge *Einträge*(*e*) der in einem *R-Eintrag* direkt enthaltenen Einträge

$$e = (Vorbedingung, \{e_1, e_2, ..., e_n\}, \boldsymbol{P}, Typ) \Rightarrow Einträge(e) \equiv \{e_1, e_2, ..., e_n\}$$

Da ein *R-Eintrag e* eine vollständige Reiseroute spezifiziert, dürfen die Vorbedingungen der direkten Einträge von *e* jeweils nur von den direkten Einträgen von *e* (d.h. von *Einträge*(*e*)) abhängen, d.h. $Vorbedingung(e_i)=p_i(e_1, e_2,..., e_n)$ für alle $e_i \in Einträge(e)$.

Definition 3-10: (Geschachtelte) Reiseroute $\mathfrak{R}$ eines Agenten

Die Reiseroute $\mathfrak{R}$ eines Agenten ist ein Spezialfall eines *R-Eintrages* mit dem Typ *r* (“Reiseroute”) und der Vorbedingung *wahr*:

$$\mathfrak{R}= (\text{wahr}, \{e_1, e_2, ..., e_n\}, \boldsymbol{P}, r)$$

vereinfacht (analog zu Definition 3-1):

$$\mathfrak{R}= (\{e_1, e_2, ..., e_n\}, \boldsymbol{P})$$

Eine Reiseroute $\mathfrak{R}$ bildet einen Baum mit $\mathfrak{R}$ als Wurzel und den in der Reiseroute enthaltenen *B-Einträgen* als Blättern. Es ist daher möglich, die bei Bäumen üblichen Bezeichnungen *Knoten* für *R-Einträge*, *Blätter* für *B-Einträge*, *Kinder* eines *R-Eintrages* für die in einem *R-Eintrag* (direkt) enthaltenen Einträge, *Nachfahren* eines *R-Eintrages* für die rekursive Hülle der Kinder, *Vater* eines Eintrages für den *R-Eintrag*, in dem der Eintrag enthalten ist, *Vorfahren* für alle *R-Einträge* auf dem Weg zur Wurzel und *Geschwister* eines Eintrages *e* für die Einträge, die denselben Vater wie *e* haben, zu verwenden.

Das folgende Beispiel verdeutlicht diese Definitionen.

Beispiel 3-6. Spezifikation der Reiseroute $\mathfrak{R}$ aus dem vorherigen Beispiel mittels einer geschachtelten Reiseroute. Die *B-Einträge* $e_{i+1}, ..., e_{2i}$, welche die Verhandlungsrunde beschreiben, müssen nach den *B-Einträgen* $e_1, ..., e_i$, die das anfängliche Einholen der Angebote beschreiben, ausgeführt werden:

$e_k = (wahr, n_k, holeAngebot)$ für $k = 1 ... i$

$e_{i+k} = (wahr, n_k, verhandleAngebot)$ für $k = 1 ... i$

$e_{2i+1} = (D(e_1) \wedge D(e_2) \wedge ... \wedge D(e_i), \{e_{i+1}, e_{i+2}, ..., e_{2i}\}, \emptyset, \text{Typ})$

$\mathfrak{R}$ = (*wahr*, $\{e_1, ..., e_i, e_{2i+1}\}$, ∅, *r*) bzw. $\mathfrak{R}$ = ($\{e_1, ..., e_i, e_{2i+1}\}$, ∅)

Der *R-Eintrag* e_{2i+1} spezifiziert, daß mit der Ausführung der in ihm enthaltenen *B-Einträge* e_{i+1}, ..., e_{2i} erst begonnen werden darf, nachdem die Einträge e_1, ..., e_i alle ausgeführt wurden. Der Einsatz des *R-Eintrages* ermöglicht es also, die komplexe Vorbedingung "zentral" an einer Stelle zu spezifizieren mit der Folge, daß alle *B-Einträge* nur die triviale Vorbedingung *wahr* besitzen. Bemerkenswert ist, daß die Einträge e_{i+1}, ..., e_{2i} durch die Verwendung des eingeschachtelten *R-Eintrages* e_{2i+1} gar nicht mehr direkt sondern nur noch indirekt in der Reiseroute $\mathfrak{R}$ enthalten sind (d.h. *Einträge*($\mathfrak{R}$) $\cap \{e_{i+1}, e_{i+2}, ..., e_{2i}\} = \emptyset$).

Für die Ausführung einer geschachtelten Reiseroute sind prinzipiell zwei verschiedene Semantiken denkbar: Wenn in der Reiseroute in Beispiel 3-6 ein weiterer *B-Eintrag* e_{2i+2} mit der trivialen Vorbedingung *wahr* existieren würde (zum Beispiel ein Eintrag zum Ausführen einer Geldüberweisung auf einer Bank), könnte dieser Eintrag dann verschränkt mit der Ausführung des R-Eintrages e_{2i+1} erfolgen (d.h. nachdem mit der Verhandlung der Angebote begonnen wurde jedoch bevor das Nachverhandeln der Angebote in der zweiten Runde beendet ist)? In diesem Beispiel und in vielen anderen Anwendungen ist dies sicher möglich, in wiederum anderen Anwendungen jedoch nicht. Wenn die Ausführung der Einträge von e_{2i+1} beispielsweise innerhalb einer einzigen Transaktion geschehen müßten, dann wäre es im Sinne einer möglichst kurzen Blockierung der betroffenen Ressourcen wünschenswert, daß alle diese Einträge an einem Stück ausgeführt werden. Um diese beiden Semantiken zu unterstützen, werden zwei weitere Typen von *R-Einträgen* unterschieden:

Definition 3-11: *Offener R-Eintrag e*=(*Vorbedingung*, $\{e_1, ..., e_n\}$, ***P***, *o*)

Die Ausführung eines solchen *R-Eintrages e* darf mit der Ausführung von *B-Einträgen*, welche nicht zu den Nachfahren von *e* gehören (aber durchaus zu anderen R-Einträgen gehören dürfen), verschränkt werden.

Hierbei kann theoretisch der – auf semantisch unsaubere Spezifikation der Vorbedingungen zurückzuführende – Fall auftreten, daß die *Vorbedingung* des *R-Eintrages e* oder die Vorbedingung eines seiner Vorfahren während der Ausführung des *R-Eintrages* zu *falsch* wird (durch die Ausführung von *B-Einträgen*, welche nicht zu den Nachfahren von *e* gehören). Die Ausführung von *e* wird dadurch jedoch nicht gestoppt, da die *Vorbedingung* nur zum Start der Ausführung von *e* relevant ist. Das folgende abstrakte Beispiel illustriert kurz die Problematik:

Beispiel 3-7. Die Ausführung zweier Mengen $e_1, e_2, ..., e_m$ und $e_{m+1}, ..., e_n$ von Basis-Einträgen kann beliebig überlappen, mit der Ausführung der Menge $e_1, e_2, ..., e_m$ soll jedoch nur begonnen werden, wenn der Basis-Eintrag e_{n+1} noch nicht ausgeführt wurde:

e_i = (*wahr*, n_i, m_i) für $i = 1 ... n+1$

e_{n+2} = ($\neg D(e_{n+1})$, $\{e_1, e_2, ..., e_m\}$, ∅, *o*)

e_{n+3} = (*wahr*, $\{e_{m+1}, e_{m+2}, ..., e_n\}$, ∅, *o*)

$\mathfrak{R}$ = (*wahr*, $\{e_{n+1}, e_{n+2}, e_{n+3}\}$, ∅, *r*)

Mit der Ausführung der *B-Einträge* e_1, e_2, ..., e_m darf nur begonnen werden, solange e_{n+1} noch nicht ausgeführt wurde. Da e_{n+2} ein offener *R-Eintrag* ist, kann e_{n+1} ausgeführt werden nachdem mit der Ausführung der Einträge in e_{n+2} begonnen wurde aber bevor der letzte Eintrag von e_{n+2} ausgeführt wurde. Obwohl hierdurch die Vorbedingung von e_{n+2} zu *falsch* wird, werden die noch nicht ausgeführten Einträge von e_{n+2} vollends ausgeführt. Da sowohl e_{n+2} als auch e_{n+3} offene *R-Einträge* sind, ist die Ausführungsreihenfolge der *B-Einträge* e_1, ..., e_n beliebig.

Definition 3-12: *Geschlossener R-Eintrag e*=(*Vorbedingung*, $\{e_1, ..., e_n\}$, ***P***, *g*)

Während der Ausführung eines solchen *R-Eintrages* dürfen nur dessen Nachfahren ausgeführt werden.

Sobald also der erste *B-Eintrag*, welcher zu den Nachfahren von *e* gehört, ausgeführt wurde, dürfen nur noch Nachfahren von *e* ausgeführt werden, bis für alle *B-Einträge*, welche Nachfahren von *e* sind, gilt, daß sie entweder ausgeführt worden sind oder daß sie nicht mehr ausgeführt werden können (genauere Definition erfolgt weiter unten).

Beispiel 3-8. Ein Agent soll eine Reise buchen und Informationen zum Urlaubsort sammeln. Da gute Erfahrungen mit der Fluggesellschaft American Airlines und dem Autovermieter Avis gemacht wurden, sollen bei diesen Firmen Flug und Mietauto gebucht werden. Das Hotel für den Urlaubsort kann bei einer Hotelagentur gebucht werden.
Der Flug wird zuerst gebucht, damit die Termine der Reise feststehen. Danach können Hotel und Auto in beliebiger Reihenfolge gebucht werden:

Flug = (*wahr*, *AmericanAirlines*, *bucheFlug*)
Auto = (*D*(Flug), *Avis*, *bucheAuto*)
Hotel = (*D*(*Flug*), *HotelAgentur*, *bucheHotel*)

Um sicherzustellen, daß entweder alle drei oder keine der Buchungen erfolgreich sind, erfolgen die Buchungen innerhalb einer Transaktion.
Um genauere Information über den Urlaubsort und die nähere Umgebung zu erlangen, soll der Agent noch bei 3 Tourismusinformationen Informationsmaterial besorgen:

ti1 = (*wahr*, *TourismusInformation1*, *holeInfos*)
ti2 = (*wahr*, *TourismusInformation2*, *holeInfos*)
ti3 = (*wahr*, *TourismusInformation3*, *holeInfos*)

Um die bei der Buchungstransaktion verwendeten Ressourcen nicht unnötig lange zu blockieren muß verhindert werden, daß während der Transaktion eine Tourismusinformation besucht wird. Dies geschieht am einfachsten dadurch, daß die komplette Transaktion in einen geschlossenen *R-Eintrag* verlagert wird:

Buchungen = (*wahr*, {*Flug*, *Auto*, *Hotel*}, ∅, *g*)

Die komplette Reiseroute ergibt sich dann zu

$\Re$ = (*wahr*, {*Buchungen, ti1, ti2, ti3*}, ∅, *r*)

Speziell durch die Einführung der offenen *R-Einträge* ist das Prädikat $D(e)$ (mit der Bedeutung Eintrag e komplett ausgeführt) alleine nicht mehr ausreichend zur Spezifikation von Vorbedingungen, wie das folgende Szenario veranschaulicht: Eine Reiseroute enthält unter anderem zwei *R-Einträge* e_1 und e_2 die nicht verschränkt ausgeführt werden dürfen. Andere Einträge der Reiseroute dürfen jedoch verschränkt mit e_1 bzw. e_2 ausgeführt werden. Dies bedeutet, daß e_1 und e_2 offen sein müssen und die sequentielle Ausführung von e_1 und e_2 durch die Vorbedingungen erzwungen werden muß. Die Vorbedingungen müssen daher festlegen, daß mit der Ausführung von e_1 begonnen werden darf, falls e_2 entweder schon komplett ausgeführt wurde oder falls noch keiner der Nachfahren von e_2 ausgeführt wurde (und umgekehrt). Aus diesem Grunde wird ein weiteres Prädikat $S(e)$ (von engl.: started) benötigt, welches angibt, ob mit der Ausführung eines Eintrages bereits begonnen wurde. Die rekursive Definition von $S(e)$ ist gegeben durch

Definition 3-13: Prädikat $S(e)$

$$S(e) \equiv wahr \Leftrightarrow$$
$$((e \text{ ist } \textit{B-Eintrag}) \wedge (e \text{ wurde schon ausgeführt})) \vee$$
$$((e \text{ ist } \textit{R-Eintrag}) \wedge (\exists e_i \in \textit{Einträge}(e)\text{: } S(e_i)))$$

Die Definition sagt aus, daß $S(e)$ genau dann wahr ist, wenn entweder e ein *B-Eintrag* ist und e schon ausgeführt wurde oder e ein *R-Eintrag* ist und mit der Ausführung einer der Einträge aus *Einträge*(e) bereits begonnen wurde. Der Fall, daß ein *B-Eintrag* im Moment der Prädikatauswertung ausgeführt wird, muß hierbei nicht beachtet werden, da die Auswertung der Vorbedingungen (und dadurch der Prädikate) immer nur zum Migrationszeitpunkt (d.h. zwischen den Ausführungen von *B-Einträgen*) stattfindet. Analog zu $d(e)$ wird die Funktion s(e) definiert:

Definition 3-14: Funktion $s(e)$

$$s: e \rightarrow \{0,1\}$$
$$s(e) \equiv 1 \Leftrightarrow S(e) \equiv wahr$$

Jetzt kann auch die endgültige Definition des Prädikates $D(e)$ gegeben werden:

Definition 3-15: Prädikat $D(e)$

$$D(e) \equiv wahr \Leftrightarrow$$
$$((e \text{ ist } \textit{B-Eintrag}) \wedge (e \text{ wurde schon ausgeführt})) \vee$$
$$((e \text{ ist } \textit{R-Eintrag}) \wedge$$
$$(\forall e_i \in \textit{Einträge}(e)\text{: } (D(e_i) \vee (\neg S(e_i) \wedge (\textit{Vorbedingung}(e_i) = \textit{falsch})))))$$

Während der erste Teil der Definition, welcher $D(e)$ für einen *B-Eintrag e* definiert, mit Definition 3-3 übereinstimmt, bedarf der zweite Teil, welcher $D(e)$ für einen *R-Eintrag e* definiert, der Erläuterung. Um für einen *R-Eintrag* die Entscheidung treffen zu können, daß er komplett ausgeführt wurde, müssen dessen einzelne Einträge die Bedingung erfüllen, daß sie entweder komplett ausgeführt wurden (wird rekursiv getestet) oder daß sie nicht mehr ausgeführt werden können. Notwendige Bedingung für die Entscheidung, daß keiner dieser noch nicht ausgeführten Einträge noch ausgeführt werden kann ist, daß deren Vorbedingungen alle zu *falsch* ausgewertet werden. Dies ist für *B-Einträge* auch die hinreichende Bedingung. Bei *R-Einträgen* existiert jedoch noch die Möglichkeit, daß deren Ausführung schon begonnen hat und (trotz nicht erfüllter Vorbedingung, siehe oben) noch nicht beendet ist. Deshalb muß hier zusätzlich auch noch gelten, daß mit der Ausführung des Eintrages noch nicht begonnen wurde.

Da die Vorbedingung eines Eintrages nur von seinen Geschwistern abhängt, kann das Prädikat $D(e)$ immer "lokal" und eindeutig berechnet werden. Die Anforderung, $D(e)$ immer lokal und vor allem eindeutig berechnen zu können ist auch der Grund, weshalb keine anwendungsspezifischen Prädikate in den Vorbedingungen erlaubt sind. Wären solche anwendungsspezifischen Prädikate erlaubt, könnte bei einem offenen *R-Eintrag e* nur dann auf $D(e)$=*wahr* entschieden werden, wenn alle *B-Einträge* seiner Nachkommen ausgeführt wurden. Hätte nämlich einer dieser *B-Einträge* ein anwendungsspezifisches Prädikat als Vorbedingung und wäre er noch nicht ausgeführt worden, könnte diese Vorbedingung jederzeit durch Änderungen im Agentenzustand erfüllt werden (somit wäre die Entscheidung $D(e)$=*wahr* nur am Ende der Ausführung des Agenten möglich).

Die notwendigen Änderungen an der EBNF der Vorbedingungen zeigt Abbildung 3-4:

Prädikat	=	("*D*(" Eintrag ")" | "*S*(" Eintrag ")").
Funktion	=	("*d*(" Eintrag ")" | "*s*(" Eintrag ")").

Abbildung 3-4. Änderungen an der Syntax einer Vorbedingung in EBNF

Mit diesen Definitionen kann nun schließlich auch definiert werden, unter welchen Bedingungen ein *B-Eintrag* einer Reiseroute ausgeführt werden kann:

Definition 3-16: Ausführbarkeit eines *B-Eintrages* $e \in$ *Nachfahren*($\Re$)

Ein *B-Eintrag e*, welcher in einer Reiseroute $\Re$ enthalten ist, kann ausgeführt werden wenn gilt:

(*Vorbedingung*(e)=*wahr*) $\wedge$
($\forall$*Vorfahren* e_i von e: ((*Vorbedingung*(e_i)=*wahr*) $\vee$ ($S(e_i)$=*wahr*))) $\wedge$
($\forall$geschlossenen Einträge e_j aus der Menge der *Nachfahren* von $\Re$:
$((S(e_j) \wedge \neg D(e_j)) \Rightarrow e_j$ ist *Vorfahr* von e))

Um einen *B-Eintrag* ausführen zu können muß nicht nur dessen Vorbedingung erfüllt sein (erster Teil der Bedingung aus Definition 3-16), sondern der Zustand der Reiseroute muß mit in Betracht gezogen werden. Offensichtlich ist, daß die Ausführbarkeit vom Zustand der Vorfahren des *B-Eintrages abhängt*: entweder muß deren Vorbedingung erfüllt sein oder es muß schon mit ihrer Ausführung begonnen worden sein (zweiter Teil der Bedingung aus Definition 3-16). Weniger offensichtlich ist, daß beachtet werden muß, ob momentan geschlossene *R-Einträge* ausgeführt werden (ein (geschlossener) *R-Eintrag e* wird dann momentan ausgeführt, falls $S(e) \wedge \neg D(e)$ zu *wahr* ausgewertet wird). Ist dies der Fall, dann müssen alle momentan ausgeführten geschlossenen *R-Einträge* Vorfahren des *B-Eintrages* sein (dritter Teil der Bedingung aus Definition 3-16).

Die Flexibilität und Mächtigkeit des vorgestellten Konzeptes demonstriert das folgende Beispiel.

3.2.3 Beispiel einer komplexen Reiseroute

Das in dem Beispiel betrachtete Szenario ist die Organisation einer Konferenzreise. Der hierfür zuständige Agent muß dazu seinen Auftraggeber bei der Konferenz als Teilnehmer registrieren, einen Flug, ein Auto und die Unterbringung buchen, bei Touristeninformationen allgemeine Informationen über touristische Attraktionen und Veranstaltungen am Konferenzort sammeln und letztendlich das Ergebnis seiner Tätigkeit beim Auftraggeber abliefern. Um die Reiseroute zusammenzustellen, benötigt der Agent auftragsspezifische Informationen von seinem Auftraggeber (welche Konferenz, wo muß registriert werden, Aufenthaltszeit am Konferenzort, etc.), allgemeine Informationen aus dem Benutzerprofil des Auftraggebers (welche Fluggesellschaften werden bevorzugt, etc.) und andere Informationen, welche über Informationsdienste bereitgestellt werden (wo können Flüge gebucht werden, etc.). Nachdem die Informationen vorliegen, kann in diesem Fall die Reiseroute auch schon komplett spezifiziert werden.

Da die Konferenzgebühren bei der Registrierung per (elektronischem) Bankscheck bezahlt werden müssen, muß der Agent zuerst einen Scheck bei einer der Banken des Benutzers abholen, bevor er die Registrierung bei der Konferenz durchführen kann. Bei den Banken hat er die Auswahl zwischen der Barclays Bank oder der CityBank. Das Abholen des Schecks und die Registrierung spezifizieren die folgenden Einträge:

$bank1 = (\neg D(bank2), Barclays, holeScheck)$
$bank2 = (\neg D(bank1), CityBank, holeScheck)$
$Konferenz = (D(bank1) \vee D(bank2), IEEE, RegistriereFürTeilnahme)$

Der Auftraggeber bevorzugt die Barclays Bank. Deshalb wird (*bank1*, *bank2*) in die Prioritätsrelation des *R-Eintrages* aufgenommen, der *bank1* und *bank2* enthält.

Um das Beispiel in überschaubaren Grenzen zu halten werden hier nur zwei Fluggesellschaften und zwei Autovermieter verwendet. Hierbei bestehen jeweils zwischen einer Fluggesellschaft

und einem Autovermieter Verträge, die den Kunden Rabatte gewähren, wenn beim jeweils anderen Vertragspartner der Flug gebucht bzw. das Auto gemietet wird. Aus diesem Grund werden Buchungen nur in den Kombinationen AmericanAirlines/Avis (was die bevorzugte Kombination ist) und AirCanada/Hertz gemacht. Für jede dieser Firmen gibt es mehrere Geschäftsstellen, welche alternativ besucht werden können. Der Einfachheit halber werden das Buchen des Fluges und des Autos in einem *R-Eintrag* "versteckt":

FlugAuto= (*wahr*, {*AmericanAvis*, *CanadaHertz*},
{(*AmericanAvis*, *CanadaHertz*)}, *o*)

Da bei der ersten Alternative AmericanAirlines/Avis die Flugbuchung und die Reservierung des Autos in einer Transaktion durchgeführt werden muß, wird hierfür ein geschlossener *R-Eintrag* verwendet. AmericanAirlines hat drei Niederlassungen, Avis hat deren zwei. Um den Rabatt gewährt zu bekommen, muß der Flug vor dem Auto gebucht werden:

AmericanAvis = (¬*S*(*CanadaHertz*), {*American1*, *American2*, *American3*, *Avis1*,
Avis2},∅, *g*)
American1 = (¬(*D*(*American2*) ∨ *D*(*American3*)), *AmericanAirlines1*, *bucheFlug*)
American2 = (¬(*D*(*American1*) ∨ *D*(*American3*)), *AmericanAirlines2*, *bucheFlug*)
American3 = (¬(*D*(*American1*) ∨ *D*(*American2*)), *AmericanAirlines3*, *bucheFlug*)
Avis1 = ((*D*(*American1*)∨*D*(*American2*)∨*D*(*American3*))∧¬*D*(*Avis2*),
AvisNode1, *bucheAuto*)
Avis2 = ((*D*(*American1*)∨*D*(*American2*)∨*D*(*American3*))∧¬*D*(*Avis1*),
AvisNode2, *bucheAuto*)

Da *CanadaHertz* ein offener *R-Eintrag* ist (siehe weiter unten), muß in der Vorbedingung von *AmericanAvis* mit dem Prädikat *S*(...) gearbeitet werden, um ein gleichzeitiges Buchen bei beiden Gesellschaften zu verhindern. Sollte es noch mehr alternative Fluggesellschafts- oder Autovermietungsniederlassungen geben, ist natürlich auch hier die Verwendung zusätzlicher R-Einträge sowohl für die Fluggesellschaften als auch die Autovermieter ratsam. Bei AirCanada und Hertz gibt es jeweils nur zwei Niederlassungen. Auch hier muß der Flug zuerst gebucht werden:

CanadaHertz= (¬*D*(*AmericanAvis*), {*Canada1*, *Canada2*, *Hertz1*, *Hertz2*}, ∅, *o*)
Canada1 = (¬*D*(*Canada2*), *AirCanada1*, *bucheFlug*)
Canada2 = (¬*D*(*Canada1*), *AirCanada2*, *bucheFlug*)
Hertz1 = ((*D*(*Canada1*)∨*D*(*Canada2*))∧¬*D*(*Hertz2*), *HertzNode1*, *bucheAuto*)
Hertz2 = ((*D*(*Canada1*)∨*D*(*Canada2*))∧¬*D*(*Hertz1*), *HertzNode2*, *bucheAuto*)

In diesem Falle reicht es, daß *CanadaHertz* in der Vorbedingung mit dem Prädikat *D*(...) arbeitet, da *AmericanAvis* ein geschlossener *R-Eintrag* ist. Wird zuerst ein Eintrag von *AmericanAvis* ausgeführt, dann wird dieser *R-Eintrag* zuerst komplett fertig ausgeführt bevor mit irgend einem anderen Eintrag weitergemacht wird. Für *CanadaHertz* ist damit die Vorbedingung *falsch*. Wird jedoch zuerst ein Eintrag von *CanadaHertz* ausgeführt, dann wird sofort die Vorbedingung für

AmericanAvis falsch. Somit ist sichergestellt, daß nur einer dieser beiden *R-Einträge* abgearbeitet wird.

Um das Hotel zu buchen, muß der Agent die genauen Daten für die Übernachtung wissen. Aus diesem Grund kann das Buchen des Zimmers erst geschehen, nachdem der Flug bereits gebucht wurde. Aus Gründen der Übersichtlichkeit wurden auch hier nur zwei Agenturen zum Buchen des Hotels berücksichtigt:

$Hotel = (D(FlugAuto), \{hotelAgentur1, hotelAgentur2\}, \emptyset, o)$
$hotelAgentur1 = (\neg D(hotelAgentur2), HotelAgentur1, bucheZimmer)$
$hotelAgentur2 = (\neg D(hotelAgentur1), HotelAgentur2, bucheZimmer)$

Es gibt insgesamt fünf verschiedene Touristik-Zentren, welche verschiedene Arten an Informationen bieten. Bevor der Agent diese Zentren besucht, muß er zuerst den Flug gebucht haben, um die genaue Zeitspanne zu kennen, für welche er Informationen sammeln muß:

$ti1 = (D(FlugAuto) \wedge \neg D(home), TourismusInformation1, holeInfos)$
$ti2 = (D(FlugAuto) \wedge \neg D(home), TourismusInformation2, holeInfos)$
$ti3 = (D(FlugAuto) \wedge \neg D(home), TourismusInformation3, holeInfos)$
$ti4 = (D(FlugAuto) \wedge \neg D(home), TourismusInformation4, holeInfos)$
$ti5 = (D(FlugAuto) \wedge \neg D(home), TourismusInformation5, holeInfos)$

Bevor der Agent zu seinem Heimatort zurückkommt, um seinem Auftraggeber die Ergebnisse zu übergeben, muß er alle aufgeführten Arbeiten erledigen. Ausnahme hierbei ist der Besuch der Touristik-Informationen. Sollten hier nicht alle erreichbar sein (z.B. wegen Rechner-/Netzwerkausfällen) ist es ausreichend, wenn der Agent mindestens drei davon besucht. Dies wird im *R-Eintrag home* spezifiziert:

$home = (D(Konferenz) \wedge D(Hotel) \wedge (3 \leq d(ti1)+d(ti2)+d(ti3)+ d(ti4)+d(ti5)),$
$BenutzerRechner, berichteErgebnisse)$

Um jedoch festzulegen, daß der Agent nach Möglichkeit trotzdem alle 5 Touristik-Informationen besucht, kann die Prioritätsrelation verwendet werden: wird den Touristik-Informationen eine größere Priorität als dem *home*-Eintrag zugeordnet, so werden diese auch – sofern möglich – alle besucht, bevor der *home*-Eintrag ausgeführt wird. Der zweite Teil der Vorbedingungen der Touristik-Informationen-Einträge stellt sicher, daß der Agent nach Beendigung seiner eigentlichen Aufgabe nicht noch den Besuch bei einer Touristik-Information nachholt, falls er diese vor dem Abliefern seiner Ergebnisse nicht besuchen konnte.

Die komplette Reiseroute wird spezifiziert mit

$$\mathfrak{R} = (wahr, \{bank1, bank2, Konferenz, FlugAuto, Hotel, ti1, ti2, ti3, ti4, ti5, home\},$$
$$\{(bank1, bank2), (ti1, home), (ti2, home), (ti3, home), (ti4, home),(ti5, home)\}, r)$$

Die erste Ebene und einen Teil der zweiten Ebene des Baums der möglichen Pfade, welche der Agent laut der spezifizierten Reiseroute durchlaufen kann, zeigt Abbildung 3-5. Die riesige Anzahl der Knoten im gesamten Baum zeigt die Mächtigkeit und Flexibilität des Reiseroutenkonzeptes: Wenn der Agent alle Touristik-Informationen besucht, muß er insgesamt lediglich 11

Knoten besuchen. Der Baum hat folglich 11 Ebenen. Während es in der ersten Ebene nur sieben und in der zweiten nur 26 Knoten gibt, umfaßt die dritte Ebene 136 und die vierte Ebene bereits über 900 Knoten. Der Agent hat also bei der Wahl des ersten Zielortes 7 Wahlmöglichkeiten, bei der Wahl des zweiten Zielortes im Durchschnitt ca. 4 Wahlmöglichkeiten, bei der Wahl des dritten Zielortes ca. 5 Wahlmöglichkeiten und bei der Wahl des vierten Zielortes durchschnittlich ca. 7 Wahlmöglichkeiten. Im weiteren Verlauf reduziert sich dies jedoch wieder, bis letztendlich beim letzten zu besuchenden Knoten nur noch der Heimatknoten zur Auswahl steht.

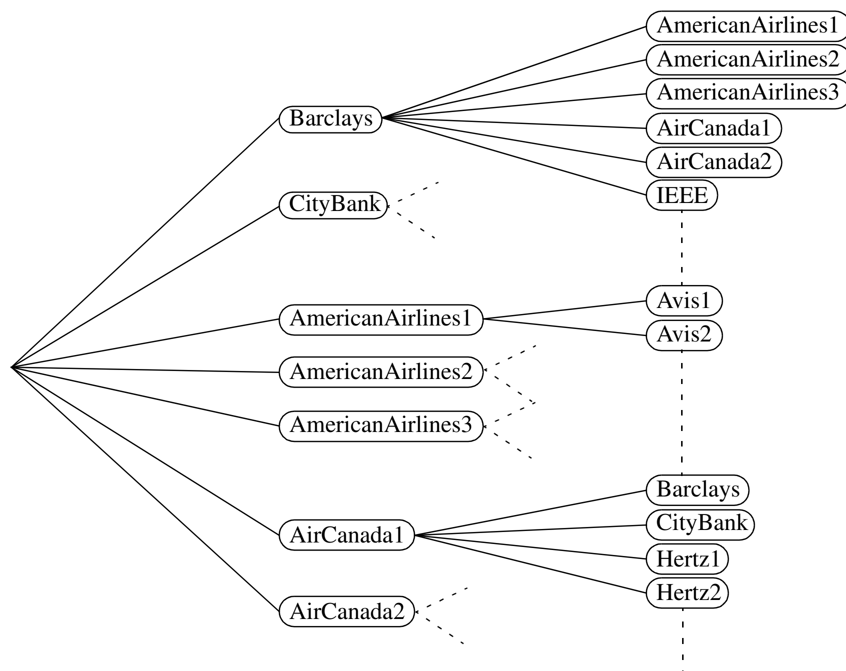


Abbildung 3-5. Baum der möglichen Pfade des Konferenzplanungsagenten

3.2.4 Diskussion

Wie vor allem das Beispiel des vorhergehenden Abschnittes zeigt, ist das in diesem Kapitel vorgestellte Reiseroutenkonzept der geschachtelten Reiseroute sehr flexibel und mächtig. Inwieweit das Ziel, bei der Migration mehrere alternative Migrationsziele zur Auswahl zu haben, erreicht werden kann hängt jedoch letztendlich von der Anwendung selbst ab. Je weniger die Aufgabe die Reihenfolge der notwendigen Schritte impliziert, desto stärker kann mittels dem hier vorgestellten Reiseroutenkonzept daraus Vorteil gezogen werden. Ist die Reihenfolge der notwendigen Schritte von der Aufgabenstellung jedoch fest vorgegeben, kann auch das Reiseroutenkonzept keine alternativen Migrationsziele zur Auswahl stellen.

Die Beispiele des Kapitels haben gezeigt, daß die manuelle Spezifikation einer Reiseroute und hierbei vor allem die Spezifikation der Vorbedingungen sehr aufwendig sein kann. BUSCHLE (1999) beschreibt eine Implementierung des Reiseroutenkonzeptes, welche den Anwendungsprogrammierer hierbei sehr stark unterstützt.

Kapitel 4

Genau-einmal Ausführung

Die zuverlässige, fehlertolerante Ausführung mobiler Agenten ist Voraussetzung für deren Einsatz in kommerziellen Anwendungen. In diesem Kapitel wird ein Protokoll entwickelt, welches die blockierungsfreie genau-einmal Ausführung mobiler Agenten garantiert.

Voraussetzung für die Entwicklung fehlertoleranter Anwendungen ist die Kenntnis der zu tolerierenden Fehler. Der folgende Abschnitt führt deshalb zuerst die den Algorithmen zugrunde liegenden System- und Fehlermodelle ein. Anschließend wird der Begriff der "genau-einmal Ausführung" im Kontext mobiler Agenten definiert. Das eigentliche Protokoll wird daraufhin in zwei Schritten entwickelt. Zuerst wird ein Basisprotokoll entwickelt, welches die genau-einmal Ausführung von Agenten realisiert. Da dieses Protokoll jedoch noch anfällig für die Blokkierung eines Agenten durch Rechner- und Netzwerkausfälle ist, wird es dann um eine Überwachungskomponente ergänzt. Hierbei überwachen mehrere Beobachterknoten, welche zusammen mit dem ausführenden Knoten eine *Stufe* (engl.: *stage*) bilden, die Ausführung des Agenten. Fällt der ausführende Knoten aus, kann einer der Beobachter die Ausführung des Agenten übernehmen. Daran anschließend wird die Nachrichtenkomplexität des entwickelten Protokolles untersucht und ein Algorithmus entwickelt, welcher durch geschickte Auswahl der Knoten einer Stufe die Nachrichtenkomplexität reduziert. Eine ausführliche analytische Bewertung des Protokolls und Leistungsmessungen einer in die Agentenplattform Mole integrierten Implementierung des Protokolls schließen das Kapitel.

Die in diesem Kapitel vorgestellten Protokolle und Mechanismen wurden erstmals in ROTHERMEL UND STRASSER (1997), ROTHERMEL UND STRASSER (1998) und STRASSER, ROTHERMEL UND MAIHÖFER (1998) veröffentlicht. Teilaspekte dieses Kapitels wurden in Diplomarbeiten von MAIHÖFER (1997), FRIEDEL (1998) und PAPOULIDIS (1999) erarbeitet.

4.1 System- und Fehlermodell

Um fehlertolerante verteilte Systeme zu entwickeln ist es notwendig, die Bestandteile des Systems und deren Fehler zu kennen. Einen detaillierten Überblick über Fehlertoleranz in verteilten Systemen gibt JALOTE (1994), dessen Terminologie und Klassifikationen weitestgehend (in deutscher Übersetzung) in der vorliegenden Arbeit übernommen wurden. Um das Verständnis zu erleichtern, wird auf die relevanten Begriffe an dieser Stelle kurz eingegangen.

Ein Systemmodell beschreibt die relevanten Bestandteile eines Systems, das Fehlermodell die im System möglichen Fehler. Ein (verteiltes) System kann hierbei aus zwei verschiedenen Blickwinkeln betrachtet werden. Die eine Sicht ist die der physischen Komponenten, aus denen das System besteht. Dies wird auch das *physische Systemmodell* genannt. Die andere Sicht ist die der Verarbeitung, d.h. die Sicht des Benutzers auf das System und die vom System angebotenen Dienste. Diese Sicht wird auch als das *logische Systemmodell* bezeichnet. Fehlertoleranz, vor allem in verteilten Systemen, beschäftigt sich häufig damit, die Eigenschaften bzw. Dienste des logischen Modells trotz *Ausfällen* (engl.: *failure*) in den Komponenten des physischen Systems sicherzustellen.

4.1.1 Fehlerklassifikation

Ein Ansatz der Klassifikation von Fehlern in verteilten Systemen beruht darauf, wie sich die physischen Komponenten des Systems bei einem Ausfall verhalten. Die Ausfälle (engl.: failure) der physischen Komponenten sind hierbei aus Sicht der Anwendung *Fehler* (engl.: *fault*) des Systems. CRISTIAN, AGHILI UND STRONG (1986) schlagen eine Klassifikation in die 4 Kategorien *Crash*, *Omission*, *Zeit* und *byzantinisch* vor:

Crash. Bei einem *Crash* hält eine Komponente sofort an. Sie macht keine falschen Zustandsübergänge und produziert keine falschen Ausgaben. In diesem Modell verhält sich eine Komponente also entweder gemäß Spezifikation oder hält an. Oftmals ist es nicht einfach festzustellen, ob eine Komponente ausgefallen ist. Aus diesem Grunde gibt es eine von SCHNEIDER (1984) vorgestellte Variante dieses Fehlertyps, bei welcher der Ausfall einer Komponente festgestellt werden kann. Die von Schneider vorgestellte Version eines Prozessors mit diesem Fehlertyp wird *fail stop processor* genannt.

Omission. Durch einen Fehler dieser Art reagiert eine Komponente sporadisch nicht auf Ein-/Ausgaben. Hier kann man unterscheiden zwischen einer *Receive-Omission*, bei der eine Komponente eine Nachricht nicht empfängt, und einer *Send-Omission*, bei der eine Komponente eine Nachricht nicht sendet.

Zeitfehler (engl.: *timing fault*). Ein Fehler dieser Art veranlaßt eine Komponente, zu früh oder zu spät zu reagieren.

Byzantinischer Fehler (engl.: *byzantine fault*). Ein Fehler, bei dem das Verhalten einer Komponente vollkommen zufällig von der Spezifikation abweicht (und dabei auch unerwartete/ falsche Ausgaben liefern kann).

Diese Fehler bilden eine Hierarchie. Hierbei ist der Crash-Fehler der einfachste, aber auch speziellste der Fehler, der byzantinische Fehler der allgemeinste der Fehler. Die aus SCHNEIDER (1984) übernommene Abbildung 4-1 zeigt, daß die allgemeineren Fehler die spezielleren Fehler als echte Teilmenge enthalten.

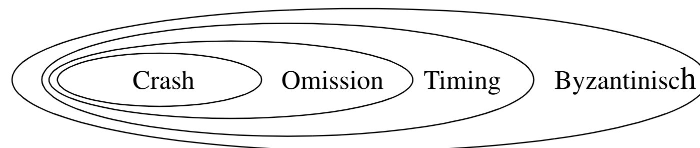


Abbildung 4-1. Fehlerklassifikation

4.1.2 Verwendetes Systemmodell

In diesem Abschnitt wird das in der vorliegenden Arbeit durchgängig verwendete Systemmodell vorgestellt. Ein verteiltes System besteht aus mehreren autonomen Knoten (d.h. Rechnern) und einem diese Knoten verbindenden Netzwerk.

Ein Knoten besteht aus einem Prozessor und privatem flüchtigen und stabilen Speicher. Pro Knoten können mehrere Programme ausgeführt werden. Ein Programm wird innerhalb eines Prozesses ausgeführt. Innerhalb eines Prozesses ist die nebenläufige Ausführung mehrerer Threads möglich. Prozesse auf einem Knoten können miteinander nur über Nachrichten kommunizieren. Um Alarme (z.B. Timeouts) verwalten zu können, besitzt jeder Knoten eine korrekte Uhr.

Die Kommunikation zwischen den Knoten erfolgt mittels Nachrichtenaustausch über das Netzwerk. Die Kommunikation erfolgt über Kommunikationskanäle. Zwischen je zwei Rechnern des Systems existiert ein Kommunikationskanal.

Alle Komponenten des Systems arbeiten asynchron, d.h. es existiert keine obere Zeitschranke für die Ausführung einer Sequenz von Anweisungen oder für die Übertragung einer Nachricht. Die in dieser Arbeit entwickelten Algorithmen funktionieren jedoch auch in synchronen Systemen, d.h. wenn obere Zeitschranken für das Ausführen von Anweisungen und die Nachrichtenübertragung existieren.

4.1.3 Verwendetes Fehlermodell

Analog zum Systemmodell wird im Fehlermodell zwischen Knotenfehlern und Netzfehlern unterschieden.

Knoten unterliegen lediglich Crash-Fehlern. Bei einem Knotenfehler hält der Knoten die Ausführung aller Programme an; der systembedingte Ausfall einer echten Teilmenge der Prozesse eines Knotens ist ausgeschlossen. Alle im flüchtigen Speicher liegenden Informationen (Ausführungszustand der laufenden Prozesse, Daten der Prozesse) gehen hierbei verloren. Auf stabilem Speicher abgelegte Daten gehen bei einem Knotenzusammenbruch nicht verloren. Der stabile Speicher selbst ist fehlerfrei (vgl. LAMPSON (1981)). Die Kommunikation zwischen Prozessen eines Knotens ist fehlerfrei.

Auch das Netzwerk unterliegt nur Crash-Fehlern. Hierbei treten jedoch nur Netzwerkpartitionierungen auf. Knoten innerhalb einer Partition können kommunizieren, Knoten unterschiedlicher Partitionen nicht. Die Kommunikation zwischen Knoten innerhalb einer Partition ist fehlerfrei.

Alle Komponenten des Systems arbeiten asynchron (d.h. es gibt keine obere Zeitschranke für die Ausführung von Operationen). Knoten- und Netzwerkausfall sind nicht unmittelbar erkennbar. Insbesondere sind Knoten- und Netzwerkausfall nicht unterscheidbar, da Kommunikation über das Netzwerk die einzige Möglichkeit der Kontaktaufnahme zu anderen Knoten ist. Dies bedeutet letztendlich, daß der Ausfall einer Komponente lediglich vermutet werden kann, Gewißheit besteht jedoch nicht.

Gängige Praxis in realen Systemen ist, daß eine Komponente bei einem Fehler repariert wird. Aus diesem Grunde wird angenommen, daß die obigen Fehler jeweils nur vorübergehend sind und die Komponenten nach Reparatur und/oder Neustart wieder zur Verfügung stehen. Der Programmzustand und der Zustand des flüchtigen Speichers eines Knotens wird nach Reparatur bzw. Neustart des Knotens nicht automatisch wieder hergestellt. Dieser Fehlertyp wird in AGUILERA, CHEN UND TOUEG (1998) vorgeschlagen und *Crash-Recovery Model* genannt.

4.2 Definition “Genau-einmal Ausführung”

Die Eigenschaft der genau-einmal Ausführung von Operationen in verteilten Systemen wurde erstmalig im Bereich der Client/Server-Interaktion definiert. SPECTOR (1982) spezifizierte die *nur-einmal-Typ-2* Fehlersemantik (engl.: *only-once-type-2*), nach SCHILL (1992A) später auch unter dem Begriff *genau-einmal* Fehlersemantik (engl.: *exactly-once*) bekannt, für die Ausführung eines einzelnen Aufrufes einer entfernten Prozedur. Im Kontext mobiler Agenten ist diese Definition jedoch nicht ausreichend. Anstatt der Ausführung einer einzelnen Prozedur muß hier die Ausführung der gesamten (in der Reiseroute spezifizierten) Aufgabe des Agenten, welcher die Ausführung einer ganzen Sequenz von Schritten des Agenten entspricht, betrachtet werden.

Sei $P=\{P_1, P_2, ..., P_n\}$ die Menge der durch eine gegebene Reiseroute spezifizierten möglichen Pfade eines Agenten, sei $L(P_i)$ die Anzahl der Knoten in Pfad $P_i=[N_{i,1}, N_{i,2}, ..., N_{i,L(Pi)}]$ und sei $S_{i,j}$ der auf dem j-ten Knoten $N_{i,j}$ des Pfades P_i auszuführende Schritt ($1 \leq i \leq n$, $1 \leq j \leq L(P_i)$). Die genau-einmal Ausführung ist dann wie folgt definiert:

Definition 4-1: Genau-einmal Ausführung eines mobilen Agenten

Ein Agent wird genau einmal ausgeführt $\Leftrightarrow$
((der Agent führt ausschließlich die zu **einem** Pfad $P_i \in P$ gehörigen Schritte
$S_{i,1}, ..., S_{i,L(Pi)}$ auf den ihnen zugewiesenen Knoten $N_{i,1}, ..., N_{i,L(Pi)}$ aus) $\wedge$
(der Agent führt Schritt $S_{i,j}$ vor Schritt $S_{i,j+1}$ aus ($1 \leq j < L(P_i)$)) $\wedge$
(jeder Schritt $S_{i,j}$ ($1 \leq j \leq L(P_i)$) wird genau einmal vollständig ausgeführt))

In dem in Abschnitt 3.1 vorgestellten Szenario würde eine Agentenplattform, welche die genau-einmal Ausführung mobiler Agenten garantiert, sicherstellen, daß der Butler-Agent letztendlich nur den Blumenladen, einen der Kino-Knoten und den dazugehörigen Restaurant-Knoten in einer der durch Abbildung 3-2 spezifizierten Reihenfolgen besucht und die dort auszuführenden Schritte genau einmal ausführt.

Es ist durch diese Definition jedoch nicht ausgeschlossen, daß der Agent beispielsweise zuerst auf einen der beiden Kino-Knoten transportiert wird, um die Karten zu kaufen, und erst beim Fehlschlagen dieser Aktion (z.B. wegen Rechnerausfalls) dann auf den anderen Kino-Knoten transportiert wird, um letztendlich dort die Karten zu kaufen. Die Agentenplattform muß in diesem Falle nur sicherstellen, daß der Agent wieder in den korrekten Zustand versetzt wird und daß auf dem Knoten, auf dem der Kartenkauf fehlschlug, die durch den Agenten verursachten Änderungen rückgängig gemacht werden.

Ebenfalls wird durch diese Definition nicht ausgeschlossen, daß der Agent die Reiseroute im Laufe seiner Ausführung ändert bzw. ergänzt, wobei schon abgearbeitete Teile nicht mehr verändert werden dürfen. Eine Agentenplattform, welche die genau-einmal Ausführung mobiler Agenten garantiert, stellt in diesem Fall sicher, daß nach Beendigung der Ausführung des Agenten (erst dann sind alle möglichen Pfade endgültig bekannt) die Definition 4-1 zutrifft.

4.3 Basisprotokoll

Die im vorangegangenen Abschnitt definierte genau-einmal Ausführung von mobilen Agenten kann auf einfache Weise durch den Einsatz von Transaktionen und stabilem (transaktionalem) Speicher erreicht werden. Die zugrundeliegende Idee veranschaulicht Abbildung 4-2.

Der mobile Agent wird zwischen der Ausführung zweier Schritte auf stabilem, transaktionalem Speicher gespeichert. Hierfür besitzt jeder Knoten eine *Eingangswarteschlange* (in der Abbildung mit Q_i bezeichnet), welche auf transaktionalem, stabilen Speicher gespeichert wird. Zur

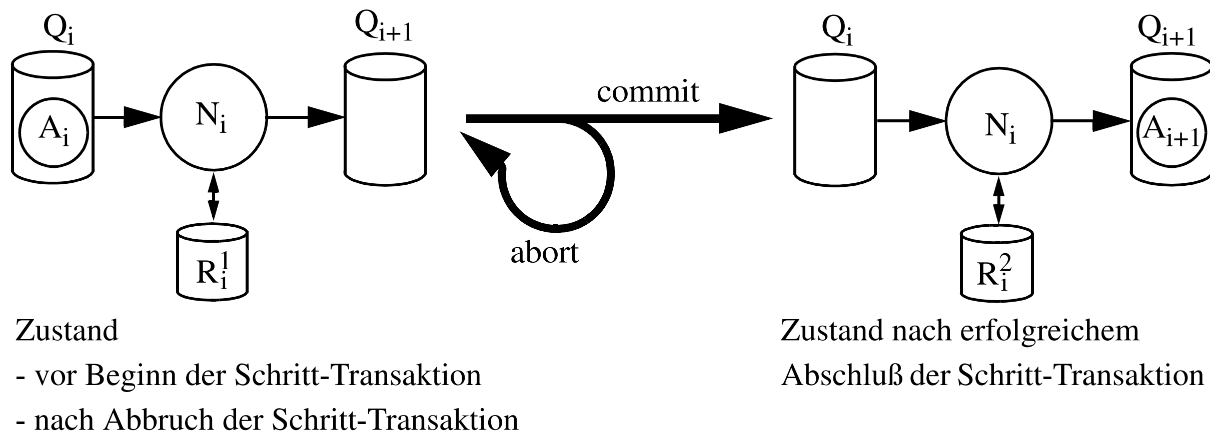


Abbildung 4-2. Genau-einmal Ausführung eines einzelnen Schrittes innerhalb einer Schritt-Transaktion

Ausführung des Agenten wird dieser aus der Eingangswarteschlange entnommen (d.h. gelesen und gelöscht), der auszuführende Schritt wird ausgeführt und der Agent schließlich in die Eingangswarteschlange jenes Knoten gestellt, auf dem der nächste Schritt auszuführen ist. Alle diese Aktionen werden im Kontext einer ACID-Transaktion (vgl. z.B. GRAY UND REUTER (1993)) ausgeführt, die vor dem Lesen des Agenten aus der Eingangswarteschlange begonnen und nach dem Schreiben in die Warteschlange des folgenden Knoten beendet wird. Hierdurch bilden die drei Aktionen (lesen, ausführen, schreiben) eine atomare Einheit. Da die Transaktion der Ausführung eines Schrittes dient, wird sie im folgenden auch als *Schritt-Transaktion* bezeichnet.

Unter der Voraussetzung, daß die durch den Agenten genutzten Dienste und Ressourcen transaktional sind, ist damit die genau-einmal Ausführung eines Schrittes garantiert. Ist die Transaktion erfolgreich, stellt das Transaktionsmanagement sicher, daß alle in der Transaktion durchgeführten Änderungen an den Ressourcen dauerhaft sind. Dies gilt nicht nur für die von dem mobilen Agenten genutzten Ressourcen und Dienste, sondern auch für das Löschen des Agenten aus der Eingangswarteschlange des lokalen Knotens und das Schreiben des Agenten in die Eingangswarteschlange des nächsten Knotens. Das Resultat einer erfolgreichen Schritt-Transaktion ist also, daß die Ressourcen und Dienste des den Schritt ausführenden Knotens die durch die Schrittausführung erzeugten Änderungen widerspiegeln und daß der Agent in dem durch den gerade ausgeführten Schritt erzeugten Zustand in der Eingangswarteschlange des Knotens liegt, auf dem der nächste Schritt auszuführen ist.

Schlägt die Ausführung einer Schritt-Transaktion fehl (z.B. durch Abbruch der Transaktion durch das Transaktionsmanagement oder durch Ausfall des den Schritt ausführenden Knotens), so sorgt das Transaktionsmanagement dafür, daß die durch die Schritt-Transaktion erzeugten Änderungen komplett rückgängig gemacht werden. In diesem Falle befindet sich also der Agent nach wie vor in der Eingangswarteschlange des ausführenden Knotens; die vom Agenten benutzten Ressourcen und Dienste befinden sich in demselben Zustand wie vor dem Beginn der

Schritt-Transaktion. Die Ausführung des Agenten wird fortgesetzt, indem die Schritt-Transaktion (bei Knotenausfall erst nach Neustart des Knotens) erneut gestartet wird.

Eine einfache Möglichkeit, dieses Protokoll zu implementieren, ist die Verwendung transaktionaler *Nachrichtenwarteschlangen* (engl.: *message queue*, vgl. z.B. GRAY UND REUTER (1993)) zur Realisierung der Eingangswarteschlangen der Knoten. Nachrichtenwarteschlangen bieten die Möglichkeit der asynchronen Kommunikation zwischen Prozessen, welche sich sowohl auf dem selben als auch auf unterschiedlichen Knoten befinden können. Der Sender einer Nachricht schreibt diese mittels einer *Put*-Operation in die Nachrichtenwarteschlange. Der Empfänger der Nachricht kann diese dann zu einem beliebigen späteren Zeitpunkt mittels einer *Get*-Operation aus der Nachrichtenwarteschlange lesen, wobei die Nachricht wahlweise aus der Nachrichtenwarteschlange entfernt wird oder für weiteres Lesen darin verbleibt. Wird bei transaktionalen Nachrichtenwarteschlangen, die gespeicherte Nachrichten auf stabilem Speicher ablegen, die Nachricht beim Lesen entfernt, dann garantieren transaktionale Warteschlangen unabhängig von auftretenden Fehlern die genau-einmal Auslieferung von Nachrichten an exakt einen Empfänger. Hierzu müssen die *Put*- und *Get*-Operationen innerhalb des Kontextes einer ACID-Transaktion aufgerufen werden.

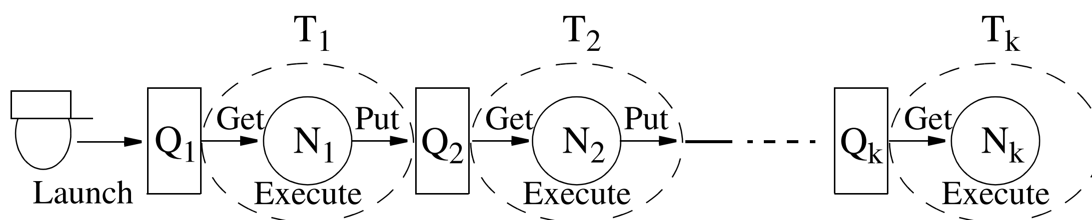


Abbildung 4-3. Genau-einmal Ausführung eines Agenten mit dem Basisprotokoll.

Abbildung 4-3 zeigt ein Beispiel für die genau-einmal Ausführung eines mobilen Agenten anhand einer Realisierung des Basisprotokolls mittels transaktionaler Nachrichtenwarteschlangen (Q_i ist die Eingangswarteschlange des Knoten N_i). In diesem Beispiel bewegt sich der Agent entlang des Pfades N_1, N_2, ..., N_k (auf dem einzelne Knoten auch mehrfach besucht werden können), welcher einer der durch die Reiseroute des Agenten gegebenen möglichen Pfade ist. Sobald der Agent in der ersten Eingangswarteschlange gespeichert ist (Q_1 in der Abbildung), ist garantiert, daß der Agent letztendlich genau einmal ausgeführt wird.

Algorithmus 4-1 zeigt die Pseudo-Code-Implementierung des Basisprotokolls, welches auf jeder Agentenplattform ausgeführt wird. Nach dem Start einer Schritt-Transaktion wird ein Agent aus der transaktionalen Nachrichtenwarteschlange gelesen und entfernt. Der durch den Agent auf dem Knoten auszuführende Schritt wird dann mittels *Execute* ausgeführt. Anschließend werden anhand der Reiseroute des Agenten die nächstmöglichen Schritte des Agenten be-

```
ForEach Agent a in NodeInputQueue q{
    Begin Transaction{ // step transaction
        q.ReadAndDestroy(Agent a)
        Execute(Agent a)
        nextPossibleSteps = QueryItinerary()
        if (nextPossibleSteps.notEmpty()){
            // execution not yet finished
            nextStep = ChooseOneOf(nextPossibleSteps)
            Write(Agent a) to NodeInputQueue of node on which nextStep takes place
            if (not successful){
                Abort Transaction
        }   }
    }Commit Transaction
}
```

Algorithmus 4-1. Basisprotokoll zur genau-einmal Ausführung mobiler Agenten

stimmt. Sind keine weiteren Schritte auszuführen, wird die Transaktion sofort mit *Commit* beendet und damit die Bearbeitung des Agenten abgeschlossen. Sind weitere Schritte auszuführen, wird der Agent mittels *Write* in die Eingangswarteschlange des Knotens, auf dem der nächste Schritt ausgeführt werden soll, geschrieben, bevor die Transaktion mittels *Commit* beendet wird. Stehen mehrere Schritte als mögliche nächste Schritte zur Auswahl, wird einer dieser Schritte ausgewählt. Mögliche Auswahlkriterien hierbei sind beispielsweise die momentane Erreichbarkeit der Zielknoten, Effizienz (kürzeste Route wählen,...) oder in der Reiseroute definierte Prioritäten.

Hiermit stellt der Basisalgorithmus sicher, daß Agenten *genau-einmal* gemäß Definition 4-1 ausgeführt werden: Die Ausführung der richtigen Schritte in der richtigen Reihenfolge wird durch das Auswahlverfahren des nächsten auszuführenden Schrittes sichergestellt. Die Ausführung innerhalb der Schritt-Transaktion stellt sicher, daß ein Schritt genau einmal ausgeführt wird und daß nach der Ausführung des Schrittes der Agent in die Eingangswarteschlange des den nächsten Schritt ausführenden Knotens geschrieben wird. Durch die Annahmen, daß alle ausgefallenen Komponenten wieder neu gestartet werden und stabiler Speicher seinen Inhalt nicht verliert, ist gesichert, daß letztendlich auch alle auszuführenden Schritte eines mobilen Agenten durch das Protokoll (genau einmal) ausgeführt werden.

Das vorgestellte Basisprotokoll hat zwei Nachteile, die nicht verschwiegen werden sollen. Das erste Problem ist, daß durch lang andauernde Schritte alle Ressourcen, auf die der Agent während des Schrittes zugreift, bis zum Ende des Schrittes blockiert sind. Abhilfe kann hier vom Entwickler eines Agenten geschaffen werden, indem lang andauernde Schritte möglichst in mehrere Schritte unterteilt werden.

Das zweite Problem ist, daß ein Agent durch den Ausfall von Komponenten beliebig lange blockiert werden kann. Während beispielsweise beim Aufruf eines Servers durch einen Client der Client den Server überwachen und bei Ausfall des Servers gegebenenfalls geeignete Maßnahmen treffen kann, geschieht die Ausführung eines Agenten autonom – es gibt keine “natürliche”

Instanz, welche die Ausführung des Agenten überwacht. Fällt also ein Knoten aus, so werden alle Agenten, die in der Eingangswarteschlange des Knoten liegen, erst dann weiter ausgeführt, wenn der Knoten neu gestartet wurde. Dies ist jedoch für viele Anwendungsgebiete, vor allem im Bereich des elektronischen Handels, nicht akzeptabel. Im folgenden Abschnitt wird daher das Basisprotokoll so erweitert, daß die Wahrscheinlichkeit, daß ein Agent durch ausgefallene Komponenten blockiert wird, drastisch reduziert wird.

4.4 Blockierungsfreies Protokoll

Das in diesem Abschnitt beschriebene Protokoll zur (annähernd) blockierungsfreien Ausführung mobiler Agenten ist komplex. Aus diesem Grund wird die Beschreibung in mehrere Teile untergliedert. Im ersten Teil wird ein Überblick über das gesamte Protokoll gegeben. In den darauf folgenden Teilen werden zuerst die einzelnen Komponenten des Protokolles sehr detailliert beschrieben und schließlich die Korrektheit des Protokolles diskutiert.

4.4.1 Überblick über das Protokoll

Um die Wahrscheinlichkeit, daß ein Agent durch den Ausfall von Komponenten in seiner Ausführung blockiert wird, zu verringern, wird das Basisprotokoll um das Konzept der *Stufen* (engl.: *stages*) erweitert. Analog zu SCHNEIDER (1997A) existiert für jeden auszuführenden Schritt eine (nicht-leere) Menge von Knoten, welche Stufe genannt wird. Im Gegensatz zu SCHNEIDER (1997A) wird der Agent jedoch nicht auf allen Knoten der Stufe ausgeführt. Ein Knoten einer Stufe, *Arbeiter* (engl.: *worker*) genannt, beginnt mit der Ausführung des Agenten. Die anderen Knoten der Stufe, *Beobachter* (engl.: *observer*) genannt, sind dafür zuständig, die Ausführung des Agenten durch den Arbeiter zu beobachten. Fällt der Arbeiter aus, wird dies von den Beobachtern entdeckt und ein neuer Arbeiter aus der Menge der Beobachter bestimmt. Die Bestimmung des initialen Arbeiters und – bei dessen Ausfall – von weiteren Arbeitern geschieht mittels einer Priorität, die jedem Knoten einer Stufe zugewiesen wird und die innerhalb der Stufe eindeutig sein muß. Neben der Wahl des Arbeiter mittels des Auswahl-Protokolles wird diese Priorität auch für das Votier-Protokoll (vgl. weiter unten) benötigt. Der Knoten mit der höchsten Priorität wird initialer Arbeiter der Stufe. Abbildung 4-4 zeigt die Ausführung eines Agenten in 2 Stufen. In der ersten Stufe S_1 führt der Knoten mit der höchsten Priorität den Agenten aus, die anderen 4 Knoten sind die Beobachter. In der zweiten Stufe S_2 fällt der Knoten mit der höchsten Priorität während der Ausführung des Agenten aus, so daß der Knoten mit der zweithöchsten Priorität die Ausführung übernehmen muß (und deshalb nur noch ein intakter Beobachter zur Verfügung steht).

Um im Falle des Ausfalles eines Arbeiters die Ausführung des Agenten übernehmen zu können, müssen auf allen Knoten einer Stufe die zwei folgenden Voraussetzungen erfüllt sein. Erstens muß der Agent (Daten, Code) von allen Knoten der Stufe aus zugreifbar sein. Dies wird sicher-

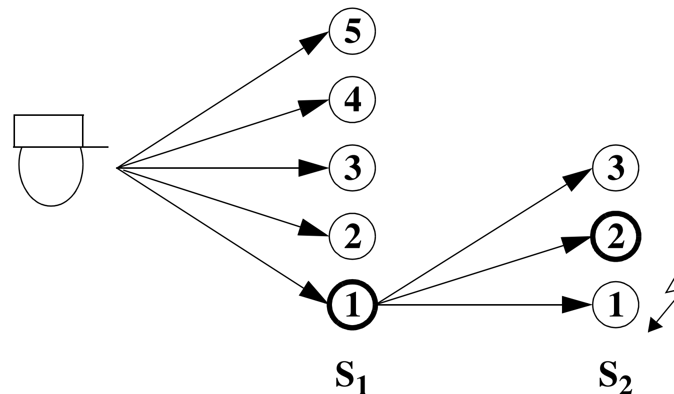


Abbildung 4-4. Ausführung eines Agenten in 2 Stufen.

gestellt, indem der Agent inklusive der Information über den Aufbau der Stufe (Knoten der Stufe und deren Prioritäten) in die Eingangswarteschlangen aller Knoten der Stufe geschrieben wird. Anhand der Informationen über den Aufbau der Stufe kann dann der Knoten mit der höchsten Priorität autonom feststellen, daß er der initiale Arbeiter ist und folglich mit der Ausführung des Agenten beginnen. Ebenso können die restlichen Knoten der Stufe autonom feststellen (indem sie die Informationen über die Stufe aus der Eingangswarteschlange lesen ohne sie zu löschen), daß sie Beobachter sind.

Die zweite Voraussetzung ist, daß alle Knoten der Stufe in der Lage sein müssen, den Agent auszuführen. Hierzu muß auf jeden Fall auf den Knoten der Stufe eine Agentenplattform installiert sein. Man kann dann zwei verschiedene Arten von Knoten in einer Stufe unterscheiden. *Reguläre Knoten* sind Knoten, auf welchen laut Reiseplan des Agenten zum momentanen Zeitpunkt eine Teilaufgabe ausgeführt werden kann. Diese Knoten bieten also die Möglichkeit, einen regulären Schritt des Agenten auszuführen. Hierbei sind die Teilaufgaben auf den verschiedenen Knoten im allgemeinen nicht dieselben. *Ausnahmebehandlungs-Knoten* sind Knoten, auf denen momentan keine Teilaufgabe des Agenten ausgeführt werden kann. Auf diesen Knoten wird der Agent nur dann ausgeführt, wenn keine regulären Knoten verfügbar sind. In diesem Fall hat der Agent die Möglichkeit, auf die Ausnahmesituation zu reagieren (Reiseroute ändern; auf Verfügbarkeit der regulären Knoten warten; u.s.w). Wird ein Agent auf einem Ausnahmebehandlungs-Knoten ausgeführt, wird dies in die Reiseroute als zusätzlicher Schritt eingefügt. Hierdurch wird die Ausnahmebehandlung einerseits in der Ausführungsgeschichte des Agenten dokumentiert und andererseits bleibt dadurch Definition 4-1 auch bei Ausführung auf Ausnahmebehandlungs-Knoten erfüllt.

Anhand des Szenariums aus Abschnitt 3.2.3, in dem der Agent eine Konferenzreise organisieren soll, kann man die zwei unterschiedlichen Knotentypen illustrieren. Da die Ausführung der Aufgabe dem Eigentümer sehr wichtig ist, soll die Blockierwahrscheinlichkeit des Agenten sehr gering sein. Das System entscheidet daher, daß pro Stufe 5 Knoten zur Verfügung stehen sollen (den Zusammenhang zwischen Anzahl der Knoten und der Blockierwahrscheinlichkeit erläutert Abschnitt 4.6). Aus dem in Abbildung 4-5 nochmals abgebildeten Baum der möglichen Pfade

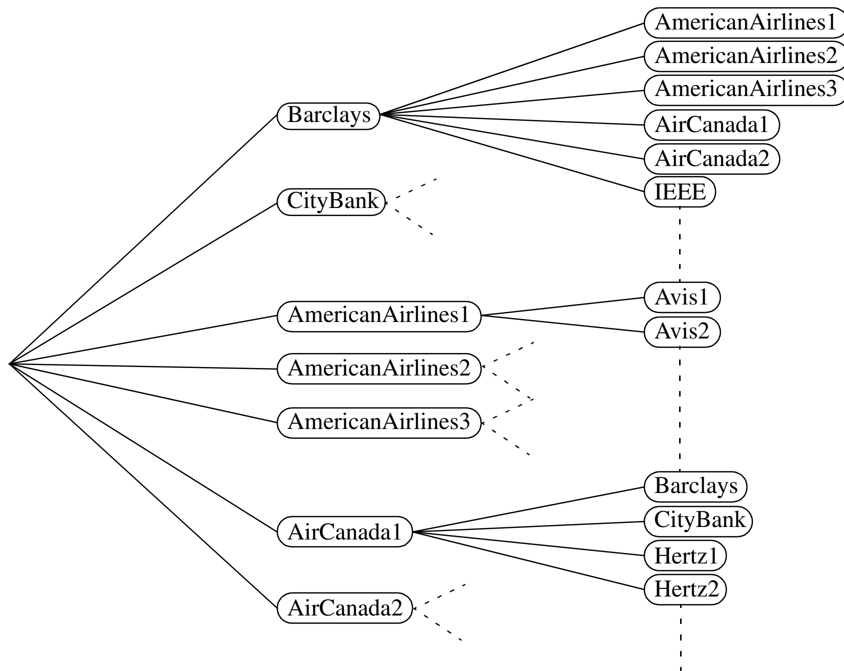


Abbildung 4-5. Baum der möglichen Pfade des Konferenzplanungsagenten

des Agenten ergibt sich, daß für den ersten Schritt des Agenten – und damit zur Bildung der ersten Stufe – 7 reguläre Knoten zur Auswahl stehen. Aus diesen 7 Knoten können also 5 ausgewählt werden. Wird der erste Schritt bei einer der beiden Banken (*Barclays* oder *CityBank*) ausgeführt, stehen auch für die Bildung der 2. Stufe ausreichend reguläre Knoten zur Verfügung. Wird der erste Schritt jedoch auf einem der Knoten von *AmericanAirlines* ausgeführt, dann stehen zur Bildung der zweiten Stufe nur zwei reguläre Knoten (*Avis1* und *Avis2*) zur Verfügung. Um auf die gewünschten 5 Knoten pro Stufe zu kommen, müssen nun noch drei beliebige Ausnahmebehandlungs-Knoten ausgewählt werden. Einen Algorithmus zur Bestimmung der Knoten einer Stufe stellt Abschnitt 4.5 vor.

Die Ausführung des Agenten durch den Arbeiter geschieht analog zur Ausführung des Agenten im Basisprotokoll. Algorithmus 4-2 und Abbildung 4-6 zeigen die Ausführung eines Agenten

```
Begin Transaction{ // step transaction
    inputQueue.ReadAndDestroy(Agent a, StageInfo S)
    Execute(Agent a)
    nextStage = computeNextStageUsingItinerary()
    if (nextStage.notEmpty()){
        // execution not yet finished
        Write(Agent a, StageInfo nextStage) to all
          inputQueues of nodes of next stage
        if (not successful for all nodes of next stage){
            try to correct or Abort Transaction
    }   }
}Commit Transaction
```

Algorithmus 4-2. Ausführung eines Agenten durch einen Arbeiter

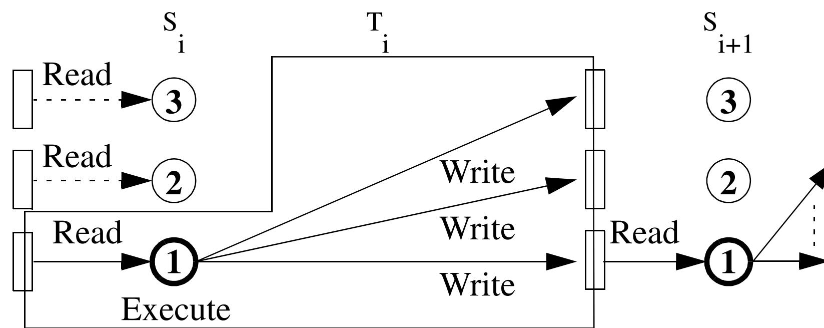


Abbildung 4-6. Transaktionale Ausführung eines mobilen Agenten in einer Stufe.

durch einen Arbeiter. Der Agent und die Stufeninformation werden nach dem Start der Schritt-Transaktion von stabilem Speicher gelesen und gelöscht und der Agent ausgeführt. Danach werden die Knoten der nächsten Stufe und deren Priorität berechnet und der Agent inklusive der Informationen über die nächste Stufe in die Eingangswarteschlangen der Knoten der nächsten Stufe geschrieben. Ist das Schreiben in diese Eingangswarteschlangen nicht auf allen Knoten erfolgreich – beispielsweise weil einer dieser Knoten ausgefallen oder durch eine Partitionierung nicht erreichbar ist – so gibt es zwei Korrekturmöglichkeiten. Eine Möglichkeit ist, die ausgefallenen bzw. nicht erreichbaren Knoten durch andere Knoten zu ersetzen. In dem Fall werden Agent und Stufeninformation auf diese zusätzlichen Knoten neu übertragen und die Stufeninformationen auf den anderen Knoten der nachfolgenden Stufe werden auf den neuen Stand gebracht. Falls dies nicht möglich ist oder falls nicht festgestellt werden kann, ob Daten auf einem Knoten angekommen sind oder nicht, muß die Transaktion abgebrochen werden. Ist der Transport jedoch erfolgreich, wird die Schritt-Transaktion mit Commit() beendet. Wichtig hinsichtlich der Blockierwahrscheinlichkeit des Agenten ist die Feststellung, daß die Zugriffe auf den stabilen Speicher durch die Beobachter einer Stufe (zum Lesen der Informationen zum Aufbau der Stufe) nicht innerhalb der Schritt-Transaktion geschieht. Würde dies innerhalb dieser Transaktion geschehen, würde dadurch die Wahrscheinlichkeit, daß die Beendigung der Transaktion durch einen Knotenausfall verzögert bzw. unmöglich wird, wesentlich erhöht. Die Einbeziehung aller Knoten der darauffolgenden Stufe in die Transaktion durch die Schreiboperation in deren Eingangswarteschlangen erhöht hingegen diese Wahrscheinlichkeit nur sehr unwesentlich, da die Verfügbarkeit der Knoten kurz vor dem Schreiben des Agenten in die Eingangswarteschlangen geprüft werden kann.

Der Start des Agenten geschieht innerhalb einer Starttransaktion, welche auf dem Knoten des Agentenbesitzers bzw. der Anwendung, welche den Agenten startet, ausgeführt wird. In dieser Starttransaktion hat der Agent die Möglichkeit, sich zu initialisieren und beispielsweise seine Reiseroute (partiell) aufzustellen. Nach der Initialisierungsphase werden die Knoten der ersten Stufe bestimmt, der Agent auf diese Knoten transportiert und die Starttransaktion beendet. Algorithmus 4-3 zeigt den Ablauf der Starttransaktion im Überblick. Sobald diese Starttransaktion

```
startAgent(Agent a, parameters){
    Begin Transaction{ // start transaction
        instantiate Agent a(parameters)
        ExecuteInitialisation(a)
        nextStage = computeNextStageUsingItinerary()
        if (nextStage.notEmpty()){
            // execution not yet finished
            Write(Agent a, StageInfo nextStage) to all
              inputQueues of nodes of next stage
            if (not successful for all nodes of next stage){
                try to correct or Abort Transaction
        }   }
    }Commit Transaction
    return completion state of transaction
}
```

Algorithmus 4-3. Starttransaktion

erfolgreich beendet ist, kann sich der Benutzer bzw. die den Agenten startende Anwendung sicher sein, daß der Agent blockierungsfrei genau einmal ausgeführt wird.

Im Gegensatz zum Basisprotokoll ist bei dem erweiterten Protokoll wichtig, daß sich der Agent zum Zeitpunkt des erfolgreichen Abschlusses der Transaktion tatsächlich auf den Knoten der nächsten Stufe befindet. Bei der Kommunikation mittels gängiger Nachrichtenwarteschlangen-Produkte (z.B. IBM MQSeries) ist beispielsweise nur garantiert, daß eine innerhalb einer Transaktion verschickte Nachricht auf jeden Fall ausgeliefert werden wird. Die Nachricht befindet sich nach dem Commit der Transaktion jedoch meist noch in einer lokalen Warteschlange des sendenden Knotens und wird asynchron zum Zielknoten transportiert. Fällt daher der Knoten zwischen Transaktionsende und Weiterbeförderung der Nachricht aus, wird die Nachricht erst nach Wiederanlauf des Knotens weitergeleitet. Für das Protokoll würde dies bedeuten, daß der Agent in diesem Falle ebenfalls bis zum Wiederanlauf des Knotens blockiert wäre. Aus diesem Grund ist die Verwendung gängiger Nachrichtenwarteschlangen-Produkte für die Implementierung dieses Protokolles nicht möglich. Die Implementierung des Protokolles hat auf jeden Fall sicherzustellen, daß vor dem Beenden der Schritt-Transaktion der Agent allen Eingangswarteschlangen der Knoten der nächsten Stufe (welche sich direkt auf den entsprechenden Knoten befinden und als Ressourcenmanager, d.h. als transaktionale Ressourcen, implementiert sind) übergeben wurde.

Um den Ausfall eines Arbeiters zu erkennen, wird ein *Beobachtungsprotokoll* verwendet. Sobald einer der Beobachter den Ausfall des Arbeiters feststellt, bestimmen die Beobachter der Stufe mittels des *Auswahlprotokolles* einen neuen Arbeiter. Da nach einem Knotenausfall die auf dem Knoten nicht vollständig ausgeführte Schritt-Transaktion zurückgesetzt wird, kann nun der neu bestimmte Arbeiter damit beginnen, den auf ihm auszuführenden Schritt wie oben beschrieben auszuführen. Beobachtungs- und Auswahlprotokoll werden detailliert in Abschnitt 4.4.3.1 und Abschnitt 4.4.3.2 beschrieben

Diese Lösung ist jedoch aus zwei recht offensichtlichen Gründen noch nicht vollständig: Erstens kann bisher nicht sicher festgestellt werden, ob die Schritt-Transaktion des Arbeiters nicht schon vor dessen Ausfall beendet war. Zweitens ist es nach unserem Fehlermodell und auch in der Realität nicht möglich, sicher zwischen Knoten- und Netzwerkausfall zu unterscheiden. Daher kann auch nicht sicher festgestellt werden, ob der aktuelle Arbeiter nun tatsächlich ausgefallen ist oder ob er zwar nicht mehr erreichbar ist, der Agent jedoch nach wie vor noch dort ausgeführt wird. Deshalb ist es möglich, daß innerhalb einer Stufe zwei oder mehr Arbeiter existieren, welche beide innerhalb einer Schritt-Transaktion einen Schritt des Agenten oder eine Ausnahmebehandlungsprozedur ausführen. Da dies der genau-einmal Ausführung widerspricht muß sichergestellt werden, daß nur eine dieser Schritt-Transaktionen erfolgreich abgeschlossen werden kann. Dies wird erreicht, indem ein *Votierprotokoll* in das 2-Phasen-Commit-Protokoll (kurz: 2PC-Protokoll) der Schritt-Transaktionen integriert wird. Dieses Protokoll stellt sicher, daß nur bei Zustimmung der Mehrheit der Knoten einer Stufe die Schritt-Transaktion erfolgreich beendet werden kann. Außerdem wird mittels dieses Votierprotokolles auch dafür gesorgt, daß die Beobachter nach Abschluß einer Stufe alle die Stufe betreffenden Informationen – z.B. auch den Agent in ihrer Eingangswarteschlange – löschen können. Das Votierprotokoll wird in Abschnitt 4.4.2 detailliert beschrieben.

Eine wichtige Eigenschaft des Protokolles ist, daß es sich problemlos in gängige Transaktionssysteme mit 2PC-Protokoll integrieren läßt. Die hierbei zugrunde gelegte Architektur lehnt sich stark an die *X/Open Distributed Transaction Processing Architektur* (vgl. XOPEN (1991)) an. Für die Ausführung des 2PC-Protokolles sind in dieser Architektur Transaktionsmanager zuständig. Die durch den Transaktionskontext geschützten Daten (engl.: *recoverable data*) werden durch Ressourcenmanager verwaltet.

Die für den Abschluß einer Schritt-Transaktion (und daher den Abschluß einer Stufe) wichtigen Komponenten und Interaktionen zeigt Abbildung 4-7. Nachdem der Arbeiter der Stufe S_i für die Schritt-Transaktion das *Commit()* aufruft, initiiert der lokale Transaktionsmanager (kurz: TM) das 2PC-Protokoll, welches den Arbeiter und alle Knoten der nachfolgenden Stufe S_{i+1} betrifft. Während des 2PC-Protokolles interagieren die betroffenen TMs mit den Ressourcenmanagern (kurz: RMs), welche bei der Bearbeitung der Stufe beteiligt waren. Neben den Ressourcen, auf die der Agent während der Ausführung des Schrittes zugegriffen hat, sind dies vor allem die Eingangswarteschlangen der Knoten der nachfolgenden Stufe S_{i+1}. Zusätzlich interagiert der TM des Arbeiters mit einem weiteren RM, welcher *Orchestrator* genannt wird. Der Orchestrator ist dafür zuständig, das Votierprotokoll zu orchestrieren. Hierfür existiert auf jedem Knoten der Stufe ein Votierer, welcher vom Orchestrator zur Stimmabgabe aufgefordert wird. Die vorgeschlagene Integration des Votierprotokolles hält die Schnittstelle zwischen Votierprotokoll und 2PC-Protokoll minimal. Der Orchestrator ist aus der Sicht des Transaktionsmanagers nur ein weiterer RM, welcher gegenüber dem TM dieselbe Schnittstelle wie alle anderen RMs (z.B. ein XA-Interface, vgl. BERNSTEIN UND NEWCOMER (1997), XOPEN (1991)) bietet. Dies ermöglicht eine einfache Integration des Votierprotokolles in gängige Middleware-Systeme.

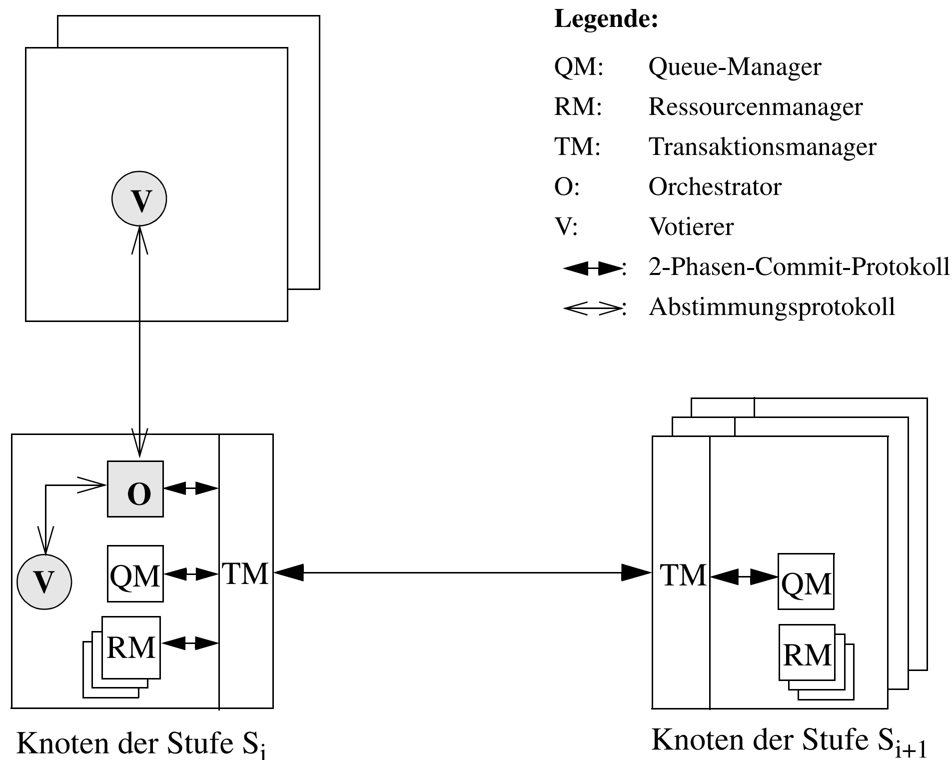


Abbildung 4-7. Für den Abschluß einer Schritt-Transaktion relevante Komponenten und Interaktionen.

Details zu den einzelnen Teilprotokollen werden in den beiden folgenden Abschnitten präsentiert.

4.4.2 Votierprotokoll

Dieser Abschnitt konzentriert sich auf die Beschreibung des Votierprotokolles und dessen Integration in das 2PC-Protokoll. Hierbei wird auf die Funktionalität des 2PC-Protokoll nur soweit eingegangen, wie es zum Verständnis des vorgestellten Votierprotokolles notwendig ist. Eine detaillierte Beschreibung des 2PC-Protokoll findet man beispielsweise in BERNSTEIN UND NEWCOMER (1997). Das vorgestellte Votierprotokoll basiert auf dem fehlertoleranten Votieren mit Mehrheitsentscheidung (engl.: majority consensus). Eine genauere Beschreibung dieses Algorithmus findet man in GIFFORD (1979) und THOMAS (1979). Da das Votierprotokoll prinzipbedingt parallel von mehreren Orchestratoren einer Stufe gestartet werden kann, wird der Algorithmus um einen Mechanismus erweitert, welcher mittels Prioritäten die Bestimmung eines eindeutigen Gewinners des Votierprotokolles erlaubt.

4.4.2.1 Integration in das 2-Phasen-Commit-Protokoll

Wie bereits im letzten Abschnitt beschrieben, ist der Orchestrator aus der Sicht des TM ein normaler RM. Für unser Protokoll-Design nehmen wir an, daß jeder RM eine XA-ähnliche Schnittstelle mit den folgenden Operationen anbietet: *rm_prepare*, *rm_commit* und *rm_rollback*. Abbildung 4-8 zeigt das Zusammenspiel von TM und RM beim 2PC-Protokoll. Wie der Name schon sagt, besteht das 2PC-Protokoll aus zwei Phasen. Die erste Phase beginnt, indem der TM bei den RMs *rm_prepare* aufruft. Jeder RM entscheidet dann für die von ihm verwalteten Daten, ob er die Transaktion erfolgreich abschließen kann. Ist dies der Fall, schreibt er die zum Abschluß der Transaktion notwendigen Daten auf stabilen Speicher und gibt als Antwort ein *rm_yes* zurück. Ist es dem RM nicht möglich, die Transaktion erfolgreich abzuschließen – beispielsweise wegen einer entdeckten Verklemmung (engl.: deadlock) oder wegen anderer Fehler – dann wird *rm_no* als Antwort zurückgegeben. Hat der TM von allen betroffenen RM eine Antwort bekommen, entscheidet er, ob die Transaktion erfolgreich abgeschlossen (nur *rm_yes* als Antworten erhalten) oder abgebrochen wird (mindestens ein *rm_no* erhalten). Diese Entscheidung wird auf stabilen Speicher geschrieben.

In der zweiten Phase werden die RMs vom Ausgang der Transaktion informiert. Mit *rm_commit* wird ein erfolgreicher Ausgang der Transaktion signalisiert, mit *rm_rollback* ein Abbruch der Transaktion. Eine Optimierung überträgt bei Abbruch der Transaktion wie in unserem Beispiel in Abbildung 4-8 b) gezeigt, die Entscheidung nur zu jenen RMs, welche *rm_yes* votiert haben (die anderen RMs wissen bereits, daß die Transaktion abgebrochen wird). Die RMs machen nun entweder die Änderungen an den Daten dauerhaft (bei *rm_commit*) bzw. verwerfen die Ände-

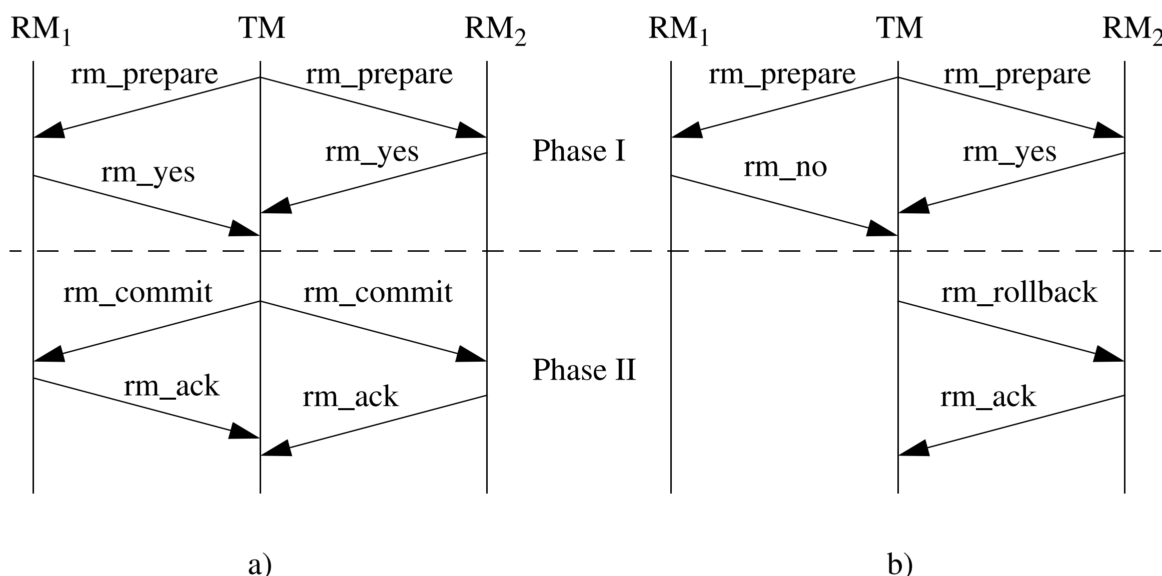


Abbildung 4-8. 2PC-Protokoll
a) Transaktion erfolgreich beendet; b) Transaktion abgebrochen

rungen an den Daten (bei *rm_rollback*). Anschließend wird der TM mittels *rm_ack* informiert, daß die zur Transaktionsbeendigung notwendigen Aktionen vom RM ausgeführt wurden. Sobald der TM von allen RM eine Antwort erhalten hat, kann er die mit der Transaktion assoziierten Daten vergessen.

Die Integration des Votierprotokolles in das 2PC-Protokoll geschieht dadurch, daß der Orchestrator die beiden Phasen des Votierprotokolles durchführt, wenn der TM bei ihm *rm_prepare* (Phase I) bzw. *rm_commit*/*rm_rollback* (Phase II) aufruft. Wenn beim Orchestrator *rm_prepare* aufgerufen wird, sendet jener eine Votieranforderung zu den Votierern der Stufe und sammelt die eingegangenen Voten. Nur wenn er eine Mehrheit von positiven Antworten (*YES-Voten*) bekommt, gibt der Orchestrator *rm_yes* an den TM zurück. Gibt es keine Mehrheit, wird *rm_no* zum TM zurückgeliefert. Hierdurch wird erreicht, daß die Schritt-Transaktion nur dann erfolgreich beendet werden kann, wenn eine Mehrheit von Votierern mit *YES* votiert. Um sicherzustellen, daß dies pro Stufe nur bei genau einer Transaktion geschehen kann, wird von dem hier beschriebenen Protokoll sichergestellt, daß ein Votierer letztendlich jeweils nur für einen Orchestrator einer Stufe ein *YES-Votum* abgibt. Eine detaillierte Beschreibung der beiden Phasen erfolgt in Abschnitt 4.4.2.3.

4.4.2.2 Stabile Zustände des Protokolles

Für die Realisierung des Protokolles werden zwei verschiedene Typen stabiler Zustände benötigt, welche auf stabilem Speicher gespeichert werden und daher Knotenfehler überleben.

Der *Transaktionsstatus* beschreibt den Status der Schritt-Transaktion und wird durch den Orchestrator verwaltet. Eine Schritt-Transaktion kann sich in den Zuständen “Unknown” (unbekannt), “Ready” (bereit zum Beenden der Transaktion) und “Committed” (Transaktion erfolgreich abgeschlossen) befinden. Im “Unknown”-Zustand liegen analog zur “Presumed Abort”-Optimierung des 2PC-Protokoll (vgl. beispielsweise BERNSTEIN UND NEWCOMER (1997)) keine Informationen auf stabilem Speicher vor.

Der *Stufenstatus* beschreibt den Status einer Stufe. Eine Stufe kann sich in den Zuständen “Unknown” (unbekannt) und “Active” (aktiv) befinden. Im Gegensatz zum Transaktionsstatus wird der Stufenstatus auf jedem Knoten der Stufe gehalten. Die Verwaltung dieses Status geschieht auf dem Arbeiter-Knoten der Stufe im Kontext der Ausführung des Agenten, und auf den Beobachterknoten durch die Votierer der Stufe. Die Zustandsinformation einer aktiven Stufe wird in einem *Stufen-Record* (engl.: *stage record*) auf stabilem Speicher abgelegt.

Der Stufen-Record setzt sich aus den folgenden Informationen zusammen:

- Ein *Stufenbezeichner* (engl.: *stage identifier*, kurz StufenId), welcher sich aus dem Agentennamen und einem Schrittzähler zusammensetzt. Der Agentenname identifiziert den Agent global eindeutig. Der Schrittzähler zählt die Anzahl der durch den Agenten bereits ausgeführten Schritte.

- Die Liste der Knoten der Stufe. Hierbei ist für jeden Knoten neben dessen (eindeutigem) Knotennamen auch die Priorität enthalten.

Wenn ein Agent zur nächsten Stufe “migriert”, wird nicht nur der Agent sondern auch der Stufen-Record in die Eingangswarteschlangen der Knoten der nächsten Stufe geschrieben. Sobald der Stufen-Record in der Eingangswarteschlange eines Knoten steht, ist die Stufe auf diesem Knoten “Active”. Da das Schreiben aller Stufen-Records einer Stufe innerhalb der Schritt-Transaktion der vorherigen Stufe geschieht, sind entweder alle Knoten einer Stufe “Active” oder keiner. Die Knoten der Stufe lesen diesen Stufen-Record dann, ohne ihn aus der Eingangswarteschlange zu entfernen, und entscheiden anhand der Prioritäten, ob sie initial Arbeiter oder Beobachter sind.

4.4.2.3 Phasen des Votierprotokolles

Das Protokoll besteht analog zum 2PC-Protokoll aus zwei Phasen. In der ersten Phase werden die Voten von den Votierern der Stufe gesammelt. In der zweiten Phase wird das Ergebnis der Schritt-Transaktion den Votierern der Stufe bekannt gemacht. Ist die Stufe abgeschlossen, entfernen die Votierer der Stufe die mit der Stufe assoziierten, auf stabilem Speicher liegenden Informationen (Stufen-Record und Agent).

Pro Knoten gibt es jeweils nur einen Orchestrator und einen Votierer. Diese sind für die Bearbeitung sämtlicher auf dem Knoten aktiven Stufen zuständig. Zur Identifikation des Orchestrators und des Votierers eines Knotens wird der (eindeutige) Knoten-Bezeichner des Knotens verwendet. Der Empfänger einer Nachricht (Orchestrator, Votierer) ergibt sich jeweils aus dem Typ der Nachricht, die zugehörige Stufe wird jeweils mittels dem in der Nachricht mitübertragenen Stufenbezeichner identifiziert. Um die Lesbarkeit zu erleichtern, werden in den folgenden Abschnitten, abhängig vom Kontext, die Begriffe *Votierer-Bezeichner* (kurz: *VotiererId*) und *Orchestrator-Bezeichner* (kurz: *OrchId*) anstatt *Knoten-Bezeichner* (kurz: *KnotenId*) verwendet.

Phase I. Algorithmus 4-4 und Algorithmus 4-5 zeigen Phase I des Votierprotokolles von Orchestrator und Votierer in einer Pseudo-Code-Notation. Die meisten der im Code verwendeten Variablen hängen von der StufenId ab. Um den Code möglichst kompakt zu halten, werden diese Variablen jedoch nur als x anstatt $x[StufenId]$ geschrieben.

Phase I des Votierprotokolles wird gestartet, wenn ein Orchestrator *rm_prepare* vom TM empfängt. Der Orchestrator versendet dann zuerst eine *VOTE*-Aufforderung an jeden Votierer der zur Stufe gehörenden Knoten (einschließlich dem sich auf dem Knoten des Orchestrators befindlichen Votierers). Die Aufforderung enthält den Stufenbezeichner, den Bezeichner des Orchestrators und den (lokalen) Zeitstempel des Beginns der Phase I. Danach wartet der Orchestrator auf die Voten. Dabei versendet er die *VOTE*-Aufforderungen periodisch an jene Votierer, die noch nicht geantwortet haben. Der Zeitstempel wird benötigt, um die Voten den *VOTE*-Aufforderungen zuzuordnen, da es beispielsweise durch Knotenfehler vorkommen kann, daß ein

```
Receive rm_prepare(transactionId){ // from TM
    stageVoters = set of voterIds in stage
    YV=NV=CYV={}// set of received votes
    voteStart = timestamp() // unique timestamp
    Repeat{ //periodic sending of vote request
        Send VOTE (stageId, orchId, voteStart)
          To stageVoters\(YV+NV+CYV)
        Wait(timeout)
    } Until (voteResult is set)
    Send voteResult(transactionId) To TM
}

Receive YES(stId, voterId, time){
    If (stId==stageId and timeStamp==voteStart){
        YV = YV + voterId
    }
    If (YV contains majority){
        transactionState = Ready
        set voteResult = rm_yes
}   }

Receive COND_YES(stId, voterId, orchSet, time){
    If (stId==stageId and timeStamp==voteStart){
        CYV = CYV + (voterId, orchSet)
        CheckCondYes(CYV, YV)
    }
    If (YV contains majority){
        transactionState = Ready
        set voteResult = rm_yes
}   }

Receive GIVE_UP(StageId){
    abort local transaction
}

Receive NO(stId, voterId, time){
    If (stId==stageId and time==voteStart){
        NV = NV + voterId
    }
    If (NV contains majority){
        set voteResult = rm_no
        Send UN_VOTE(stageId, orchId, voteStart)
          To voters in YV+CYV
}   }

Receive HIGHER_PRIO(StageId){
    If (transactionState==Unknown)
        set voteResult = rm_no
        Send UN_VOTE(stageId, orchId, voteStart)
          To voters in YV+CYV
        Reply GAVE_UP
    }Else{
        Reply ALREADY_DONE
}   }

Procedure CheckCondYes(CYV, YV){
    Do{
        changes=false
        ForEach (vId, orchSet) in CYV Do{
            If (orchSet\YV == {}){
                CYV = CYV\(vId, orchSet)
                YV = YV + vId
                changes=true
        }   }
    } While (changes == true)
}
```

Algorithmus 4-4. Votierprotokoll Phase I des Orchestrators

Votierer mehrere Votierrunden startet und in diesem Fall das Votum einer “alten” Votierrunde ignoriert werden muß. Anstatt des Zeitstempels kann auch – falls verfügbar – der Transaktionsbezeichner der aktuellen Schritt-Transaktion verwendet werden. Der Zeitstempel ermöglicht es allerdings, auch Reihenfolgefehler bei Nachrichten zu tolerieren.

Die Votierer merken sich die von ihnen für eine Stufe abgegebenen Voten in einer Menge namens *OrchSet*. Für jede momentan auf seinem Knoten aktive Stufe verwaltet der Votierer ein *OrchSet*. Die *OrchSets* werden auf stabilem Speicher abgelegt. Immer wenn der Votierer eine *YES*-Votum oder *COND_YES*-Votum für einen Orchestrator einer Stufe abgibt, fügt er dessen Bezeichner in das entsprechende *OrchSet* ein. Solange keine Fehler auftreten und deshalb der Agent auch nur auf einem Knoten ausgeführt wird, enthält ein *OrchSet* beim Beenden der Stufe nur einen Orchestrator-Bezeichner. Treten jedoch Fehler auf, kann das *OrchSet* mehrere Bezeichner enthalten.

```
Receive VOTE(stageId, orchId, time){
    If (orchSet=={}){ (1)
        orchSet = orchSet + orchId
        Send YES(stageId, voterId, time) To orchId
    }Else (2) If(orchId ∉ orchSet){
        N = node with highest priority in orchSet
        If (prio(orchId) < prio(N)){ (3)
            Send NO(stageId, voterId, time) To orchId
        }Else{(4) // orchId has highest priority in set
            If (N ≠ voterId){(5)
                Send COND_YES(stageId, voterId,
                  orchSet, time) To orchId
                orchSet = orchSet + voterId
            } Else{(6)
                Send HIGHER_PRIO(stageId) To
                  local orchestrator
                Receive ANSWER from local
                  orchestrator
                If (ANSWER == GAVE_UP){ (7)
                    orchSet = orchSet\{N}
                    If (orchSet == {}){ (8)
                        Send YES(stageId, voterId,
                          time) To orchId
                    }Else{(9)
                        Send COND_YES(stageId,
                          voterId, orchSet, time)
                          To orchId
                    }
                        orchSet = orchSet + orchId
                }Else{ (10)// ALREADY_DONE
                    Send NO(stageId, voterId, time) To
                      orchId
            }   }   }
    } Else (11) // orchId already in orchSet
        N = node with highest priority in orchSet
        If (orchId==N){(12)
            Send COND_YES(stageId, voterId,
              orchSet, time) To orchId
        }Else{(13)
            Send NO(stageId, voterId, time) To orchId
            orchSet = orchSet\{orchId}
        }
    }
    If (voted YES or COND_YES){
        DoInquiry( stageId, orchId)
        // see recovery section
    }
}

Receive UN_VOTE(stId, orchId, time){
    orchSet = orchSet \  orchId
}
```

Algorithmus 4-5. Votierprotokoll Phase I des Votierers

Sobald ein Votierer eine *VOTE*(*StufenId*, *OrchId*, *Zeitstempel*)-Aufforderung erhält, bestimmt er seine Antwort anhand des zur Stufe gehörigen *OrchSets*. Ist das *OrchSet* leer **(1)**[1], hat der Votierer für diese Stufe entweder noch keine Stimme abgegeben oder die von ihm abgegebenen Stimmen wurden von den Orchestratoren zurückgegeben (wird weiter unten erläutert). In diesem Fall wird *OrchId* zum *OrchSet* der Stufe *StufenId* hinzugefügt und ein *YES*(*StufenId*, *VotiererId*, *Zeitstempel*)-Votum an den Orchestrator *OrchId* verschickt. Der Zeitstempel entspricht dem in der *VOTE*(...)-Aufforderung empfangenen. Abbildung 4-9 zeigt ein Szenario, in dem nur ein Orchestrator das Votierprotokoll initiiert. Weder Votierer V_1 noch Votierer V_2 haben schon ein Votum abgegeben und können daher Orchestrator O_1 jeweils ein *YES*(...)-Votum schicken.

Ist das *OrchSet* nicht leer **(2)**, konkurrieren offensichtlich mehrere Orchestratoren um die Stimme. Um sicherzustellen, daß letztendlich nur einer der Orchestratoren die notwendige Mehrheit erhält, bevorzugt das Votierprotokoll den Orchestrator des Knotens mit der höchsten Priorität. Sei *N* der Orchestrator im *OrchSet*, dessen Knoten im Vergleich mit den Knoten der anderen Orchestratoren im *OrchSet* die höchste Priorität hat. Wenn *OrchSet* nicht leer ist und der Knoten von *N* eine höhere Priorität hat als der Knoten von *OrchId* **(3)**, dann hat der Votierer schon für

1. Die Zahlen in Klammern beziehen sich auf die Fall-Numerierungen in Algorithmus 4-5

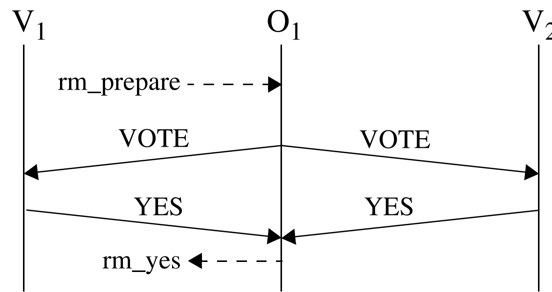


Abbildung 4-9. Phase I: Ein Orchestrator initiiert das Votieren

einen Knoten mit höherer Priorität mit *YES*(...) votiert. In diesem Fall sendet der Votierer ein *NO*(*StufenId*, *VotiererId*, *Zeitstempel*)-Votum an den Orchestrator *OrchId* zurück – *OrchId* bekommt von diesem Votierer keine Stimme. Abbildung 4-10 zeigt ein Szenario, in dem zwei Orchestratoren zur beinahe selben Zeit das Votierprotokoll initiieren. Votierer V_1 hat seine Stimme bereits an Orchestrator O_1, dessen Knoten eine höhere Priorität besitzt als der Knoten des Orchestrators O_2, vergeben. Aus diesem Grunde sendet V_1 ein *NO*-Votum an O_2.

Wenn *OrchSet* nicht leer ist und der Knoten von *N* eine niedrigere Priorität hat als der Knoten des Orchestrators *OrchId* **(4)**, dann hat der Votierer zwar schon für die Stufe votiert – jedoch nur für einen (oder mehrere) Knoten mit niedrigerer Priorität. Ist *N* nicht der Orchestrator des eigenen Knotens **(5)**, sendet der Votierer ein *COND_YES*(*StufenId*, *VotiererId*, *OrchSet, Zeitstempel*)-Votum an den Orchestrator *OrchId* zurück und fügt *OrchId* zum *OrchSet* hinzu. Die Semantik dieses Votums ist, daß der Votierer mit *YES* votiert – vorausgesetzt, daß die Votierer der Knoten, deren Orchestratoren in *OrchSet* enthalten sind, zustimmen. Im Szenario in Abbildung 4-10 hat der Votierer V_3 bereits dem Orchestrator O_2 ein *YES*-Votum geschickt als vom Orchestrator O_1 eine Votieraufforderung ankommt. Da der Knoten von O_1 eine höhere Priorität hat als

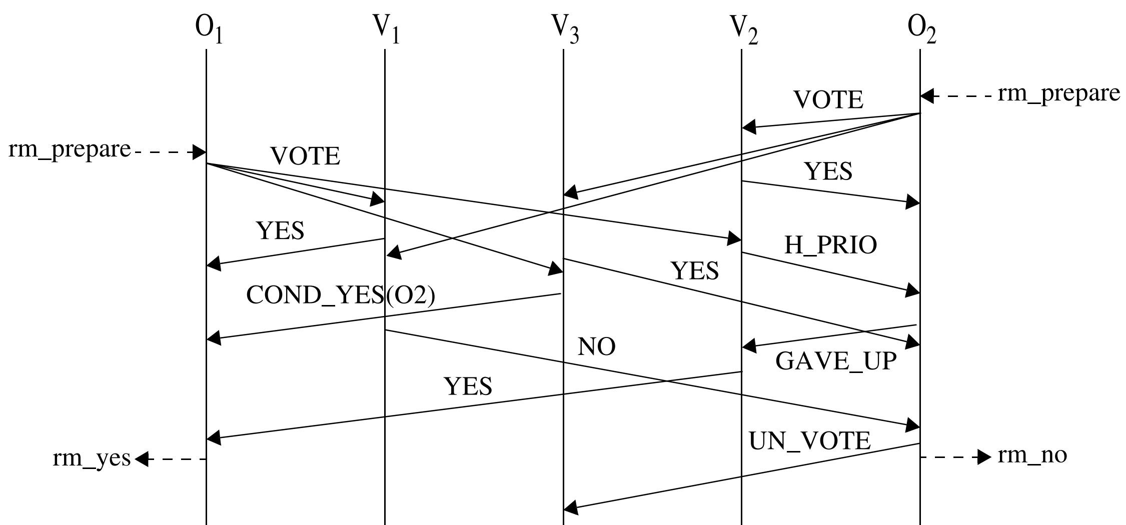


Abbildung 4-10. Phase I: Zwei Orchestratoren initiieren das Votieren parallel

der Knoten von O_2, informiert V_3 den Orchestrator O_1 mittels des *COND_YES*-Votums, daß er zustimmt, falls V_2 (der Votierer auf Knoten von O_2) ebenfalls zustimmt. Da V_2 letztendlich (vgl. nächster Absatz) auch ein *YES*-Votum zu O_1 schickt, verwandelt sich damit das konditionelle *COND_YES*-Votum von V_3 in ein *YES*-Votum.

Ist allerdings im letzten Falle *N* der Orchestrator auf dem Knoten des Votierers **(6)**, dann hat der Orchestrator auf dem Knoten des Votierers ebenfalls schon mit dem Votierprotokoll begonnen. Da der Knoten des Orchestrators *OrchId* eine höhere Priorität hat, sollte sich der lokale Orchestrator möglichst geschlagen geben und dem Orchestrator *OrchId* den Vorrang lassen. Hierzu sendet der Votierer *VoterId* dem lokalen Orchestrator eine *HIGHER_PRIO(StufenId)*-Anforderung und gibt dem Orchestrator damit bekannt, daß ein Knoten mit höherer Priorität ebenfalls die Stufe abschließen möchte. Ist der Transaktionsstatus der Stufentransaktion noch "Unknown", antwortet der Orchestrator mit *GAVE_UP* um anzuzeigen, daß er die Stufentransaktion abbricht. Ist die Transaktion schon in einem der Zustände "Ready" oder "Committed", dann antwortet der Orchestrator mit *ALREADY_DONE*. Antwortet der Orchestrator mit *ALREADY_DONE* **(10)**, sendet der Votierer *VoterId* ein *NO*(*StufenId*, *VotiererId*, *Zeitstempel*)-Votum an den Orchestrator *OrchId*. Ansonsten **(7)** sendet er ein *COND_YES*(*StufenId*, *VotiererId*, *OrchSet*-{*N*}, *Zeitstempel*)-Votum und entfernt *N* aus dem *OrchSet* **(9)**. Ist *OrchSet*-{*N*} die leere Menge, kann hierbei aufgrund der Semantik von *COND_YES* ein *YES*-Votum anstatt des *COND_YES*-Votums geschickt werden **(8)**.

Im Szenario in Abbildung 4-10 erhält der Votierer V_2 eine *VOTE*-Aufforderung von O_1 nachdem er schon ein *YES*-Votum an den lokalen Orchestrator O_2 vergeben hat. Da der Knoten von O_1 die höhere Priorität hat, sendet V_2 eine *HIGHER_PRIO(StufenId)*-Anforderung an O_2, welcher, da das Votierprotokoll (und daher die Schritt-Transaktion) bei ihm noch nicht abgeschlossen ist, mit *GAVE_UP* antwortet. Das ermöglicht V_2, ein *YES*-Votum an O_1 zu schicken.

Bisher nicht betrachtet wurde der Fall, daß der Orchestrator *OrchId* schon im *OrchSet* enthalten ist **(11)**. Dies kann beispielsweise dann passieren, wenn die Antwort des Votierers an den Orchestrator wegen einer kurzfristigen Netzwerkpartitionierung verloren ging (und der Orchestrator die *VOTE*-Aufforderung erneut verschickt) oder wenn der Knoten des Orchestrators kurzzeitig ausgefallen war (Zusammenbruch während der ersten Phase des Votierprotokolles) und nach dem Neustart den Agent erneut ausgeführt hat. Hier muß man nur zwei Fälle beachten. Ist *OrchId* der Orchestrator im *OrchSet*, dessen Knoten die höchste Priorität in der Stufe hat **(12)**, dann schickt der Votierer ein *COND_YES*(*StufenId*, *VotiererId*, *OrchSet*-{*OrchId*}, *Zeitstempel*)-Votum (bzw. *YES*-Votum, wenn *OrchSet*-{*OrchId*} die leere Menge ist). Ansonsten **(13)** wird ein *NO*-Votum verschickt und *OrchId* aus *OrchSet* entfernt.

Zur Verwaltung der gerade im Orchestrator aktuell ablaufenden Votierrunden unterhält der Orchestrator für jede Votierrunde, welche durch die *StufenId* des Agenten und den *Zeitstempel* des Votierbeginns (bzw. alternativ den Transaktionsbezeichner der Schritt-Transaktion) gekennzeichnet ist, drei Mengen im flüchtigen Speicher: *YesVoten*, *NoVoten* und *CondYesVoten* (in Al-

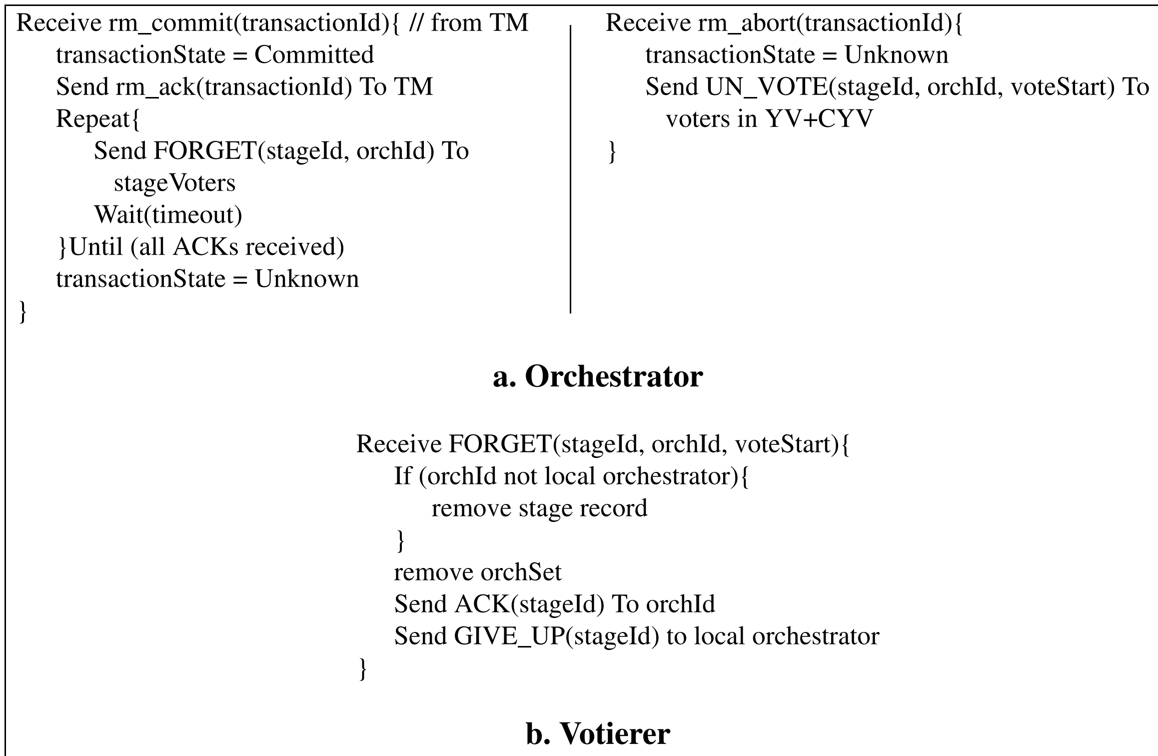
gorithmus 4-4 abgekürzt mit *YV*, *NV* und *CYV*). Erhält der Orchestrator nun ein Votum von einem Votierer, kann er anhand der *StufenId* und des *Zeitstempels* feststellen, ob es sich hierbei um eine Antwort auf eine gerade laufende Votierrunde handelt. Ist das Votum ein aktuelles *YES*- bzw. *NO*-Votum, so wird die in dem Votum enthaltene *VotiererId* in die *YesVotes*- bzw. *NoVotes*-Menge eingefügt. Bei einem *COND_YES*-Votum wird das in dem Votum enthaltene (*VotiererId*, *OrchSet*)-Paar in die *CondYesVoten*-Menge eingefügt. Sobald alle Votierer der Knoten, deren Orchestratoren in *OrchSet* enthalten sind, in der *YesVoten*-Menge enthalten sind, wird das (*VotiererId*, *OrchSet*)-Paar aus der *CondYesVoten*-Menge entfernt und *VotiererId* in die *YesVoten*-Menge eingefügt (d.h. aus dem *COND_YES*-Votum wird ein *YES*-Votum). Anders ausgedrückt: gilt für ein (*VotiererId*, *OrchSet*)-Paar aus der *CondYesVoten*-Menge, daß *OrchSet*-*YesVoten* die leere Menge ist, kann das (*VotiererId*, *OrchSet*)-Paar aus der *CondYesVoten*-Menge entfernt und *VotiererId* in die *YesVoten*-Menge eingefügt werden.

Sobald die *YesVoten*-Menge eine Mehrheit der Voten (d.h. mehr als die Hälfte der Votierer der Stufe müssen letztendlich mit *YES* gestimmt haben) enthält, geht der Orchestrator in den "Ready"-Zustand über (auf stabilem Speicher!) und antwortet dem lokalen TM mit *rm_yes* (vgl. Orchestrator O_1 in den obigen Szenarien). Dann wartet der Orchestrator auf die Commit- bzw. Abbruch-Entscheidung des TM. Wird jedoch eine Mehrheit an Voten unmöglich (d.h. *NoVoten* enthält mindestens die Hälfte der Votierer der Stufe), antwortet der Orchestrator dem lokalen TM mit *rm_no*, sendet eine *UN_VOTE*(*StufenId*, *OrchId*)-Aufforderung an alle Votierer, welche in *YesVoten* und *CondYesVoten* enthalten sind und vergißt dann die Transaktion (vgl. beispielsweise Orchestrator O_2 in Abbildung 4-10). Zu beachten ist, daß durch ein *rm_no* der TM die Schritt-Transaktion abbricht. Hierdurch wechselt die Rolle des Knotens vom Arbeiter zum Beobachter (vgl. auch Abschnitt 4.4.3.1).

Wenn der Orchestrator eine *HIGHER_PRIO(StufenId)*-Anforderung bekommt, antwortet er mit *ALREADY_DONE* wenn sich die Schritt-Transaktion schon im Zustand "Ready" oder "Committed" befindet. Befindet sich die Transaktion jedoch noch im "Unknown"-Zustand, antwortet er mit *GAVE_UP*, gibt *rm_no* an den lokalen TM zurück, sendet *UN_VOTE*-Aufforderungen an alle Votierer in *YesVoten* und *CondYesVoten* und vergißt die Transaktion. Analog zu oben wechselt dann die Rolle des Knotens vom Arbeiter zum Beobachter.

Ein Orchestrator bekommt von seinem lokalen Votierer eine *GIVE_UP*-Aufforderung, wenn ein anderer Knoten die Schritt-Transaktion bereits erfolgreich abgeschlossen hat, auf dem lokalen Knoten eine Schritt-Transaktion läuft und der Votierer vom Orchestrator noch keine *VOTE*-Aufforderung bekommen hat. In diesem Falle kann die lokale Schritt-Transaktion abgebrochen und vergessen werden.

Phase II. Algorithmus 4-6 zeigt Phase II des Votierprotokolles von Orchestrator und Votierer. Wenn der TM die Transaktion erfolgreich abschließt (d.h. Commitment durchführt), verschickt er an alle teilnehmenden Ressourcenmanager *rm_commit*. Wenn den Orchestrator ein solches *rm_commit* erreicht, geht der Orchestrator in den "Committed"-Zustand über (auf stabilem



```
Receive rm_commit(transactionId){ // from TM
    transactionState = Committed
    Send rm_ack(transactionId) To TM
    Repeat{
        Send FORGET(stageId, orchId) To
          stageVoters
        Wait(timeout)
    }Until (all ACKs received)
    transactionState = Unknown
}

Receive rm_abort(transactionId){
    transactionState = Unknown
    Send UN_VOTE(stageId, orchId, voteStart) To
      voters in YV+CYV
}
```

a. Orchestrator

```
Receive FORGET(stageId, orchId, voteStart){
    If (orchId not local orchestrator){
        remove stage record
    }
    remove orchSet
    Send ACK(stageId) To orchId
    Send GIVE_UP(stageId) to local orchestrator
}
```

b. Votierer

Algorithmus 4-6. Votierprotokoll Phase II

Speicher!) und antwortet dem lokalen TM mit *rm_ack*. Anschließend sendet er allen Votierern der Stufe eine *FORGET*(*StufenId*, *OrchId*)-Nachricht und wartet auf deren Empfangsbestätigungen. Hierbei versendet der Orchestrator die *FORGET*-Nachrichten periodisch, bis er ein *ACK*(*StufenId*) (von engl.: acknowledgement, (Empfangs-)Bestätigung) von allen Votierern der Stufe erhalten hat. Sobald alle *ACK*s angekommen sind, geht der Orchestrator in den “Unknown”-Zustand über, d.h. er löscht die Zustandsinformation vom stabilen Speicher, und vergißt die Transaktion.

Sobald ein Votierer eine *FORGET*(*StufenId*, *OrchId*)-Nachricht erhält, entfernt er alle vorhandenen Informationen über die Stufe. Trifft die Nachricht von einem anderen Knoten ein, so wird dazu der Stufenstatus des Knotens atomar in den “Unknown”-Zustand überführt, d.h. der Stufen-Record inklusive des Agenten wird atomar aus der Knoteneingangswarteschlange entfernt. Dies ist auf dem Knoten, auf dem die Schritt-Transaktion beendet wurde, nicht notwendig, da diese Aktion schon Teil der Schritt-Transaktion ist. Auf jeden Fall muß dann der Votierer aber noch das *OrchSet* vom stabilen Speicher löschen und an *OrchId* eine *ACK*(*StufenId*)-Nachricht senden. Sollte der lokale Orchestrator zu diesem Zeitpunkt ebenfalls an einer Schritt-Transaktion für die gerade beendete Stufe *StufenId* beteiligt sein, wird diesem eine *GIVE_UP*(*StufenId*)-Nachricht geschickt, worauf die lokale Schritt-Transaktion abgebrochen wird.

Abbildung 4-11 zeigt die Fortführung des Beispieles aus Abbildung 4-10. Nachdem sich Orchestrator O_1 in der Phase I gegen Orchestrator O_2 durchgesetzt hat, tritt er in Phase II ein nachdem

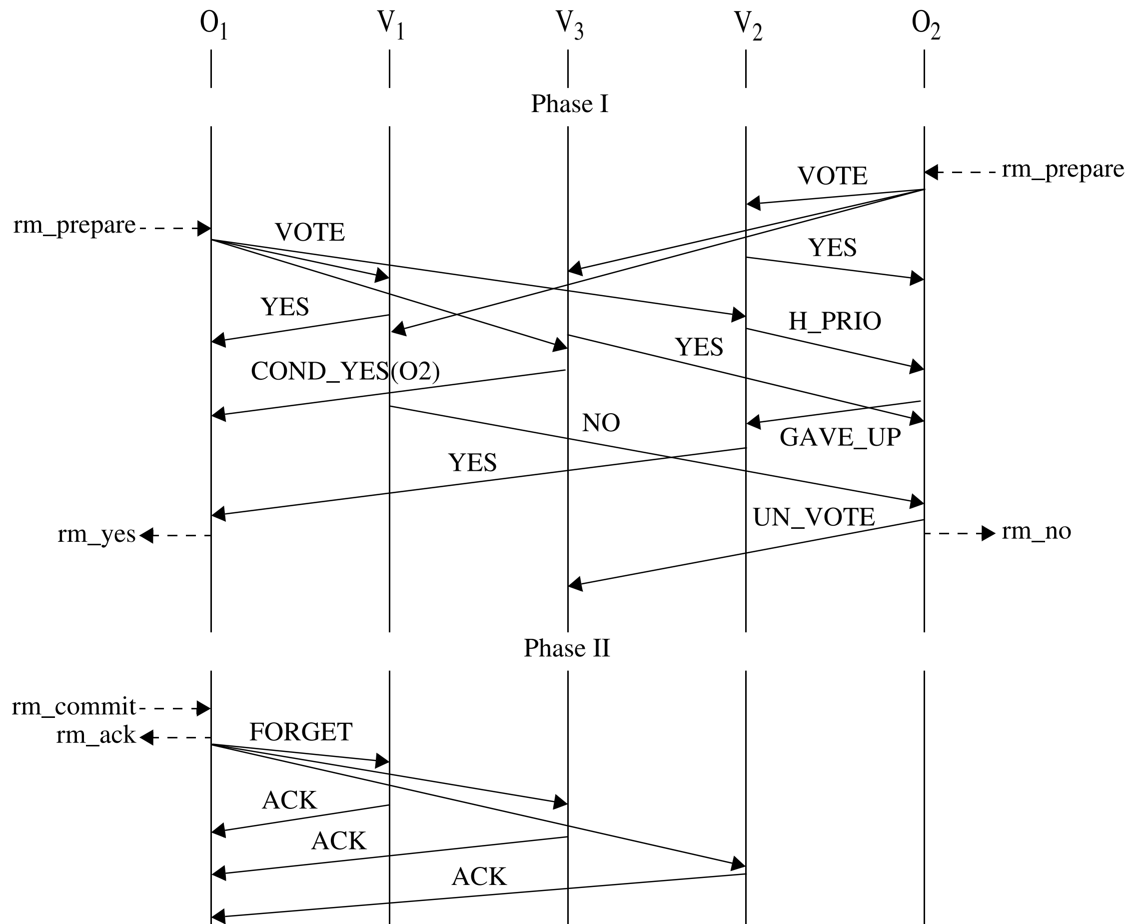


Abbildung 4-11. Phasen I + II des Votierprotokolles

er vom Transaktionsmanager das *rm_commit* erhält. Er schickt eine *FORGET*-Nachricht an die beteiligten Votierer und erwartet deren *ACK*-Nachricht.

Wenn jedoch der Orchestrator vom TM anstatt des *rm_commit* ein *rm_abort* empfängt, geht er in den Transaktionsstatus “Unknown” über und sendet dann *UN_VOTE*(*StufenId*, *OrchId*)-Aufforderungen an alle Votierer in *YesVoten* und *CondYesVoten*. Danach wird der Agent erneut (in einer Schritt-Transaktion) ausgeführt.

Votierer, welche eine *UN_VOTE*(*StufenId*, *OrchId*)-Aufforderung erhalten, entfernen *OrchId* aus dem zur Stufe *StufenId* gehörigen *OrchSet*. Dieser UN_VOTE-Mechanismus, welcher es einem Orchestrator erlaubt, ein von einem Votierer erhaltenes Votum “zurückzugeben” ist notwendig, damit Knoten mit einer niedrigeren Priorität eine Voten-Mehrheit erhalten können, nachdem ein Orchestrator auf einem Knoten mit höherer Priorität aufgegeben hat.

4.4.2.4 Fehlerbehandlung

Gemäß dem in Abschnitt 4.1.3 vorgestellten Fehlermodell muß das Protokoll mit Knotenausfällen und mit Nachrichtenverlusten im Falle von Netzwerkpartitionierungen zurechtkommen.

Drei Mechanismen zur Behandlung solcher Fehler sind in das in den letzten Abschnitten beschriebene Votierprotokoll eingebaut. Der Orchestrator versendet seine *VOTE*-Aufforderungen periodisch bis er eine Stimmenmehrheit erhalten hat (vgl. Algorithmus 4-4). Hierdurch wird das Protokoll sowohl gegen den Verlust des *VOTE*-Aufrufes als auch gegen kurzfristigen Ausfall eines Votierers (durch Knotenausfall) immun. Der Verlust des Votums eines Votierers wird durch das erneute Versenden des *VOTE*-Aufrufes ebenfalls kompensiert. Dabei spielt dann auch der zweite eingebaute Mechanismus eine Rolle. Trifft beim Votierer ein *VOTE*-Aufruf von einem Orchestrator ein, der sich schon im entsprechenden *OrchSet* befindet, erfährt dieser *VOTE*-Aufruf die weiter oben vorgestellte Sonderbehandlung (vgl. Algorithmus 4-5 Fall **(11)**). Hierdurch werden noch zwei weitere Fehlerfälle behandelt: Ausfall des Knotens des Orchestrators während der ersten Phase des Votierprotokolles mit anschließendem Neustart der Schritt-Transaktion nach Wiederanlauf des Knotens und Verlust einer *UN_VOTE*-Nachricht durch Netzwerkpartitionierung. Der dritte eingebaute Mechanismus ist das periodische Versenden der *FORGET*-Nachrichten (vgl. Algorithmus 4-6 a.). Dieser dient ebenso dazu, sowohl den Verlust der *FORGET*- und *ACK*-Nachrichten als auch den (temporären) Ausfall anderer Knoten der Stufe zu kompensieren.

Wie Abbildung 4-12 zeigt, ist es jedoch trotz dieser Mechanismen nach wie vor möglich, daß in einem *OrchSet* Einträge verbleiben, welche nicht zu aktuell stattfindenden Abstimmungen gehören. In Abbildung 4-12 empfängt der Orchestrator in der zweiten Phase des Votierprotokolles ein *rm_abort* vom Transaktionsmanager und verschickt daraufhin *UN_VOTE*-Nachrichten an die Votierer. Die für Votierer V_2 bestimmte Nachricht geht jedoch wegen einer Netzwerkpartitionierung verloren und V_2 entfernt deshalb Orchestrator O_1 nicht aus seinem *OrchSet*. Um auch solche Einträge entfernen zu können, wird jeder *OrchSet*-Eintrag mit einem Timeout versehen. Läuft der Timeout ab, bevor vom zugehörigen Orchestrator eine *FORGET*-, eine

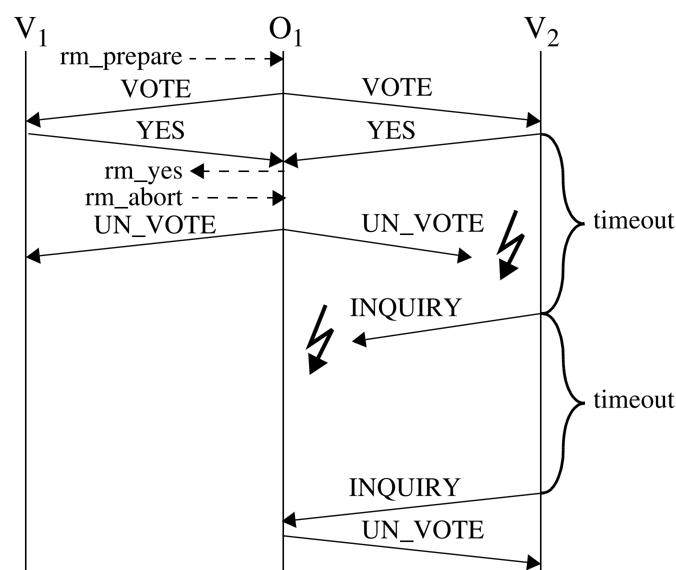
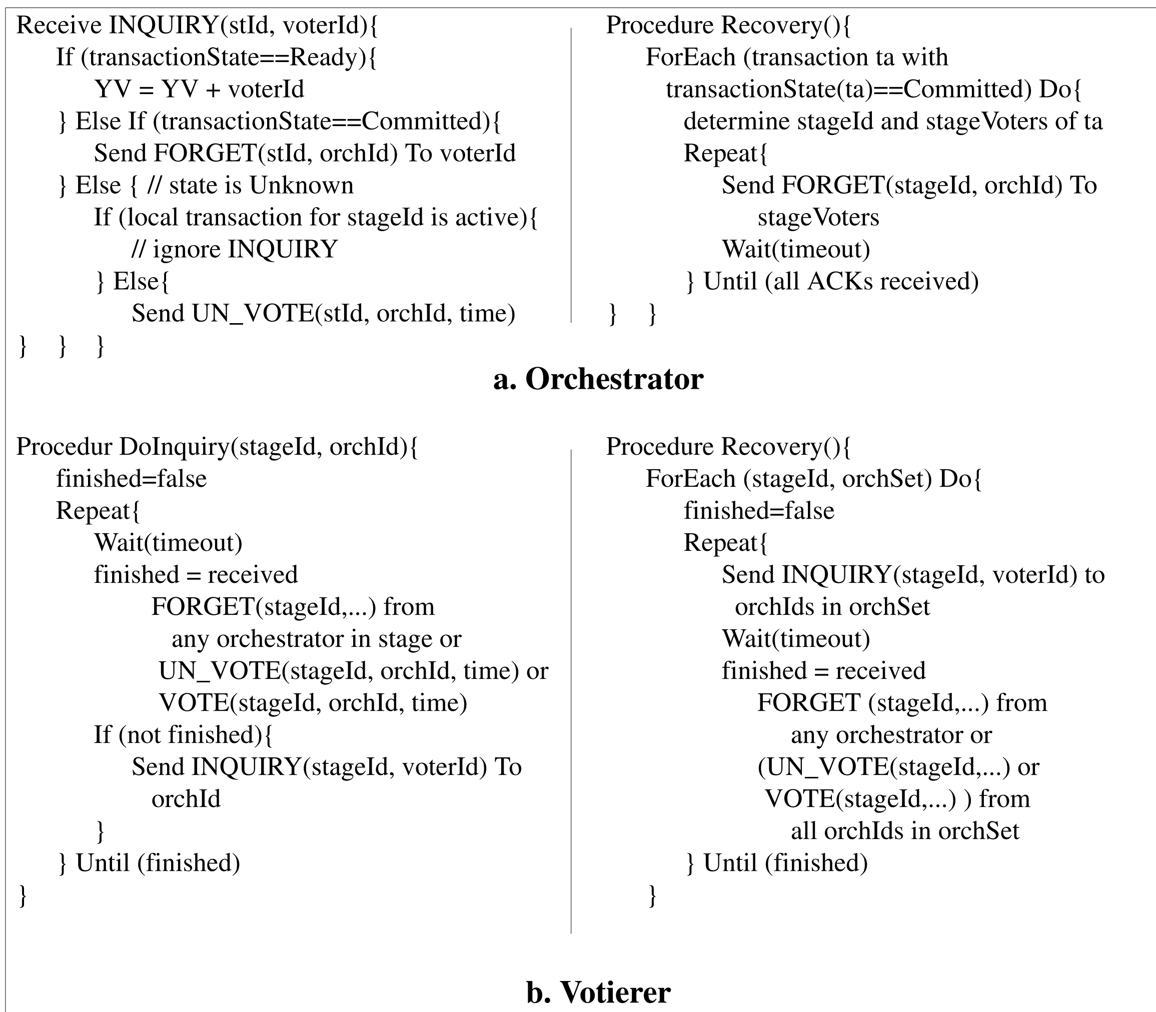


Abbildung 4-12. Verlust von UN_VOTE- und INQUIRY-Nachrichten



```
Receive INQUIRY(stId, voterId){
    If (transactionState==Ready){
        YV = YV + voterId
    } Else If (transactionState==Committed){
        Send FORGET(stId, orchId) To voterId
    } Else { // state is Unknown
        If (local transaction for stageId is active){
            // ignore INQUIRY
        } Else{
            Send UN_VOTE(stId, orchId, time)
}   }   }

Procedure Recovery(){
    ForEach (transaction ta with
      transactionState(ta)==Committed) Do{
        determine stageId and stageVoters of ta
        Repeat{
            Send FORGET(stageId, orchId) To
                stageVoters
            Wait(timeout)
        } Until (all ACKs received)
}   }
```

a. Orchestrator

```
Procedur DoInquiry(stageId, orchId){
    finished=false
    Repeat{
        Wait(timeout)
        finished = received
                FORGET(stageId,...) from
                  any orchestrator in stage or
                 UN_VOTE(stageId, orchId, time) or
                 VOTE(stageId, orchId, time)
        If (not finished){
            Send INQUIRY(stageId, voterId) To
              orchId
        }
    } Until (finished)
}

Procedure Recovery(){
    ForEach (stageId, orchSet) Do{
        finished=false
        Repeat{
            Send INQUIRY(stageId, voterId) to
              orchIds in orchSet
            Wait(timeout)
            finished = received
                FORGET (stageId,...) from
                    any orchestrator or
                (UN_VOTE(stageId,...) or
                 VOTE(stageId,...) ) from
                    all orchIds in orchSet
        } Until (finished)
    }
```

b. Votierer

Algorithmus 4-7. Fehlerbehandlung

UN_VOTE- oder eine *VOTE*-Nachricht empfangen wird, schickt der Votierer periodisch eine *INQUIRY*(*StageId*, *VotiererId*)-Nachricht an diesen Orchestrator, bis dieser entweder mit einer *UN_VOTE*-Nachricht antwortet oder eine weitere *VOTE*-Aufforderung schickt oder bis einer der Orchestratoren der Stufe eine *FORGET*-Nachricht schickt (vgl. Algorithmus 4-7b, *DoInquiry(..)*). Im Beispiel in Abbildung 4-12 muß der Votierer zwei *INQUIRY*-Nachrichten an den Orchestrator schicken, da die erste *INQUIRY*-Nachricht durch die noch anhaltende Netzwerkpartitionierung verloren geht. Ist das Ziel der *INQUIRY*-Nachrichten der lokale Orchestrator, wird das Senden dieser Nachrichten auch beendet, sobald der Orchestrator auf eine *HIGHER-PRIO*-Anforderung mit einem *GAVE_UP* antwortet (in Algorithmus 4-7 nicht berücksichtigt).

Die Antwort eines Orchestrators auf eine eintreffende *INQUIRY*(*StufenId*, *VotiererId*)-Nachricht hängt vom aktuellen Status der Schritt-Transaktion ab (vgl. Algorithmus 4-7a). Ist diese im Zustand “Ready” – der Fall kann beispielsweise durch eine lang andauernde erste Phase des 2PC-Protokolles eintreten – fügt der Orchestrator *VotiererId* zu *YesVotes* hinzu (sofern nicht

schon enthalten). Dies stellt sicher, daß der Votierer benachrichtigt wird, sobald der Orchestrator entweder *rm_commit* oder *rm_abort* vom Transaktionsmanager empfängt.

Geht die *FORGET*-Nachricht verloren oder kommt zu spät an, dann ist die Schritt-Transaktion beim Eintreffen der *INQUIRY*-Nachricht im Transaktionsstatus "Committed". In diesem Fall antwortet der Orchestrator mit einer *FORGET*-Nachricht.

Ist der Transaktionsstatus der Schritt-Transaktion jedoch "Unknown", müssen zwei Fälle unterschieden werden: Findet auf dem Knoten des Orchestrators keine zur Stufe *StufenId* gehörende Schritt-Transaktion statt, dann antwortet der Orchestrator mit einer *UN_VOTE*-Nachricht. Dies ist im Beispiel in Abbildung 4-12 der Fall. Ist jedoch eine Schritt-Transaktion aktiv, dann kann die *INQUIRY*-Nachricht ignoriert werden. Dies begründet sich wie folgt: Ist eine Schritt-Transaktion aktiv, aber noch im Zustand "Unknown" bedeutet dies, daß entweder das Votierprotokoll noch gar nicht gestartet wurde oder es noch nicht beendet ist. Ist das Votierprotokoll noch nicht gestartet, dann gehört die *INQUIRY*-Nachricht zu einer zuvor ausgeführten Schritt-Transaktion. In diesem Fall erhält der Votierer von der aktuellen Schritt-Transaktion eine neue *VOTE*-Aufforderung, sobald das Votierprotokoll startet. Befindet sich das Votierprotokoll bei Ankunft der *INQUIRY*-Nachricht in der ersten Phase, dann kann die Nachricht entweder noch zu einer alten Schritt-Transaktion oder zur aktuellen gehören. Gehört sie zur alten Schritt-Transaktion, ist zwischenzeitlich eine *VOTE*-Aufforderung entweder beim Votierer angekommen oder zum Votierer unterwegs. Gehört die *INQUIRY* zur aktuellen Schritt-Transaktion, dann ist entweder die Antwort des Votierers verloren gegangen oder die erste Phase wird demnächst abgeschlossen. Ist die Antwort des Votierers nicht verloren gegangen, erhält er in der zweiten Phase auf jeden Fall entweder ein *FORGET* oder *UN_VOTE*. Abbildung 4-13 zeigt eine solche Situation, bei der Phase I des Votierprotokolles durch wiederholten Verlust der *VOTE*-Anforderung an Votierer V_2 lange dauert. Die *INQUIRY* von V_1 kann ignoriert werden, da V_1 in der zweiten Phase eine

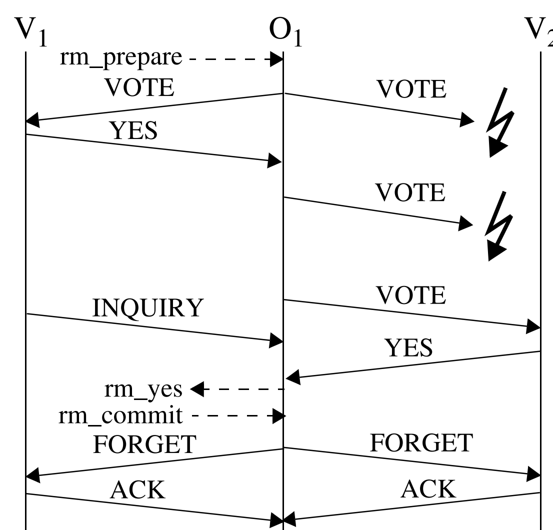


Abbildung 4-13. FORGET als implizite Antwort auf INQUIRY

FORGET-Nachricht erhält und damit die *INQUIRY* implizit beantwortet wird. Ist die Antwort des Votierers auf die *VOTE*-Aufforderung verloren gegangen, wird dies im Normalfall durch das periodische Versenden der *VOTE*-Aufforderungen abgefangen. Es ist jedoch möglich, daß schon vor dem erneuten Versenden dieser Aufforderungen (aber nach der Ankunft der *INQUIRY*) eine Mehrheit der Stimmen erreicht wird. Dann empfängt der Votierer entweder das *FORGET* vom Orchestrator oder der Votierer schickt ein erneutes *INQUIRY*, welches den Orchestrator dann in einem anderen Zustand vorfindet. Abbildung 4-14 zeigt eine solche Situation mit drei Votierern. Votierer V_1 erhält die Votieranforderung und schickt ein *YES*-Votum an den Orchestrator. Votierer V_3 erhält zwar die Votieranforderung, sein *YES*-Votum geht jedoch durch eine in diesem Moment erst aufgetretene, kurzfristige Netzwerkpartitionierung verloren. Votierer V_2 erhält durch Netzwerkpartitionierung erst die dritte Votieranforderung. Wegen der langen Verzögerung schicken V_1 und V_3 *INQUIRY*-Nachrichten. Da der Orchestrator im Zustand "Unknown" ist, werden diese Nachrichten ignoriert. Nachdem V_2 letztendlich das *YES*-Votum schickt, hat der Orchestrator eine Mehrheit und schickt ein *rm_yes*. Der Transaktionsmanager veranlaßt jedoch den Abbruch der Transaktion. Da der Orchestrator nur von V_1 und V_2 YES-Voten bekommen hat, schickt er auch nur diesen ein *UN_VOTE*. V_3 schickt nach einem weiteren Timeout nochmals eine *INQUIRY*. Da auf dem Knoten des Orchestrators jetzt keine Schritt-Transaktion aktiv ist, sendet der Orchestrator wie oben beschrieben eine *UN_VOTE*-Nachricht.

Wenn ein Knoten nach einem Zusammenbruch neu startet und einen konsistenten Zustand herstellt, liest er die Transaktionszustände und die Stufenzustände vom stabilen Speicher. Beim Orchestrator sind zur Herstellung des konsistenten Zustandes einer Stufe nur dann Aktivitäten notwendig, wenn sich der Transaktionsstatus der Stufe in einem der Zustände "Ready" oder "Committed" befindet (vgl. Algorithmus 4-7a, *Recovery()*). Ist der Transaktionsstatus im Zu-

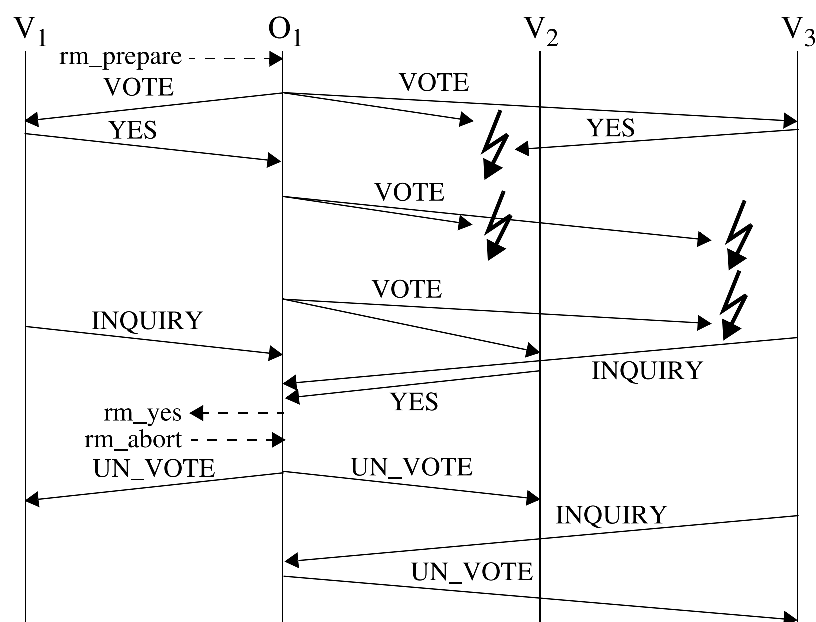


Abbildung 4-14. INQUIRY bei Verlust eines YES-Votums

stand “Ready”, wartet der Orchestrator darauf, vom TM das Ergebnis der Transaktion mitgeteilt zu bekommen. Danach verfährt er wie oben beschrieben. Ist der Transaktionsstatus schon im Zustand “Committed”, dann sendet der Orchestrator analog zu oben zu allen Votierern der Stufe periodisch FORGET, bis er von allen Votierern ein ACK erhalten hat. Danach geht er in den Zustand “Unknown” über und vergißt die Transaktion.

Bei einem Votierer sind zur Herstellung des konsistenten Zustandes einer Stufe nur dann Aktivitäten notwendig, wenn sich der Stufenstatus der Stufe im Zustand “Active” befindet. In diesem Falle muß sowohl das Votierprotokoll in einen konsistenten Zustand gebracht werden als auch die Verarbeitung des Agenten sichergestellt werden. Um das Votierprotokoll in einen konsistenten Zustand zu bringen, sendet der Votierer periodisch zu allen Orchestratoren der Stufe, die sich im *OrchSet* der Stufe befinden, *INQUIRY*-Nachrichten, bis er entweder von allen diesen Orchestratoren eine *UN_VOTE*- bzw. *VOTE*-Aufforderung erhalten hat oder einer der Orchestratoren der Stufe eine *FORGET*-Nachricht geschickt hat (vgl. Algorithmus 4-7b, *Recovery()*). Auf diese Nachrichten reagiert er wie oben beschrieben.

Zu klären bleibt noch die Frage, was nach dem Wiederanlauf eines Knotens mit jenen Agenten in seiner Eingangswarteschlange geschehen soll, deren Transaktionsstatus “Unknown” ist. Man kann annehmen, daß während des Knotenausfalles andere Knoten die Ausführung dieser Agenten übernommen und eventuell auch weitestgehend oder komplett abgeschlossen haben. Dies bedeutet für den einzelnen Agenten, daß er nun – im Vergleich zu dem wieder angelaufenen Knoten – entweder auf einem Knoten mit niedrigerer oder höherer Priorität ausgeführt wird. Will man erreichen, daß, wenn auch nur irgendwie möglich, immer ein Knoten der Stufe mit möglichst hoher Priorität die Schritt-Transaktion abschließt, dann startet der wieder angelaufene Knoten für alle Agenten mit Transaktionsstatus “Unknown” das Auswahlprotokoll und verfährt dann, wie in Abschnitt 4.4.3.2 beschrieben. Ist dies jedoch nicht so wichtig, nimmt der Knoten die Rolle des Beobachters an. Entscheidet man sich für den Start des Auswahlprotokolls bedeutet dies, daß die ganze Arbeit, die der Agent auf einem Knoten mit niedrigerer Priorität bisher in der Stufe erledigt hat, rückgängig gemacht wird (falls die Schritt-Transaktion dort noch nicht abgeschlossen ist). Dies bedeutet nicht nur einen erhöhten Bedarf an Rechenzeit, sondern verzögert die Ausführung des Agenten eventuell auch unnötig da in vielen Fällen die Priorität der Knoten innerhalb einer Stufe willkürlich festgesetzt wird (vgl. auch Abschnitt 4.5). Bei der Entscheidung für die Rolle des Beobachters spart man den Mehrbedarf an Rechenzeit und Ausführungszeit für den Agenten ein, erzwingt aber nicht, daß ein Knoten mit möglichst hoher Priorität den Agent ausführt. Enthält eine Stufe eine oder mehrere Ausnahmebehandlungsknoten (vgl. Abschnitt 4.4.1), kann bei dieser Strategie der Fall eintreten, daß eine Ausnahmebehandlung durchgeführt wird, obwohl ein regulärer Knoten verfügbar gewesen wäre.

4.4.2.5 Adaption an Fehlermodell mit Nachrichtenüberholung

Soll das Protokoll in Systemen eingesetzt werden, in denen Nachrichtenüberholungen im Fehlermodell enthalten sind, sind einige Modifikationen am Votierprotokoll durchzuführen. Kritischster Fall hierbei ist, wenn eine *VOTE*-Aufforderung eines Orchestrators eine *UN_VOTE*-Aufforderung des selben Orchestrators überholt. In diesem Falle erhält der Votierer diese *VOTE*-Aufforderung noch während der sendende Orchestrator in seinem *OrchSet* enthalten ist und reagiert entsprechend. Dies ist dann kritisch, wenn er als Antwort hierauf dann ein *YES*- oder *COND_YES*-Votum schickt. In diesem Falle löscht die zu spät ankommende *UN_VOTE*-Aufforderung den Orchestrator aus dem *OrchSet* obwohl der Votierer eine Stimme für den Orchestrator abgegeben hat. Abhilfe kann geschaffen werden, indem der mit der *VOTE*-Aufforderung übermittelte Zeitstempel mit im *OrchSet* abgespeichert wird. Eine *UN_VOTE*-Aufforderung muß dann zusätzlich jeweils noch den Zeitstempel der Votierrunde mitführen, in der sie abgeschickt wurde (womit sie eindeutig der VOTE-Aufforderung zugeordnet werden kann).

4.4.2.6 Alternativen zum Mehrheitsentscheid

Der im Votierprotokoll verwendete Mehrheitsentscheid hat zwei kleinere Schwächen. Eine Schwäche ist, daß bei dem gewählten Entscheidungsmechanismus immer alle Knoten einer Stufe gleich gewichtet werden. Hierdurch hat ein Knoten mit geringer Verfügbarkeit dasselbe Gewicht wie Knoten mit sehr hoher Verfügbarkeit bzw. ein Knoten mit hoher Priorität in der Stufe dasselbe Gewicht wie ein Ausnahmebehandlungs-Knoten (mit niedriger Priorität). Hier kann Abhilfe geschaffen werden, indem wie beim gewichteten Votieren (vgl. GIFFORD (1979)) einem Knoten mit höherer Priorität/Zuverlässigkeit mehr Stimmen zugeordnet werden als einem Knoten mit niedrigerer Priorität/Zuverlässigkeit. Eine andere Möglichkeit ist die Verwendung von *Coterien* wie in GARCIA-MOLINA UND BARBARA (1985) beschrieben. Die andere Schwäche ist, daß bei zunehmender Anzahl von Netzwerkpartitionen die Wahrscheinlichkeit wesentlich zunimmt, daß in keiner Partition die zum Abschluß der Stufe notwendige Anzahl an Stimmen erlangbar ist. Abhilfe bei lang andauernden Partitionierungen kann hier eine Erweiterung des Algorithmus schaffen, bei der im Falle einer Partitionierung in einer Partition mit einer Stimmenmehrheit dynamisch neu Stimmen zugewiesen werden, so daß nach erneuter Teilung dieser Partition auf jeden Fall in einer der beiden neu entstehenden Partitionen wieder eine Stimmenmehrheit vorhanden ist. BARBARA, GARCIA-MOLINA UND SPAUSTER (1989) und TANG (90) stellen solche Techniken zum dynamischen Zuweisen von Stimmen vor.

4.4.3 Beobachtungs- und Auswahlprotokoll

Die Kombination aus Beobachtungs- und Auswahlprotokoll stellt sicher, daß ein Agent auf einem anderen Knoten der aktuellen Stufe zur Ausführung gelangt, falls der momentane Arbeiterknoten nicht mehr verfügbar ist. Sobald das Beobachtungsprotokoll feststellt, daß der aktu-

elle Arbeiter nicht mehr verfügbar ist, wird das Auswahlprotokoll gestartet. Dieses wählt einen der verbleibenden Knoten der Stufe aus und macht ihn zum neuen Arbeiter.

Algorithmus 4-8 zeigt die beiden miteinander eng verwobenen Protokolle in einer Pseudo-Code-Notation. Um die notwendigen zeitlichen Abläufe des Protokolls zu modellieren, wird ein Zeitmesser verwendet. Mit der Prozedur *startTimer(t)* wird der Zeitmesser gestartet. Sobald die Zeit *t* verstrichen ist, wird die Prozedur *Timer()* aufgerufen. Die Prozedur *resetTimer(t)* initialisiert den Zeitmesser neu sodaß der nächste Aufruf von *Timer()* erst stattfindet, nachdem nach dem Aufruf von *resetTimer(t)* wiederum die Zeit *t* vergangen ist. Die Prozedur *stopTimer()* hält den Zeitmesser an. Die detaillierte Beschreibung der Protokolle erfolgt in den folgenden beiden Abschnitten.

4.4.3.1 Beobachtungsprotokoll

Sobald eine Stufe aktiv wird, d.h. die Schritt-Transaktion der vorhergehenden Stufe erfolgreich abgeschlossen ist und der Agent inklusive Stufen-Record in den Eingangswarteschlangen der Knoten der Stufe erscheint, lesen alle Knoten der Stufe den Stufen-Record ohne ihn aus der Eingangswarteschlange zu entfernen **(1)**[1] und entscheiden, welche initiale Rolle (Arbeiter bzw. Beobachter) sie in der Stufe haben. Der Knoten mit der höchsten Priorität wird zum Arbeiter der Stufe und führt den Agent wie in Abschnitt 4.4.1 beschrieben aus **(2)**. Alle anderen Knoten der Stufe agieren als Beobachter **(3)**.

Den Ablauf des Beobachtungsprotokolles zeigt Abbildung 4-15. Der Arbeiter *W* (engl: worker) sendet periodisch *I_AM_ALIVE*(*StufenId*, *KnotenId*)-Nachrichten an alle anderen Knoten der Stufe **(9)**. Die Beobachter einer Stufe überwachen den Arbeiter, indem sie überprüfen, ob dieser regelmäßig *I_AM_ALIVE*(..)-Nachrichten versendet **(6)**. Sobald bei einem Beobachter die seit dem letzten Empfang einer *I_AM_ALIVE*(..)-Nachricht verstrichene Zeit die bekannte Peri-

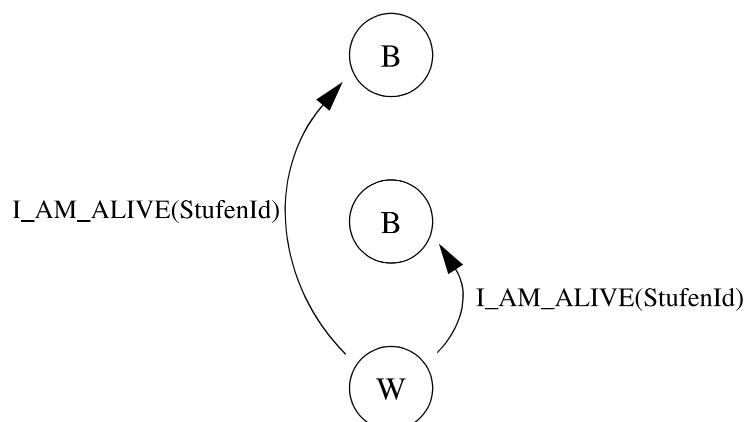


Abbildung 4-15. Beobachtungsprotokoll.

1. Die Zahlen in Klammern beziehen sich auf die Numerierung in Algorithmus 4-8

```
new agent arrived in node input queue{
    read stage record from input queue (1)
    if (node has highest priority in stage){ (2)
        executeAgent()
    }else{ (3)
        mode = observing
        startTimer(t_aliveMax)
}   }

Procedure executeAgent(){
    mode = working
    startTimer(t_p)
    start agent execution (see algorithm 4-2)
}

Receive I_AM_ALIVE(stageId, nId){
    if (mode == working) (4){
        if (node nId has higher priority in stage){ (5)
            higherPrioProcessing()
        }
    } else if (mode == observing){ (6)
        resetTimer(t_aliveMax)
    } else if (mode == selecting){
        if (node nId has higher priority in stage){ (7)
            mode == observing
            resetTimer(t_aliveMax)
        }
}   }

Procedure higherPrioProcessing(){
    Send HIGHER_PRIO(stageId) To
      local orchestrator
    Receive ANSWER From local
      orchestrator
    If (ANSWER == GAVE_UP){ (8)
        mode = observing
        resetTimer(t_aliveMax)
}   }

stage transaction commits{ // on worker
    stopTimer()
}

stage transaction aborts{ // on worker
    startSelection()
}

stage state becomes "Unknown"{
    stopTimer()
}

Procedure Timer(){
    if (mode == working){ (9)
        Send I_AM_ALIVE(stageId, nodeId) To
          all other nodes in stage
        resetTimer(t_p)
    } else if (mode == observing){ (10)
        startSelection()
    } else if (mode == selecting){ (11)
        Send I_AM_SELECTED(stageId) To
          nodes with lower priority in stage
        stopTimer()
        executeAgent()
}   }

Procedure startSelection(){
    stopTimer()
    mode = selecting
    if (node has highest priority in stage){ (12)
        Send I_AM_SELECTED(stageId) To
          other nodes in stage
        executeAgent()
    }else{ (13)
        Send ARE_YOU_THERE(stageId, nodeId)
          To nodes with higher priority in stage
        startTimer(t_selection)
}   }

Receive I_AM_THERE(stageId){
    if (mode == selecting){ (14)
        mode=observing
        resetTimer(t_aliveMax)
}   }

Receive ARE_YOU_THERE(stageId, nId){ (15)
    Send I_AM_THERE(stageId) To nId
    if (mode == observing){ (16)
        startSelection()
}   }

Receive I_AM_SELECTED(stageId){
    if (mode==working){ (17)
        higherPrioProcessing()
    }else{ (18)
        node == observing
        resetTimer(t_aliveMax)
}   }
```

Algorithmus 4-8. Beobachtungs- und Auswahlprotokoll

odendauer zwischen zwei solchen Nachrichten deutlich übersteigt, nimmt der Beobachter an, daß der Arbeiter nicht mehr verfügbar ist und initiiert das Auswahlprotokoll **(10)**.

Empfängt der Arbeiter *W* von einem anderen Knoten der Stufe eine *I_AM_ALIVE*(*StufenId*)-Nachricht **(4)** oder eine *I_AM_SELECTED*(*StufenId*)-Nachricht **(17)** (vgl. Abschnitt 4.4.3.2) gibt es offensichtlich einen weiteren, konkurrierenden Arbeiter *W'* in der Stufe. Hat *W'* eine höhere Priorität als *W* **(5)**, sendet *W* eine *HIGHER_PRIO*(*StufenId*)-Anforderung zum lokalen Orchestrator, um dem Knoten mit der höheren Priorität den Vorrang zu gewähren. Antwortet der Orchestrator mit *GAVE_UP* **(8)**, wird *W* zum Beobachter und beobachtet *W'*. Analog zu der in Abschnitt 4.4.2.4 geführten Diskussion kann diese Strategie dazu führen, daß in unserem Beispiel bei Arbeiter *W* die Ausführung des Agenten abgebrochen wird, obwohl dort die Ausführung schon beinahe beendet ist, wohingegen *W'* erst vor kurzem während einer kurzfristigen Partitionierung in einer anderen Partition zum Arbeiter gewählt wurde und daher die Ausführung auf *W'* gerade erst begonnen hat. Hierdurch verlängert sich auf jeden Fall die Ausführungszeit des Agenten. Eine Alternative ist, auf beiden Arbeitern die Ausführung des Agenten weiterlaufen zu lassen und erst durch das Votierprotokoll zu entscheiden, auf welchem Knoten die Ausführung erfolgreich beendet wird. Diese Strategie benötigt jedoch möglicherweise insgesamt mehr Rechenleistung zur Ausführung des Agenten (vor allem, wenn die Ausführung auf beiden Arbeitern noch annähernd gleich lange dauert).

Da in einem asynchronen System keine oberen Zeitschranken für die Nachrichtenlaufzeit und die Ausführung des Codes, welcher das Protokoll implementiert, existieren, kann es zu einer Initiierung des Auswahlprotokolles kommen, obwohl der Arbeiter die *I_AM_ALIVE*(...)-Nachricht verschickt hat. Wie sich in Abschnitt 4.4.3.2 zeigen wird, kann dies im Extremfall dazu führen, daß selbst innerhalb einer Netzwerkpartition ein zweiter Arbeiter gewählt wird. Um die Wahrscheinlichkeit für das Auftreten dieses Falles zu minimieren, müssen Annahmen über die minimalen und maximalen Nachrichtenlaufzeiten und die maximale Verzögerung eines Absendeereignisses beim Arbeiter getroffen werden. Sei t_p die gewünschte Periodendauer zwischen dem Versenden der *I_AM_ALIVE*(..)-Nachrichten, t_{pv} die angenommene maximale Zeit, um die sich das Absenden dieser Nachrichten verzögert, t_{nmin} die minimale Laufzeit einer Nachricht im Netz und t_{nmax} die angenommene maximale Laufzeit einer Nachricht im Netzwerk. Zwischen den Absendeereignissen zweier *I_AM_ALIVE*(..)-Nachrichten ergibt sich damit eine Maximalzeit von $t_p + t_{pv}$. Im Extremfall verstreicht damit zwischen der Ankunft zweier *I_AM_ALIVE*(..)-Nachrichten die Zeit $t_{aliveMax} = t_p + t_{pv} + t_{nmax} - t_{nmin}$ (vgl. Abbildung 4-16). Es kann daher erst nach dem Verstreichen der Zeit $t_{aliveMax}$ seit dem Empfang der letzten *I_AM_ALIVE*(..)-Nachricht angenommen werden, daß der Arbeiter nicht mehr verfügbar bzw. wegen einer Netzwerkpartitionierung nicht erreichbar ist. Nimmt man für t_{pv} und t_{nmax} eher kleine Werte an, wird auf den Ausfall eines Arbeiters bzw. eine Netzwerkpartitionierung schneller reagiert, die Wahrscheinlichkeit einer nicht notwendigen Initiierung des Auswahlprotokolles steigt jedoch an. Es sei jedoch nochmals betont, daß auch bei der konservativen Annahme großer Werte für t_{pv} und t_{nmax} eine unnötige Initiierung des Auswahlprotokolles nicht ausgeschlossen werden kann.

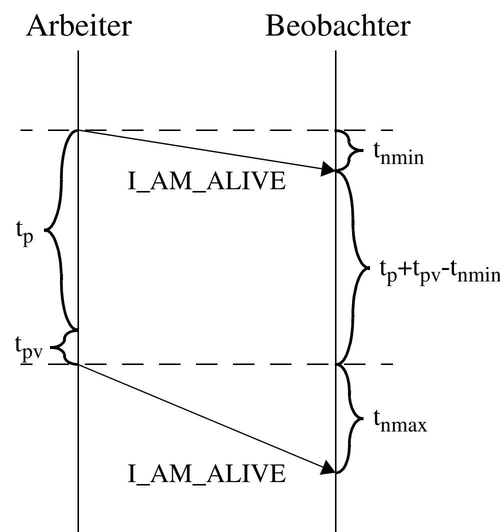


Abbildung 4-16. Maximale Zeit zwischen der Ankunft von I_AM_ALIVE(...)

Neben der Wahl der Werte für t_{pv} und t_{nmax} ist vor allem die Wahl der Periodendauer t_p ausschlaggebend dafür, wie schnell ein nicht mehr verfügbarer Arbeiter von den Beobachtern erkannt wird. Wird hierfür ein sehr kleiner Wert festgelegt, wird sehr schnell festgestellt, daß der Arbeiter nicht mehr verfügbar ist. Dafür ist jedoch der Nachrichtenaufwand sehr hoch. Bei großen Werten hingegen dauert diese Feststellung entsprechend länger, es müssen jedoch wesentlich weniger Nachrichten verschickt werden. Ist beispielsweise t_p größer als die Verweildauer des Agenten auf einem Knoten, müssen im fehlerfreien Fall eventuell überhaupt keine *I_AM_ALIVE*(..)-Nachrichten für diesen Agent verschickt werden. Bei der Festlegung des Wertes von t_p muß also zwischen Schnelligkeit der Erkennung und Nachrichtenaufwand abgewogen werden. Da die Anforderung der Anwendungen daran, wie schnell auf eine Blockierung des Agenten reagiert werden soll, sehr unterschiedlich sind, besteht daher die Möglichkeit, t_p an diese Anforderungen der Anwendungen anzupassen.

4.4.3.2 Auswahlprotokoll

Das Auswahlprotokoll lehnt sich an die Grundideen des in GARCIA-MOLINA (1982) vorgestellten Bully-Algorithmus an. Ein Beobachter, der das Auswahlprotokoll startet, verschickt eine *ARE_YOU_THERE*(*StufenId*, *KnotenId*)-Anforderung an alle Knoten der Stufe mit höherer Priorität **(13)**[1]. Verfügbare Knoten (sowohl Beobachter als auch Arbeiter) antworten hierauf mit einer *I_AM_THERE*(*StufenId*)-Nachricht **(15)** und starten, falls nicht schon geschehen, selbst das Auswahlprotokoll **(16)**. Erhält der Initiator innerhalb einer gewissen Zeit keine Antwort, so ist er damit zum neuen Arbeiter erwählt **(11)**. Er schickt daraufhin eine *I_AM_SELECTED*(*StufenId*)-Nachricht zu allen anderen Knoten der Stufe und beginnt mit der Ausführung des Agenten wie oben beschrieben. Ist der Initiator der Knoten mit der höchsten Priorität in der Stufe

1. Die Zahlen in Klammern beziehen sich auf die Numerierungen in Algorithmus 4-8

(12), entfällt das Versenden der *I_AM_THERE*(..)-Nachrichten und es können sofort die *I_AM_SELECTED*(..)-Nachrichten verschickt werden.

Die Zeit $t_{auswahl}$, nach der ein Knoten sich als "gewählt" deklarieren kann, kann analog zu Abschnitt 4.4.3.1 berechnet werden mit $t_{auswahl} = 2 * t_{nmax} + t_r$, wobei t_r die angenommene Maximalzeit ist, die ein Knoten benötigt um auf *ARE_YOU_THERE*(..) zu antworten, und t_{nmax} die angenommene maximale Laufzeit einer Nachricht im Netzwerk ist. Erhält der Initiator jedoch während dieser Zeit eine *I_AM_THERE*(..)-Nachricht **(14)**, so ist für ihn das Auswahlprotokoll beendet und er wartet auf eine *I_AM_SELECTED*(..)-Nachricht. Sobald diese ankommt, beobachtet er (und alle anderen Beobachter) den neu ausgewählten Arbeiter **(18)**.

Erhält der Initiator eine *I_AM_ALIVE*(..)-Nachricht von einem Knoten mit höherer Priorität **(7)**, dann bricht er das Auswahlprotokoll ab. Dieser Fall kann beispielsweise auftreten, wenn die Nachrichtenlaufzeit der *I_AM_ALIVE*(..)-Nachricht t_{nmax} überschreitet.

Abbildung 4-17 zeigt zwei verschiedene mögliche Abläufe des Auswahlprotokolles in einer Stufe mit 4 Knoten. Im Ausgangszustand ist der Knoten mit der Priorität 1, der Arbeiter der Stufe, ausgefallen. In Abbildung 4-17a entdeckt der Knoten mit der Priorität 2 als erster durch das Beobachtungsprotokoll, daß der Arbeiter nicht mehr erreichbar ist und startet das Auswahlprotokoll. Hierzu schickt er an alle Knoten mit höherer Priorität – hier nur der Knoten mit der Priorität 1 – eine *ARE_YOU_THERE*(..)-Nachricht. Nachdem er keine Antwort erhält, erklärt er sich mittels einer *I_AM_SELECTED*(..)-Nachricht zum Arbeiter. In Abbildung 4-17b entdeckt zu-

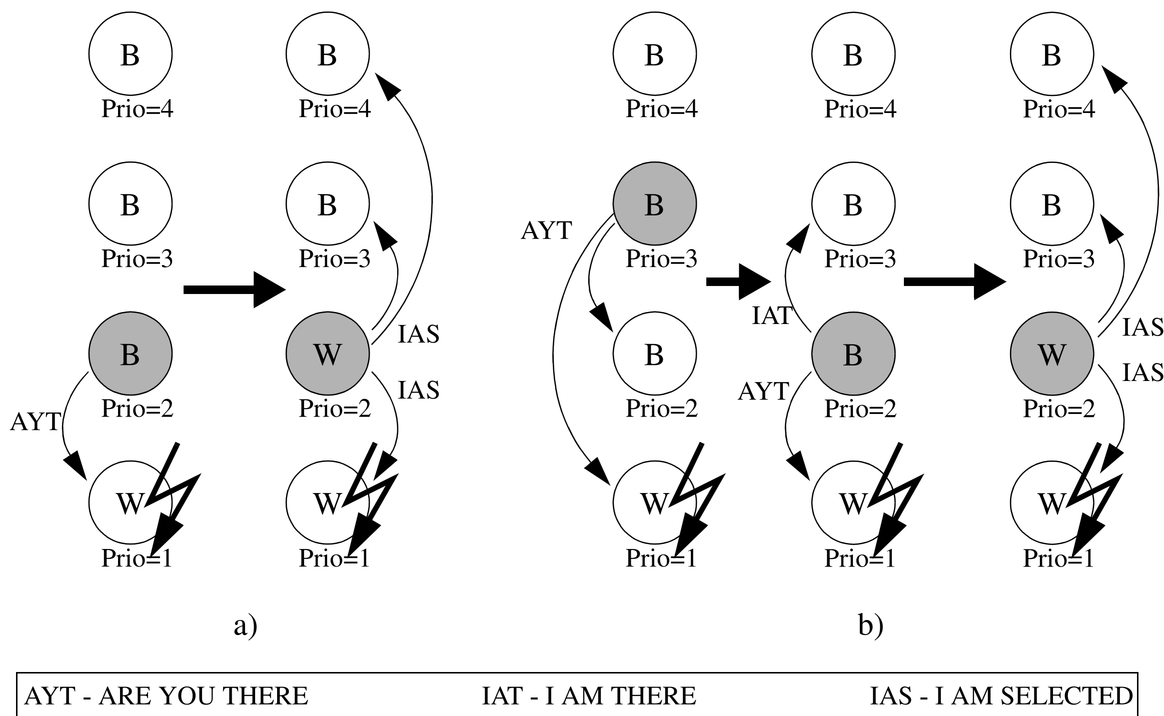


Abbildung 4-17. Ablauf des Auswahlprotokolles

erst der Knoten mit der Priorität 3, daß der Arbeiter nicht mehr erreichbar ist, und startet das Auswahlprotokoll. Hierzu schickt er den Knoten mit Priorität 1 und 2 eine *ARE_YOU_THERE*(..)-Nachricht. Der Knoten mit der Priorität 2 antwortet hierauf mit einer *I_AM_THERE*(..)-Nachricht, was den Knoten mit der Priorität 3 veranlaßt, das Auswahlprotokoll einzustellen. Der Knoten mit Priorität 2 startet dann selbst analog zu a) das Auswahlprotokoll und wird schließlich zum neuen Arbeiter.

Während einer bestehenden Netzwerkpartitionierung bestimmt dieses Protokoll einen Arbeiter für jede Partition. Hierin unterscheidet es sich von gängigen Election-Protokollen, welche genau einen Führer wählen, hierbei aber meistens keine Netzwerkpartitionierungen betrachten. Wie schon weiter oben erwähnt gibt es Fälle, bei denen in einer Netzwerkpartition mehrere Arbeiter einer Stufe existieren. Existieren beispielsweise in zwei (allgemein: n) Partitionen jeweils ein Arbeiter derselben Stufe, kann die nach Vereinigung von Partitionen entstehende größere Partition 2 oder mehr (maximal n) Arbeiter enthalten. Durch die Asynchronität des Systems kann es sogar vorkommen, daß selbst innerhalb einer Netzwerkpartition ein zweiter Arbeiter gewählt wird. Verschickt der aktive Arbeiter beispielsweise wegen extremer Überlastung längere Zeit keine *I_AM_ALIVE(..)*-Nachricht und reagiert wegen dieser Überlastung auch zu lange nicht auf die Nachrichten des Auswahlprotokolles, dann kann einer der Beobachter zum Arbeiter gewählt werden. Existieren mehrere Arbeiter einer Stufe parallel in einer Partition, sorgt spätestens das Votierprotokoll dafür, daß maximal einer der Arbeiter seine Schritt-Transaktion abschließen kann. Je nach gewählter Strategie (siehe weiter oben) geben aber auch alle Arbeiter bis auf den mit der höchsten Priorität auf, nachdem sie von diesem eine *I_AM_ALIVE*(..)-Nachricht erhalten haben.

Die Wahl des Bully-Algorithmus als Vorlage für das Auswahlprotokoll erfolgte wegen seiner Einfachheit und seiner attraktiven Zeitkomplexität (O(1)). Der Nachteil des Algorithmus ist seine Nachrichtenkomplexität von $O(n^2)$ im Worst Case (n ist hier die Anzahl der Knoten der Stufe). Diese ist jedoch in gängigen Stufengrößen (maximal 5 bis 7 Knoten) kein größeres Problem, vor allem da das Auswahlprotokoll ausschließlich im Fehlerfalle verwendet wird. Ergibt sich jedoch die Notwendigkeit, die Anzahl der Nachrichten im Auswahlprotokoll zu minimieren, können beliebige andere Election-Algorithmen (z.B. ABU-AMARA (1988), MASUZAWA ET AL. (1989) oder SINGH (1996)) als Vorlage verwendet werden. Hierbei wird jedoch i.a. die Nachrichtenkomplexität zu Lasten der Zeitkomplexität verbessert.

4.4.4 Korrektheit

In diesem Abschnitt, der auf MAIHÖFER (1997) basiert, wird nachgewiesen, daß mittels des in diesem Kapitel vorgestellten Protokolles Agenten genau-einmal im Sinne von Definition 4-1 ausgeführt werden. Hierzu wird zuerst darauf eingegangen, wie die Korrektheit eines Protokolles nachgewiesen werden kann, bevor der Nachweis selbst geführt wird. In diesem Abschnitt

wird nicht nachgewiesen, daß das Protokoll die Wahrscheinlichkeit der Blockierung des Agenten durch Systemfehler reduziert. Dieser Nachweis wird in Abschnitt 4.6 geführt.

4.4.4.1 Korrektheit eines Protokolles

Um die Korrektheit eines für ein Problem entwickelten verteilten Algorithmus' nachzuweisen ist es notwendig zu zeigen, daß der Algorithmus eine korrekte Lösung für das Problem ist (vgl. beispielsweise TEL (1994)). Hierzu muß nachgewiesen werden, daß der Algorithmus die von dem Problem spezifizierten Eigenschaften besitzt. Da ein Protokoll letztendlich ein verteilter Algorithmus ist, treffen die hier gemachten Aussagen auch auf Protokolle zu.

Viele der geforderten Eigenschaften fallen in eine der zwei essentiellen Kategorien *Sicherheitsanforderungen* (engl.: *safety requirements*) und *Lebendigkeitsanforderungen* (engl.: *liveness requirements*). Um die Korrektheit des Algorithmus' nachzuweisen, muß gezeigt werden, daß diese beiden Anforderungskategorien eingehalten werden.

Eine Sicherheitsanforderung betrifft Eigenschaften, die immer erfüllt sein müssen. Während der Ausführung des Protokolles dürfen diese Eigenschaften zu keinem Zeitpunkt verletzt werden.

Lebendigkeitsanforderungen betreffen Eigenschaften P, die letztendlich erfüllt werden müssen, d.h. die während der Ausführung des Algorithmus zumindest einmal zutreffen. Terminiert der Algorithmus (d.h. hält der Algorithmus an) und die Eigenschaften P wurden während der Ausführung erfüllt, so spricht man von korrekter Terminierung. Terminiert der Algorithmus ohne daß P erfüllt war, spricht man von einer *Verklemmung*.

Ein formaler Beweis der Korrektheit eines Algorithmus ist möglich, indem man den Algorithmus als Transitionssystem formuliert. Der Beweis der Korrektheit des Protokolles würde die Erstellung eines sehr großen Transitionssystems erfordern und würde dementsprechend wenig anschaulich. Daher erfolgt der Beweis informal.

4.4.4.2 Informaler Korrektheitsbeweis

Die Definition der genau-einmal Ausführung von mobilen Agenten (Definition 4-1) fordert, daß der Agent eine laut Reiseroute mögliche, vollständige Folge von Schritten auf den den Schritten zugewiesenen Knoten in der richtigen Reihenfolge ausführt und daß jeder dieser Schritte genau einmal ausgeführt wird. In diesem Abschnitt wird nachgewiesen, daß das in diesem Kapitel vorgestellte Protokoll die genau-einmal Ausführung mobiler Agenten nach dieser Definition sicherstellt. Hierbei wird davon ausgegangen, daß die Programmierung der mobilen Agenten und die Zusammenstellung der Reiseroute korrekt sind, sodaß der Agent bei fehlerfreiem System jede der in der Reiseroute spezifizierten Schrittreihenfolgen tatsächlich ausführen könnte.

Um die genau-einmal Ausführung mobiler Agenten zu gewährleisten, muß das Protokoll den folgenden fünf Anforderungen genügen:

(B1) Der Agent führt innerhalb einer Stufe höchstens einen Schritt einmal aus.

(B2) Innerhalb einer Stufe führt der Agent schließlich in endlicher Zeit einen Schritt aus.

(B3) Der Agent führt in einer Stufe nur laut Reiseroute ausführbare Schritte aus.

(B4) Die Übertragung des Agenten in die nächste Stufe ist fehlerfrei.

(B5) Der Agent durchläuft schließlich alle Stufen.

Die Anforderungen (B1), (B3) und (B4) sind Sicherheitsanforderungen, (B2) und (B5) sind Lebendigkeitsanforderungen.

(B1) Der Agent führt innerhalb einer Stufe höchstens einen Schritt einmal aus.

Diese Anforderung ist eine Sicherheitsanforderung und muß daher zu jedem Zeitpunkt der Ausführung des Agenten zutreffen. Da die Ausführung eines Schrittes des Agenten auf den einzelnen Knoten der Stufen innerhalb von (Schritt-)Transaktionen erfolgt und die Ausführung eines Schrittes des Agenten nur bei erfolgreichem Abschluß einer solchen Schritt-Transaktion Auswirkungen auf den Agent und seine Umgebung (Ressourcen) hat, reicht es aus nachzuweisen, daß in jeder Stufe maximal eine Schritt-Transaktion mit *Commit* erfolgreich abgeschlossen werden kann.

Um eine Schritt-Transaktion auf einem Knoten erfolgreich abzuschließen, muß der sich auf diesem Knoten befindliche Orchestrator an der Transaktion teilnehmen. Für den Fall, daß ein Agent in einer Stufe zwei mal ausgeführt wird, müßte also mindestens zwei mal pro Stufe ein Orchestrator an einer (erfolgreich abgeschlossenen) Schritt-Transaktion teilnehmen. Im folgenden wird zuerst der Spezialfall gezeigt, daß ein Agent nicht auf demselben Knoten einer Stufe zwei mal erfolgreich eine Schritt-Transaktion beenden kann. Danach wird der allgemeinere Fall gezeigt, daß der Agent nicht auf zwei verschiedenen Knoten der Stufe erfolgreich eine Schritt-Transaktion beenden kann. Der Fall, daß der Agent nicht erfolgreich auf mehr als zwei Knoten der Stufe ausgeführt werden kann, läßt sich dann einfach auf den Fall zurückführen, daß der Agent nicht auf zwei verschiedenen Knoten erfolgreich ausgeführt werden kann, und wird daher nicht gesondert betrachtet.

Der Nachweis, daß ein Agent innerhalb einer Stufe nicht mehrfach erfolgreich eine Schritt-Transaktion auf demselben Knoten beenden kann, ist vergleichsweise einfach. Bestandteil einer Schritt-Transaktion ist, daß zu Beginn der Transaktion der Agent und der Stufen-Record der Eingangswarteschlange des ausführenden Knoten entnommen werden. Die Isolationseigenschaft der Transaktionen garantiert nun, daß nachdem eine Schritt-Transaktion T_1 auf den Agent und den Stufen-Record zugegriffen hat eine weitere Transaktion T_2 erst nach dem Abschluß von T_1 wieder auf Agent und Stufen-Record zugreifen kann. Wird T_1 erfolgreich abgeschlossen, dann existieren Agent und Stufen-Record jedoch nicht mehr in der Eingangswarteschlange (sie werden durch T_1 entfernt und die Stufe ist dadurch auf dem Knoten nicht mehr "active"), T_2 kann also den Agent nicht mehr ausführen. Nur wenn T_1 erfolglos abbricht, verbleiben Agent

und Stufen-Record in der Eingangswarteschlange und können von einer weiteren Transaktion T_2 gelesen werden.

Der Nachweis, daß ein Agent nicht auf zwei verschiedenen Knoten der Stufe erfolgreich eine Schritt-Transaktion beenden kann, ist um einiges komplexer. Voraussetzung für das erfolgreiche Abschließen einer Schritt-Transaktion ist, daß der Orchestrator eine Mehrheit an *YES*-Voten erhält. Der Fall, daß auf zwei Knoten einer Stufe die Schritt-Transaktion erfolgreich abgeschlossen wird, kann nur dann eintreten, wenn die Orchestratoren auf beiden Knoten jeweils eine Mehrheit an *YES*-Voten erhalten haben. Dies ist jedoch nur dann möglich, wenn mindestens ein Votierer der Stufe für beide Orchestratoren mit *YES* votiert. Es muß also gezeigt werden, daß ein Votierer immer nur für einen Orchestrator der Stufe mit *YES* votiert.

Seien O_1 und O_2 die Orchestratoren der Knoten, auf denen jeweils eine Schritt-Transaktion der Stufe beendet werden soll. Ohne Beschränkung der Allgemeinheit sei der Knoten des Votierers O_1 der Knoten mit der höheren Priorität. Anhand des Votierers V_x der Stufe wird gezeigt, daß ein Votierer letztendlich nur für einen der beiden Orchestratoren ein *YES*-Votum vergibt.

Zuerst wird der Fall betrachtet, daß V_x die *VOTE*-Aufforderung zuerst von O_1 erhält und mit einem *YES*-Votum antwortet. Erhält nun V_x von O_2 eine *VOTE*-Aufforderung, antwortet V_x laut Protokoll mit einem *NO*-Votum, da entweder O_1 sich im *OrchSet* von V_x befindet und O_2 eine niedrigere Priorität hat als O_1 oder die Stufe auf dem Knoten von V_x schon nicht mehr aktiv ist. Es bekommt also nur O_1 ein *YES*-Votum. Gibt O_1 das von V_x erhaltene Votum mittels einer *UN_VOTE*-Nachricht zurück, bevor die *VOTE*-Anforderung von O_2 ankommt, kann V_x an O_2 ein *YES*-Votum verschicken. Letztendlich hat in diesem Falle nur O_2 ein *YES*-Votum bekommen. Gibt O_1 das von V_x erhaltene Votum zurück, nachdem die *VOTE*-Anforderung von O_2 ankam (und folglich mit einem *NO*-Votum beschieden wurde), hat in dieser “Runde” letztendlich keiner der beiden Orchestratoren von V_x ein Votum erhalten.

Abbildung 4-18 veranschaulicht den Fall, daß zuerst die *VOTE*-Aufforderung von O_2 bei V_x ankommt und diese mit einem *YES*-Votum beschieden wird. Kommt danach die *VOTE*-Aufforderung von O_1 bei V_x an, dann antwortet V_x mit einem *COND_YES*(O_2)-Votum, d.h. das Votum von V_x für O_1 hängt davon ab, wie der Votierer V_2 auf dem Knoten von O_2 für O_1 stimmt. Kommt die *VOTE*-Aufforderung von O_1 bei V_2 an bevor sich O_2 im *Ready*-Zustand befindet, so gibt O_2 seine Voten mittels *UN_VOTE* zurück und V_2 vergibt ein *YES*-Votum für O_1. Damit wandelt sich das *COND_YES*(O_2)-Votum in ein *YES*-Votum. Somit hat also letztendlich nur O1 ein YES-Votum erhalten. Kommt die *VOTE*-Aufforderung von O_1 jedoch bei V_2 an, nachdem O_2 in den *Ready*-Zustand überging, so vergibt V_2 ein *NO*-Votum für O_1. Damit wandelt sich das *COND_YES*(O_2)-Votum in ein *NO*-Votum. Somit hat also letztendlich nur O_2 ein *YES*-Votum erhalten.

Gehen Nachrichten verloren, werden diese durch die in Abschnitt 4.4.2.4 vorgestellten Mechanismen zu einem späteren Zeitpunkt erneut versandt. Dies hat auf die Korrektheit der hier gemachten Aussagen keinen Einfluß. Nachrichtenvertauschungen werden durch das Fehlermodell

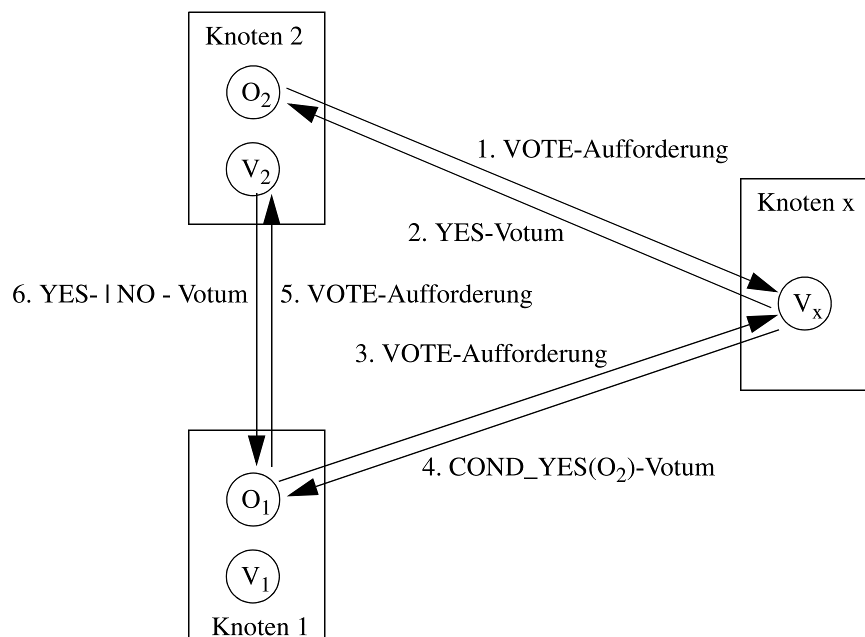


Abbildung 4-18. Auflösung eines COND_YES-Votums

ausgeschlossen, lassen sich jedoch durch die ebenfalls in Abschnitt 4.4.2.4 angedachten Mechanismen so behandeln, daß auch hierdurch die Korrektheit der getroffenen Aussagen nicht gefährdet ist.

Die Beleuchtung der möglichen Fälle bei zwei konkurrierenden Orchestratoren zeigt, daß ein Votierer letztendlich nur einem der beiden ein *YES*-Votum gibt. Hierdurch wird ausgeschlossen, daß beide Orchestratoren eine Stimmenmehrheit erhalten. Somit gibt auch nur maximal einer der beiden Orchestratoren *rm_yes* an den Transaktionsmanager weiter. Es ist also sichergestellt, daß in dem Fall maximal eine der beiden Schritt-Transaktionen erfolgreich beendet wird.

(B2) Innerhalb einer Stufe führt der Agent schließlich in endlicher Zeit einen Schritt aus.

Diese Anforderung ist eine Lebendigkeitsanforderung. Der Nachweis, daß die Anforderung erfüll wird, geschieht in zwei Teilen. Da die Ausführung des Agenten durch den Arbeiter der Stufe geschieht, muß zuerst gezeigt werden, daß das Protokoll immer in einen Zustand übergeht, in dem mindestens ein Arbeiter in der Stufe existiert und somit auf jeden Fall mit der Ausführung des Agenten begonnen wird **(I)**. Die Ausführung des Agenten auf dem Arbeiter geschieht in einer Schritt-Transaktion. Es ist also weiterhin zu zeigen, daß letztendlich ein Arbeiter die auf ihm ausgeführte Schritt-Transaktion erfolgreich beendet **(II)**.

(I) Das Protokoll geht immer in einen Zustand über, in dem (mindestens) ein Arbeiter in der Stufe existiert: Um dies zu zeigen, wird von einem Zustand ausgegangen, in dem kein Arbeiter in der Stufe existiert. Dieser Zustand tritt in zwei verschiedenen Situationen ein – am An-

fang jeder Stufe, d.h. nachdem die Stufe in den Zustand “Active” übergeht, und beim Ausfall des einzigen Arbeiters der Stufe.

Nachdem die Stufe in den Zustand “Active” übergeht, liest im Normalfall nach kurzer Zeit der Knoten der Stufe mit der höchsten Priorität den Stufen-Record aus der Eingangswarteschlange, stellt fest, daß er initialer Arbeiter der Stufe ist und startet die Schritt-Transaktion. Ist dieser Knoten jedoch ausgefallen, verfährt das Protokoll wie beim Ausfall eines Arbeiters.

Die Situation, daß ein Arbeiter ausfällt, wird im Protokoll im Rahmen der Situation “der Arbeiter ist nicht erreichbar” abgehandelt. Dies schließt neben dem Ausfall des Arbeiters auch den Fall der Netzwerkpartitionierung mit ein. Existiert kein Arbeiter in der Stufe, dann existiert zumindest eine Netzwerkpartition in der mindestens ein Beobachter der Stufe vorhanden ist. Dies kann angenommen werden, da nach dem Fehlermodell Knotenausfälle nur temporär sind und daher nach dem Neustart eines Knotens der Stufe dieser sich als Beobachter in einer Netzwerkpartition befindet (die im Extremfall nur aus diesem Knoten besteht). Sobald einer der in der Partition vorhandenen Beobachter feststellt, daß keine *I_AM_ALIVE*-Benachrichtigungen bei ihm ankommen, startet dieser Beobachter *O* das Auswahlprotokoll. *O* verschickt dazu an alle Knoten der Stufe mit höherer Priorität eine *ARE_YOU_THERE*-Nachricht. Existiert ein solcher Knoten *O*’ in der Netzwerkpartition von *O*, so antwortet dieser und startet selbst das Auswahlprotokoll. O beendet in diesem Fall für sich das Auswahlprotokoll. Letztendlich wird von den (funktionstüchtigen) Knoten, die sich in der Partition befinden, derjenige mit der höchsten Priorität das Auswahlprotokoll gestartet haben und keine Antwort auf seine verschickten *ARE_YOU_THERE*-Nachrichten erhalten. Darufhin erklärt er sich zum Arbeiter und beginnt mit der Ausführung der Schritt-Transaktion. Das Protokoll stellt also sogar sicher, daß in jeder Netzwerkpartition, die Knoten der Stufe enthält, letztendlich ein Arbeiter existiert.

(II) Mindestens ein Arbeiter beendet letztendlich erfolgreich die durch ihn ausgeführte Schritt-Transaktion: Um dies nachzuweisen, müssen die Gründe für den Abbruch einer Transaktion betrachtet werden, und es muß gezeigt werden, daß dadurch die Vollendung einer Schritt-Transaktion innerhalb einer Stufe höchstens verzögert, nicht aber verhindert wird. Man kann generell drei verschiedene Gründe unterscheiden: der Knoten fällt während der Ausführung der Transaktion aus *(a)*, der Agent selbst bricht die Transaktion ab *(b)* oder die Transaktion wird von der Transaktionsverwaltung abgebrochen *(c)*.

(II a) Knotenausfall während der Transaktion: Fällt der Arbeiter während der Ausführung des Agenten aus, dann wird wie oben gezeigt ein neuer Arbeiter bestimmt, der erneut eine Schritt-Transaktion startet. Fällt der Arbeiter hierbei aus nachdem im Votierprotokoll eine Stimmenmehrheit gesammelt wurde, so ist der Agent blockiert, bis der Arbeiter neu startet (siehe weiter unten). Um jedoch das (erfolgreiche) Beenden einer Schritt-Transaktion in einer Stufe durch Ausfall der Arbeiter unmöglich (bzw. beliebig unwahrscheinlich) zu machen, müßte die Zeit, die zur Ausführung einer Schritt-Transaktion benötigt wird, deutlich über der mittleren Zeit zwischen zwei Ausfällen eines Knotens liegen. Wie jedoch schon in Abschnitt 4.3 erwähnt soll-

te bei Verwendung des blockierungsfreien Protokolles ein einzelner Schritt eines Agenten von eher kurzer Dauer sein, da sonst die von ihm verwendeten Ressourcen zu lange blockiert werden. Daher kann davon ausgegangen werden, daß die Ausführungszeit eines Schrittes eines Agenten relativ zum mittleren Zeitraum zwischen zwei Ausfällen eines Knotens eher gering ist und deshalb die Ausführung des Agenten durch wiederholten Knotenausfall nicht unendlich verzögert wird.

(II b) Der Agent bricht die Transaktion ab: Laut der weiter oben getroffenen Annahme ist die Programmierung des Agenten korrekt, sodaß der Agent die in der Reiseroute spezifizierten Schritte auch letztendlich ausführen kann. Der Agent selbst bricht also die Ausführung eines Schrittes im Normalfall nicht selbst ab. Lediglich in dem Fall, daß eine Ressource kurzfristig nicht verfügbar ist, kann der Agent die Ausführung eines Schrittes abbrechen, um die Ausführung eines (anderen) Schrittes auf einem anderen Knoten der Stufe zu ermöglichen. Da davon ausgegangen wird, daß die benötigten Ressourcen letztendlich jedoch zur Verfügung stehen, kann der Agent letztendlich auch aus seiner Sicht die Schritt-Transaktion beenden.

(II c) Abbruch der Transaktion durch das Transaktionsmanagement: Der Abbruch der Schritt-Transaktion durch das Transaktionsmanagement erfolgt, wenn die Transaktion aus technischen Gründen zum momentanen Zeitpunkt nicht erfolgreich beendet werden kann. Dies kann beispielsweise durch eine *Verklemmung* verursacht werden. Eine andere mögliche Ursache ist, daß einer der an der Transaktion beteiligten Ressourcenmanager die Transaktion nicht erfolgreich abschließen kann. Hierbei kann es sich um eine der Ressourcen handeln, auf die der Agent während der Ausführung des Schrittes zugreift oder aber um eine der durch das Protokoll verwendeten Ressourcen (Eingangswarteschlangen, Orchestrator). Man kann die Gründe für den Abbruch einer Transaktion durch das Transaktionsmanagement folglich in zwei Gruppen gliedern.

Auf der einen Seite stehen die Abbruchgründe, die auch ohne das hier untersuchte Protokoll auftreten würden, beispielsweise Verklemmungen und vom Agenten genutzte Ressourcen können die Transaktion nicht beenden. Nach den oben getroffenen Annahmen – der Agent kann im fehlerfreien System seine Ausführung beenden, Fehler sind nur temporär – verhindern diese aber die erfolgreiche Ausführung einer Schritt-Transaktion nicht dauerhaft.

Auf der anderen Seite stehen die durch das Protokoll zusätzlich eingeführten Abbruchgründe: Orchestrator und/oder Eingangswarteschlange können die Transaktion nicht abschließen. Es muß also noch nachgewiesen werden, daß trotz diesen Abbruchgründen die Schritt-Transaktion letztendlich erfolgreich abgeschlossen werden kann. Die Eingangswarteschlange des Knotens, der die Schritt-Transaktion ausführt, kann die Schritt-Transaktion normalerweise problemlos abschließen, da es keinerlei Zugriffskonflikte mit anderen Transaktionen gibt, ebenso die Eingangswarteschlangen der Knoten der nächsten Stufe. Der Ausfall eines der Knoten der nächsten Stufe vor Abschluß der Schritt-Transaktion kann allenfalls den Abbruch der gerade ablaufenden Schritt-Transaktion verursachen. Die daraufhin neu gestartete Schritt-Transaktion wird dann

den ausgefallenen Knoten nicht mehr in die nächste Stufe aufnehmen, da die Verfügbarkeit dieser Knoten im voraus getestet wird.

Hiermit bleibt nur noch zu zeigen, daß auch durch die Orchestratoren die erfolgreiche Ausführung der Schritt-Transaktion nicht verhindert wird. Ein Orchestrator verhindert den erfolgreichen Abschluß einer Schritt-Transaktion, d.h. er sendet ein *rm_no* an den Transaktionsmanager, nur dann, wenn er keine Mehrheit an Stimmen im Votierprotokoll mehr erhalten kann, d.h. wenn er von mindestens der Hälfte der Votierer ein *NO*-Votum erhalten hat. Es ist also zu zeigen, daß schließlich ein Orchestrator in einer Stufe eine Stimmenmehrheit erhält. Zur Durchführung des Nachweises kann man zwei Fälle unterscheiden:

Fall 1: In der Stufe ist nur ein Arbeiter aktiv. Ist innerhalb einer Stufe nur ein einzelner Arbeiter aktiv, dann erhält dessen Orchestrator *O* die Stimmen aller in seiner Netzwerkpartition vorhandenen Votierer. Ergibt sich daraus keine Mehrheit, kann dies zwei Gründe haben. Eine Möglichkeit ist, daß die Mehrzahl der Knoten der Stufe entweder ausgefallen ist oder sich in einer anderen Netzwerkpartition befindet. Netzwerkpartitionierung und Knotenausfall sind laut Fehlermodell nur temporäre Fehler, sodaß nach der Behebung der Fehler in endlicher Zeit eine Mehrheit zustande kommt. Wird während des Bestehens der Netzwerkpartitionierung ein weiterer Arbeiter bestimmt, gelten die Ausführungen des zweiten Falls.

Die andere Möglichkeit ist, daß es zu einem früheren Zeitpunkt der Stufe einen Arbeiter gab, bei dem entweder der Abschluß der Schritt-Transaktion nicht erfolgreich war und dessen Orchestrator *O*' zum Beispiel wegen Knotenausfall oder Netzwerkpartitionierung die erhaltenen Stimmen nicht zurückgeben konnte oder bei dem nach erfolgreichem Abschluß der ersten Stufe des Votierprotokolles (d.h. er befindet sich im Zustand "Ready") und der ersten Phase des 2PC-Protokolles (d.h. der Transaktionsmanager der Transaktion befindet sich im Zustand Committed) der Knoten zusammengebrochen ist. Auch hier gilt, daß der verursachende Fehler nach endlicher Zeit behoben wird. Das Protokoll ist in diesen Fällen jedoch blockiert, bis der Ausfall des Knotens von *O*' wieder behoben ist. Wurde die Schritt-Transaktion nicht erfolgreich beendet, so gibt *O*' – veranlaßt durch die von den Votierern versandten *INQUIRY*-Aufforderungen – die erhaltenen Voten mittels *UN_VOTE* zurück. Hat *O* zu diesem Zeitpunkt jedoch schon von mindestens der Hälfte der Orchestratoren ein *NO*-Votum erhalten (weil der Knoten von *O*' die höhere Priorität hat), so hat er die Schritt-Transaktion abgebrochen und die erhaltenen Voten zurückgegeben. In diesem Falle sorgen Überwachungs- und Auswahlprotokoll für die Bestimmung eines neuen Arbeiters. Wurde die Schritt-Transaktion von *O*' jedoch erfolgreich beendet, so wird *O*' in der zweiten Phase des Votierprotokolles die Stufe vollends beenden.

Fall 2: In der Stufe sind mehrere Arbeiter aktiv. Gibt es in einer Stufe zwar mehr als einen Arbeiter, versucht jedoch nur einer der Arbeiter, die Stufe zum momentanen Zeitpunkt abzuschließen (d.h. nur einer der Orchestratoren führt momentan das Votierprotokoll durch), dann gelten hierfür sinngemäß die Aussagen aus Fall 1.

Interessant ist der Fall, daß mehr als ein Orchestrator in einer Stufe aktiv ist (d.h. es gibt mehr als einen Arbeiter, dessen Orchestrator zum momentanen Zeitpunkt das Votierprotokoll durchführt). In diesem Fall muß sichergestellt sein, daß letztendlich einer der aktiven Orchestratoren eine Mehrheit erhält. Ob ein Orchestrator eine Stimmenmehrheit bekommt, hängt nicht unbedingt davon ab, daß er sich in einer Netzwerkpartition befindet, welche genug (d.h. eine Mehrheit) funktionstüchtiger Knoten der Stufe des Orchestrators enthält. Dies verdeutlicht man sich am besten mit dem folgenden zwar unwahrscheinlichen, aber möglichen Szenario: Eine Stufe enthält 7 Knoten K_1, K_2, ..., K_7. K_1 ist der initiale Arbeiter mit Orchestrator O_1. Nach einer Netzwerkpartitionierung in zwei Partitionen existieren zwei Arbeiter. Hierbei sei K_1 der Arbeiter in der Partition, die die Knoten K_1, K_2, K_3 und K_4 enthält, und K_5 der neu gewählte Arbeiter (mit Orchestrator O_5) der Partition mit den Knoten K_5, K_6 und K_7. O_5 startet das Votierprotokoll zuerst (der Schritt auf K_5 ist wesentlich weniger aufwendig) und erhält die Stimmen der Votierer der Knoten K_5, K_6 und K_7. Dies alleine ergibt noch keine Mehrheit. Bevor O_1 mit dem Votierprotokoll beginnt, teilen sich beide Partitionen nochmals, sodaß nun vier Partitionen entstehen: eine Partition mit den Knoten K_1, K_2 und K_3, eine Partition mit dem Knoten K_4, eine Partition mit dem Knoten K_5 und eine Partition mit den Knoten K_6 und K_7. Bevor das Beobachtungsprotokoll in den Partitionen ohne Arbeiter nun das Auswahlprotokoll startet, vereinigen sich jeweils zwei Partitionen, sodaß sich die folgende Partitionierung ergibt: eine Partition mit den Knoten K_1, K_2, K_3, K_6 und K_7 und eine Partition mit den Knoten K_4 und K_5. O_5 erhält nun auch noch die Stimme von dem Votierer auf K_4 und hat damit eine Stimmenmehrheit gesammelt. Hierbei war K_5 niemals in einer Partition, in welcher eine Mehrheit der Knoten der Stufe enthalten war. K_1 hingegen befindet sich am Schluß zwar in einer Partition, die eine große Mehrheit der Knoten der Stufe enthält, kann aber dort die Mehrheit der Stimmen nicht bekommen. Hat K_1 eine höhere Priorität, dann bekommt O_1 zwar von den Votierern der Knoten K_6 und K_7 jeweils ein *COND_YES*(O_5)-Votum, kann diese Voten jedoch in keine *YES*-Voten umwandeln, da O_5 schon eine Mehrheit und damit die Schritt-Transaktion abgeschlossen hat.

Dieses Beispiel zeigt, daß nachgewiesen werden muß, daß bei wechselnden Partitionierungen schließlich ein Orchestrator eine Stimmenmehrheit erhält. Um die Komplexität der Betrachtung in Grenzen zu halten, wird dies in einem ersten Schritt anhand zweier aktiver Orchestratoren illustriert und danach in einem zweiten Schritt auf beliebig viele Orchestratoren verallgemeinert.

Schritt 1: Zwei Orchestratoren führen das Votierprotokoll zur selben Zeit durch. Seien die zu betrachtenden Orchestratoren die Orchestratoren O_1 und O_2, wobei der Knoten von O_1 die höhere Priorität hat. Seien ohne Beschränkung der Allgemeinheit die beiden Orchestratoren nicht in derselben Partition, wenn bei einem der Orchestratoren das Votierprotokoll startet. Sobald O_1 das Votierprotokoll beginnt, kontaktiert er erfolgreich alle Knoten der Stufe in seiner Partition. Jeder der kontaktierten Knoten antworten entweder mit einem *YES*-Votum (wenn er noch kein Votum an O_2 geschickt hat) oder mit einem *COND_YES*(O_2)-Votum, falls er schon ein Votum an O_2 vergeben hat. Dasselbe geschieht, wenn O_2 das Votierprotokoll startet, jedoch antworten

hier die Knoten der Partition, die schon ein *YES*-Votum an O_1 vergeben haben mit einem *NO*-Votum (da der Knoten von O_2 eine niedrigere Priorität hat). Kommen neue Knoten (außer dem Knoten von O_2) der Stufe zur Partition von O_1 dazu, die noch kein Votum für O_1 abgegeben haben, erhöht sich dadurch die Anzahl der *YES*-Voten bzw. *COND_YES*(O_2)-Voten von O_1. Verlassen Knoten der Stufe die Partition von O_1, ändert sich an der Stimmenzahl von O_1 nichts. Bei O_2 hingegen erhöht sich in dem Fall, daß neue Knoten (d.h. Knoten der Stufe, die noch kein Votum für O_2 abgegeben haben) außer dem Knoten von O_1 zu der Partition hinzukommen, entweder die Anzahl der *YES*-Voten oder die Anzahl der *NO*-Voten. Auch hier ändert sich die Stimmenanzahl von O_2 nicht, wenn ein Knoten der Stufe die Partition verläßt. Erreicht hierbei schon einer der Orchestratoren O_1 oder O_2 eine Stimmenmehrheit, so ist das Ziel erreicht. Erreicht O_2 eine ausreichende Zahl an *NO*-Voten (halbe Knotenzahl der Stufe), dann kann O_2 keine Mehrheit mehr erreichen, beendet das Votierprotokoll und gibt die erhaltenen Stimmen zurück. Die Knoten, die von O_2 die Stimme zurückbekommen haben und die noch kein *COND_YES* an O_1 geschickt haben, können auf eine *VOTE*-Aufforderung gleich mit einem *YES* antworten. Erreicht keiner der beiden Orchestratoren auf diese Art eine Mehrheit – beispielsweise könnten beide Orchestratoren genau die Hälfte der Stimmen der Stufe besitzen – dann tritt irgendwann, da alle Fehler und folglich auch Netzwerkpartitionen nur temporär sind, der Zustand ein, daß sich O_1 und O_2 in derselben Partition befinden. In diesem Fall bekommt O_2 von O_1 ein *NO*-Votum (da der Knoten von O_1 die höhere Priorität hat). Erreicht die *VOTE*-Anfrage von O_1 den Votierer des Knotens von O_2 bevor O_2 eine Stimmenmehrheit gesammelt hat, so gibt O_2 auf, gibt sämtliche erhaltenen *YES*-Voten zurück und der Votierer des Knotens von O_2 schickt ein *YES*-Votum an O_1. Damit werden alle *COND_YES*(O_2)-Voten zu *YES*-Voten. Es existiert jetzt nur noch ein Orchestrator, der letztendlich auch eine Mehrheit der Stimmen erhalten wird.

Schritt 2: Mehr als 2 Orchestratoren führen das Votierprotokoll zur selben Zeit durch. Die Verallgemeinerung auf mehrere Orchestratoren in einer Stufe ist offensichtlich: Sei O_{max} der Orchestrator des funktionstüchtigen Knotens mit der höchsten Priorität der Stufe. Die Anzahl der *YES*- und *COND_YES*-Voten von O_{max} nimmt monoton zu, da es keinen höher priorisierten Knoten gibt, wegen dem er das Votierprotokoll aufgeben muß. Erreicht keiner der Orchestratoren der Knoten mit niedrigerer Priorität eine Mehrheit der Stimmen, dann geben diese alle das Votierprotokoll zu Gunsten von O_{max} auf, sobald sie sich mit ihm in einer Partition befinden. Da Partitionierungen nur von kurzer Dauer sind, werden sich alle Orchestratoren und alle Knoten der Stufe schließlich zu irgend einem Zeitpunkt nach Beginn des Votierprotokolles bei O_{max} in derselben Partition wie O_{max} befunden haben. Die Orchestratoren haben dabei zu Gunsten von O_{max} das Votierprotokoll beendet (sofern sie nicht selbst schon eine Stimmenmehrheit hatten), die Votierer der Knoten haben dabei jeweils ein *YES*- oder *COND_YES*-Votum für O_{max} abgegeben. Dies ermöglicht O_{max}, schließlich eine Mehrheit der Stimmen zu erlangen.

Es hat sich also gezeigt, daß in jeder Stufe schließlich (mindestens) einer der Orchestratoren eine Stimmenmehrheit erlangt. Es folgt also, daß keiner der Gründe für einen Transaktionsabbruch die Schritt-Transaktion nachhaltig am erfolgreichen Transaktionsabschluß hindert. In

Kombination mit dem Nachweis, daß das Protokoll schließlich immer in einen Zustand übergeht, in dem mindestens ein Arbeiter in der Stufe existiert, ist damit erwiesen, daß das Protokoll der Anforderung (B2) genügt.

(B3) Der Agent führt in einer Stufe nur laut Reiseroute ausführbare Schritte aus.

Diese Anforderung ist eine Sicherheitsanforderung und muß daher zu jedem Zeitpunkt der Ausführung des Agenten zutreffen. Wie in Abschnitt 4.4.1 beschrieben besteht eine Stufe aus *regulären Knoten* und *Ausnahmebehandlungs-Knoten.* Reguläre Knoten der Stufe sind Knoten, auf denen der Agent zum Zeitpunkt der Ausführung der Stufe laut Reiseroute einen Schritt ausführen kann. In Abschnitt 4.5 wird beschrieben werden, wie die Knoten einer Stufe S_{i+1} ermittelt werden. Das Grundprinzip besteht darin, daß nach Ausführung eines Schrittes in der Stufe S_i ermittelt wird, welche Schritte auf welchen Knoten laut Reiseroute in S_{i+1} ausgeführt werden können. Ergeben sich hierbei ausreichend (momentan erreichbare) Knoten zur Bildung der Stufe, so besitzt die Stufe nur reguläre Knoten. Für den Fall, daß sich nicht genügend Knoten ergeben, werden beliebige (erreichbare) Knoten, die einen Agent ausführen können, mit in die Stufe als Ausnahmebehandlungs-Knoten aufgenommen. Der Agent wird von der Schritt-Transaktion in S_i nur auf die so ermittelten Knoten transportiert.

Wird die Schritt-Transaktion in S_{i+1} auf einem regulären Knoten ausgeführt, so führt dieser den laut Reiseroute auf ihm auszuführenden Schritt aus. Da nur ein Knoten die Schritt-Transaktion erfolgreich beenden kann (vgl. (B1)), ist in diesem Fall die Anforderung (B3) erfüllt.

Wird die Schritt-Transaktion jedoch auf einem Ausnahmebehandlungs-Knoten ausgeführt, so ist dies kein ursprünglich in der Reiseroute spezifizierter Schritt. Wie in Abschnitt 4.4.1 beschrieben wird jedoch die Ausführung der Ausnahmebehandlung auf diesem Knoten in die Reiseroute eingefügt, sodaß auch in diesem Fall letztendlich die Anforderung (B3) erfüllt ist.

(B4) Die Übertragung des Agenten in die nächste Stufe ist fehlerfrei.

Diese Anforderung ist eine Sicherheitsanforderung und muß daher zu jedem Zeitpunkt der Ausführung des Agenten zutreffen. Unter fehlerfreier Übertragung in die nächste Stufe wird verstanden, daß die korrekten Daten zum korrekten Zeitpunkt auf die korrekten Knoten übertragen werden.

Der korrekte Zeitpunkt der Übertragung des Agenten in die Stufe S_{i+1} ist, wenn in Stufe S_i ein Schritt erfolgreich ausgeführt worden ist und der Agent noch weitere Schritte auszuführen hat. Hat der Agent keine weiteren Schritte zu erledigen, ist seine Ausführung beendet; ein Transport in eine nachfolgende Stufe ist nicht notwendig. Es muß sichergestellt werden, daß die erfolgreiche Ausführung des Schrittes in S_i und der Transport zu den Knoten von S_{i+1} atomar ist. Dies bedeutet einerseits, daß sichergestellt sein muß, daß bei erfolgreicher Ausführung des Schrittes in S_i der Agent auf alle Knoten der folgenden Stufe S_{i+1} transportiert wird (sofern die Ausführung des Agenten nicht beendet ist). Es muß jedoch ebenfalls sichergestellt werden, daß wenn der Agent auf die Knoten der Stufe S_{i+1} transportiert wird, daß dann der Schritt in S_i tatsächlich

erfolgreich ausgeführt wurde. Das Protokoll erfüllt diese beiden Bedingungen, indem das Schreiben in die Eingangswarteschlangen der Knoten der Stufe S_{i+1} innerhalb der Schritt-Transaktion der Stufe S_i erfolgt. Ist die Schritt-Transaktion erfolgreich, so ist durch die Transaktionseigenschaften sichergestellt, daß die Resultate der Aktionen des Agenten während des Schrittes in Stufe S_i dauerhaft nach außen sichtbar werden und daß der Agent auf allen Knoten der Stufe S_{i+1} angekommen ist. Daß in Stufe S_i in diesem Fall kein weiterer Schritt ausgeführt wird, wurde schon beim Beweis der Anforderung (B1) ausgeführt. Schlägt die Schritt-Transaktion jedoch fehl, so ist sichergestellt, daß alle Auswirkungen der Transaktion – insbesondere die Resultate der Aktionen des Agenten in Stufe S_i und dessen Transport in Stufe S_{i+1} – rückgängig gemacht werden und nach außen nicht sichtbar werden.

Übertragung der korrekten Daten von Stufe S_i auf die (korrekten) Knoten der Stufe S_{i+1} bedeutet sowohl, daß die korrekten Daten in Richtung der korrekten Knoten abgeschickt werden als auch daß die korrekten Daten bei den Knoten von S_{i+1} ankommen. Die zu verschickenden Daten sind der Code und der Datenzustand des Agenten nach der Ausführung des Schrittes in Stufe S_i und der Stufen-Record der Stufe S_{i+1}. Hier muß angenommen werden, daß die Implementierung sowohl den Code und den Datenzustand des Agenten als auch den Stufen-Record der nachfolgenden Stufe korrekt ermittelt und diese Daten in Richtung der im Stufen-Record aufgeführten Knoten abschickt. Sowohl Veränderung der Daten auf dem Transport als auch Transport zu falschen Zielknoten schließt das zugrundeliegende Fehlermodell aus. Ein Verlust der versendeten Daten kann laut Fehlermodell nur durch Netzwerkpartitionierung entstehen. Kann in diesem Fall nicht sicher festgestellt werden, ob die Daten verloren gingen, bricht das Protokoll die Transaktion vorsichtshalber ab, ansonsten werden die Knoten der Stufe S_{i+1} neu ermittelt und die entsprechenden Informationen neu verteilt (vgl. auch Abschnitt 4.4.1).

Das Protokoll stellt also immer sicher, daß die Übertragung des Agenten in die nächste Stufe fehlerfrei ist.

(B5) Der Agent durchläuft schließlich alle Stufen.

Diese Anforderung ist eine Lebendigkeitsanforderung. Der Nachweis basiert auf den Argumentationen zu den Anforderungen (B2) und (B4). In (B2) wurde gezeigt, daß das Protokoll die Ausführung des Agenten in einer Stufe auf jeden Fall beginnt und letztendlich in der Stufe mindestens einen Schritt innerhalb einer Schritt-Transaktion durchführt (in Kombination mit (B1): genau einen Schritt). In (B4) wurde gezeigt, daß der Agent, sofern noch weitere Schritte durchzuführen sind, noch innerhalb der Schritt-Transaktion einer Stufe jeweils in die nächste Stufe transportiert wird. Somit wird also in jeder Stufe garantiert, daß ein Schritt des Agenten ausgeführt wird und der Agent schließlich in die nächste Stufe übertragen wird, solange noch weitere Schritte auszuführen sind. Sobald der Agent also in den Eingangswarteschlangen der Knoten der ersten Stufe liegt, sorgt das Protokoll dafür, daß wiederholt ein Schritt ausgeführt und der Agent in die nächste Stufe übertragen wird, bis der Agent keine weiteren Schritte mehr auszuführen hat.

4.5 Kommunikationsaufwand und Stufenkonstruktion

In diesem Abschnitt soll der durch die Protokolle eingeführte Kommunikationsaufwand über das die Knoten verbindende Netzwerk betrachtet werden. Zuerst wird der Kommunikationsaufwand ermittelt, wenn keine genau-einmal Ausführung des Agenten sichergestellt wird. Danach wird jeweils der Aufwand für das Basisprotokoll und das erweiterte blockierungsfreie Protokoll ermittelt. Eine Diskussion einiger Möglichkeiten zur Reduktion des Kommunikationsmehraufwandes wird zeigen, daß der Kommunikationsaufwand des blockierungsfreien Protokolles von der Zusammensetzung einer Stufe abhängt. Diese Erkenntnis wird dann schließlich in einem Algorithmus zur Bestimmung der Knoten einer Stufe verwertet, der den notwendigen Kommunikationsaufwand verringert.

4.5.1 Kommunikationsaufwand der Protokolle

Gängige Kriterien zur Beurteilung verteilter Algorithmen hinsichtlich des Kommunikationsaufwandes sind die Anzahl der zu übertragenden Nachrichten und die Menge der zu übertragenden Daten. Die folgenden Betrachtungen konzentrieren sich auf die Ermittlung der Anzahl zu übertragender Nachrichten. Die Menge der zu übertragenden Daten wird nur insofern berücksichtigt, als daß im Laufe der Ermittlung der Nachrichtenanzahl (nicht bei den resultierenden Gesamtzahlen) und beim Vergleich zwischen den Protokollen zwischen "kurzen" Nachrichten (bis wenige 100 Bytes) und "langen" Nachrichten bzw. Nachrichtentypen (z.B. Migrationsnachrichten, Bestätigungsnachrichten,...) unterschieden wird. Bei der Ermittlung des Kommunikationsaufwandes wird von dem ungünstigeren Fall ausgegangen, daß der Agent zur Ausführung des ersten Schrittes zuerst auf den ausführenden Knoten transportiert werden muß und daß ein Agent niemals zwei planmäßige Schritte nacheinander auf demselben Knoten ausführt, d.h. die ermittelten Werte sind eine obere Grenze.

Kommunikationsaufwand bei ungesicherter Ausführung des Agenten.

Bei ungesicherter Ausführung des Agenten beschränkt sich die Kommunikation auf die zur Migration notwendigen Nachrichten. Für die Migration von einem Knoten auf einen anderen sind minimal zwei Nachrichten notwendig: der Ausgangsknoten verschickt eine (umfangreiche) Migrationsnachricht zur Übertragung des Agentencodes und des Datenzustandes des Agenten (inklusive Reiseroute) zum Zielknoten der Migration, und der Zielknoten bestätigt die Ankunft des Agenten mit einer (kurzen) Bestätigungsnachricht. Treten keine Fehler auf, so werden also für die Ausführung eines Schrittes eine Migrationsnachricht und eine Bestätigungsnachricht benötigt. Geht eine dieser Nachrichten verloren, sind zusätzliche Nachrichten zur Übertragung des Agenten notwendig.

Bei fehlerfreier Ausführung des Agenten berechnet sich die Gesamtzahl n_{normal} der zur Ausführung des Agenten notwendigen Nachrichten bei einer Anzahl n_S von Schritten mittels

$$n_{normal} = 2 \cdot n_S \qquad (4\text{-}1)$$

Kommunikationsaufwand beim Basisprotokoll.

Wie bei der ungesicherten Ausführung des Agenten muß auch beim Basisprotokoll der Agent nur zum jeweils nächsten Knoten (bzw. auf den ersten Knoten für den ersten Schritt) mittels einer Migrationsnachricht übertragen werden. Diese Migrationsnachricht enthält zusätzlich zu Agentencode und Datenzustand des Agenten einen Transaktionskontext. Da der Agent auf dem Zielknoten innerhalb des Schritt-Transaktionskontexts (bzw. der Transaktion, in der der Agent auf den ersten Knoten transportiert wird) in die Eingangswarteschlange geschrieben wird, kommen in diesem Fall zusätzlich noch die 4 (kurzen) Nachrichten des 2PC-Protokolles zwischen Ausgangsknoten und Zielknoten hinzu. Wichtig in diesem Zusammenhang ist die Feststellung, daß die Bestätigungsnachricht, welche die Ankunft des Agenten auf dem Zielknoten bestätigt, nicht weggelassen werden kann, da ein Verlust der Nachricht, die den Agenten transportiert, nicht bemerkt werden würde. Treten keine Fehler auf, so werden also für die Ausführung eines Schrittes mittels des Basisprotokolles eine Migrationsnachricht, eine Bestätigungsnachricht und 4 Nachrichten des 2PC-Protokolles benötigt. Geht die Nachricht mit dem Agent oder die Bestätigungsnachricht verloren, sind zusätzliche Nachrichten notwendig um sicherzustellen, daß der Agent auf dem Zielknoten in die Eingangswarteschlange geschrieben wurde. Schlägt die Transaktion fehl, muß der Schritt inklusive des Transports des Agenten auf den nächsten Knoten wiederholt werden.

Bei fehlerfreier Ausführung des Agenten berechnet sich die Gesamtzahl n_{Basis} der zur Ausführung des Agenten notwendigen Nachrichten bei einer Anzahl n_S von Schritten mittels

$$n_{Basis} = 6 \cdot n_S \qquad (4\text{-}2)$$

Bietet das verwendete Transaktionssystem die Möglichkeit, daß mit der den Agenten enthaltenden Nachricht gleich das *rm_prepare* vom Transaktionsmanager an die Eingangswarteschlange des Nachfolgeknoten verschickt werden kann und mit der Bestätigungsnachricht gleich das *rm_yes* bzw. *rm_no* der Eingangswarteschlange an den Transaktionsmanager zurückübermittelt werden kann, so verringert sich die im letzten Absatz ermittelte Anzahl von 2PC-Protokoll-Nachrichten auf 2 Nachrichten. In diesem Falle reduziert sich die Gesamtzahl der zur Ausführung notwendigen Nachrichten auf

$$n_{Basis,\,opt} = 4 \cdot n_S \qquad (4\text{-}3)$$

Bei fehlerfreier Ausführung führt das Basisprotokoll im Vergleich zur ungesicherten Ausführung des Agenten also durch das 2PC-Protokoll einen Mehraufwand von 4 (kurzen) Nachrichten pro Schritt ein (die Vergrößerung der Migrationsnachricht durch den Transaktionskontext ist minimal und kann vernachlässigt werden). Kann die im letzten Absatz beschriebene Optimierung eingesetzt werden, verringert sich der Mehraufwand auf 2 Nachrichten pro Schritt.

Kommunikationsaufwand des blockierungsfreien Protokolles.

Die blockierungsfreie Variante des Protokolles führt selbst für den Fall, daß während der Ausführung des Agenten keine Fehler auftreten, einigen Mehraufwand ein. Abbildung 4-19 zeigt die zur Durchführung einer Stufe S_i notwendigen Interaktionen für den Fall, daß keine Fehler auftreten.

Während der Agent ausgeführt wird, werden vom Arbeiter regelmäßig (kurze) *I_AM_ALIVE*(..)-Nachrichten an die anderen Knoten der Stufe verschickt. Die Anzahl der hierbei verschickten Nachrichten hängt von der Anzahl $n_{K,i}$ der Knoten der Stufe i und vor allem vom Verhältnis der Dauer der Schritt-Transaktion $t_{ST,i}$ zu der Periodendauer t_p zwischen dem Versenden der *I_AM_ALIVE*(..)-Nachrichten ab (vgl. Abschnitt 4.4.3.1). Die Anzahl $n_{B,i}$ der in der Stufe S_i für das Beobachtungsprotokoll erzeugten *I_AM_ALIVE*(..)-Nachrichten berechnet sich zu

$$n_{B,i} = (n_{K,i} - 1) \cdot \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor \qquad (4\text{-}4)$$

Ist in der Stufe S_i die Dauer $t_{ST,i}$ der Schritt-Transaktion kleiner als die Periodendauer t_p zwischen zwei *I_AM_ALIVE*(..) -Nachrichten, so wird $\lfloor t_{ST,i}/t_p \rfloor$ zu Null, d.h. es müssen in diesem Fall gar keine I_AM-ALIVE-Nachrichten verschickt werden.

Nachdem die Ausführung des Agenten beendet ist, werden der Code des Agenten, der aktuelle Datenzustand des Agenten, der Stufen-Record der nachfolgenden Stufe S_{i+1} und der Transaktionskontext in einer Migrationsnachricht auf die Knoten von S_{i+1} transportiert und der Empfang

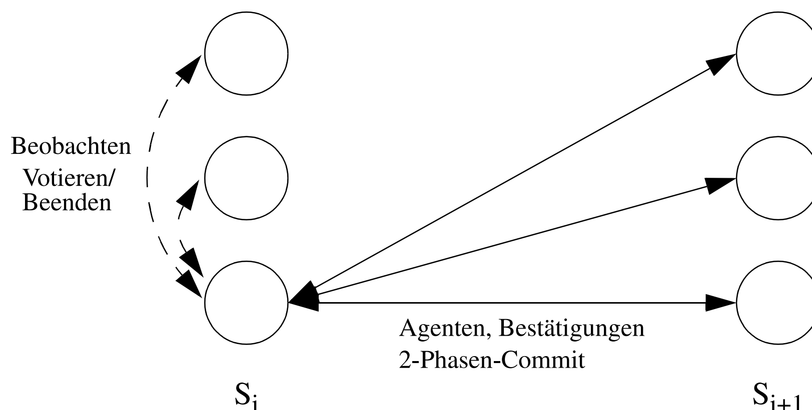


Abbildung 4-19. Interaktionen während Ausführung einer Schritt-Transaktion

durch die Knoten der nachfolgenden Stufe bestätigt. Hierdurch entstehen also $n_{K,i+1}$ (lange) Migrationsnachrichten und $n_{K,i+1}$ (kurze) Bestätigungsnachrichten ($n_{K,i+1}$ = Anzahl Knoten der Stufe S_{i+1}).

Beim Abschluß der Schritt-Transaktion entstehen Nachrichten durch das 2PC-Protokoll und durch das Votierprotokoll. Für das 2PC-Protokoll werden pro Knoten der nachfolgenden Stufe S_{i+1} insgesamt 4 (kurze) Nachrichten verschickt. Besteht die Möglichkeit der oben erwähnten Optimierung, so reduziert sich dies auf 2 Nachrichten pro Knoten. Durch das 2-Phasen-Commit entstehen also in der Stufe S_i $4*n_{K,i+1}$ Nachrichten bzw. $2*n_{K,i+1}$ Nachrichten bei Anwendung der Optimierung.

Für das Votierprotokoll tauscht der Orchestrator des Arbeiters mit den anderen Knoten der Stufe jeweils 4 (kurze) Nachrichten aus. Hierbei entfallen 2 Nachrichten auf das Einsammeln der Voten und 2 Nachrichten auf die Benachrichtigung der anderen Knoten über das Ende der Stufe. Durch das Votierprotokoll ergibt sich also in der Stufe S_i ein Aufwand von $4*(n_{K,i}-1)$ Nachrichten.

Bis auf die letzte Stufe ergibt sich insgesamt für die Anzahl n_i der Nachrichten pro Stufe S_i:

$$\begin{aligned} n_i &= n_{B,i} + (n_{K,i+1} + n_{K,i+1}) + 4 \cdot n_{K,i+1} + 4 \cdot (n_{K,i} - 1) \\ &= 6 \cdot n_{K,i+1} + \left(4 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_{K,i} - 1) \end{aligned} \tag{4-5}$$

beziehungsweise bei Anwendung der oben genannten Optimierung im 2-Phasen-Commit:

$$n_i = 4 \cdot n_{K,i+1} + \left(4 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_{K,i} - 1) \tag{4-6}$$

Unter der Annahme, daß für die Ausführung eines Agenten in allen Stufen dieselbe Anzahl n_K von Knoten verwendet wird, vereinfacht sich Gleichung (4-6) auf

$$n_i = \left(10 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_K - 1) + 6 \tag{4-7}$$

beziehungsweise bei Anwendung der Optimierung im 2-Phasen-Commit:

$$n_i = \left(8 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_K - 1) + 4 \tag{4-8}$$

Unter der Annahme $t_p > t_{ST,i}$ ergibt sich für Stufengrößen von 3 bzw. 5 Knoten pro Stufe eine Nachrichtenanzahl von 26 bzw. 46 Nachrichten im Normalfall und 20 bzw. 36 Nachrichten bei Anwendung der Optimierung im 2-Phasen-Commit.

In der Schritt-Transaktion der letzten Stufe entfällt der Transport des Agenten auf die Knoten der nachfolgenden Stufe. Jedoch muß vor der Durchführung der Schritt-Transaktion in der ersten Stufe der Agent auch innerhalb einer Transaktion auf die Knoten der ersten Stufe transportiert werden, wobei hierzu (bei konstanter Knotenanzahl pro Stufe) derselbe Aufwand notwendig ist wie der, der in der letzten Stufe entfällt.

Bei fehlerfreier Ausführung des Agenten berechnet sich die Gesamtzahl n_{FT} der zur Ausführung des Agenten notwendigen Nachrichten bei einer Anzahl n_S von Stufen mittels

$$n_{FT} = 6 \cdot n_{K,1} + \sum_{i=1}^{n_S - 1} \left(6 \cdot n_{K,i+1} + \left(4 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_{K,i} - 1)\right) + \left(4 + \left\lfloor \frac{t_{ST,n_S}}{t_p} \right\rfloor\right) \cdot (n_{K,n_S} - 1) \qquad (4\text{-}9)$$

beziehungsweise bei Anwendung der oben genannten Optimierung im 2-Phasen-Commit

$$n_{FT,\,opt} = 4 \cdot n_{K,1} + \sum_{i=1}^{n_S - 1} \left(4 \cdot n_{K,i+1} + \left(4 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_{K,i} - 1)\right) + \left(4 + \left\lfloor \frac{t_{ST,n_S}}{t_p} \right\rfloor\right) \cdot (n_{K,n_S} - 1) \qquad (4\text{-}10)$$

Unter der Annahme, daß für die Ausführung eines Agenten in allen Stufen dieselbe Anzahl n_K von Knoten verwendet wird, vereinfacht sich dies auf

$$n_{FT} = (10 \cdot n_K - 4) \cdot n_s + (n_K - 1) \cdot \sum_{i=1}^{n_S} \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor \qquad (4\text{-}11)$$

beziehungsweise bei Anwendung der Optimierung im 2-Phasen-Commit

$$n_{FT,\,opt} = (8 \cdot n_K - 4) \cdot n_s + (n_K - 1) \cdot \sum_{i=1}^{n_S} \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor \qquad (4\text{-}12)$$

Treten während der Ausführung des Agenten Fehler auf, versucht das Protokoll, diese zu eliminieren. Dabei werden weitere Nachrichten erzeugt. Sind während des Votierprotokolles Knoten der Stufe nicht erreichbar, werden Nachrichten des Votierprotokolles an diese Knoten mehrfach

verschickt. Fallen Knoten der nächsten Stufe während der Phase des Transports des Agenten zu diesen Knoten aus, so muß entweder die Zusammensetzung der Stufe neu ermittelt werden und diese Informationen neu mittels einiger weniger Nachrichten an die Knoten der nächsten Stufe verteilt werden oder, falls nicht eindeutig festgestellt werden kann, ob der Agent bei einem Knoten ankam oder nicht, die gesamte Schritt-Transaktion abgebrochen und wiederholt werden. Fällt während der Ausführung des Agenten der Arbeiter aus bzw. ist der Arbeiter nicht mehr erreichbar, so wird das Auswahlprotokoll gestartet. Ist der nicht mehr erreichbare Arbeiter der Knoten mit der höchsten Priorität, so starten im ungünstigsten Falle bei n_K Knoten in der Stufe n_K-1 Knoten gleichzeitig das Auswahlprotokoll. Hierbei verschickt jeder Knoten *ARE_YOU-THERE*(..)-Nachrichten an die Knoten mit höherer Priorität. Da aber im ungünstigsten Fall dies alle Knoten zur selben Zeit tun, bekommt jeder Knoten von allen Knoten mit geringerer Priorität eine *ARE_YOU_THERE*(..)-Nachricht, auf die er mit einer *I_AM_THERE*(..)-Nachricht antworten muß. Es verschickt also jeder der n_K-1Knoten Nachrichten an alle anderen Knoten der Stufe außer an sich selbst – eine *ARE_YOU_THERE*(..)-Nachricht an die Knoten mit höherer Priorität und eine *I_AM_THERE*(..)-Nachricht an die Knoten mit niedrigerer Priorität – insgesamt also n_K-1 Nachrichten pro Knoten. Inklusive der n_K-1 *I_AM_SELECTED*(..)-Nachrichten des Gewinners ergeben sich also im ungünstigsten Fall n_K*(n_K-1) Nachrichten.

Vergleicht man den Kommunikationsaufwand für einen Schritt des Basisprotokolls mit dem blockierungsfreien Protokoll, so ergibt sich bei fehlerfreier Ausführung des/der *i*-ten Schrittes/ Stufe für das blockierungsfreie Protokoll mit konstanter Anzahl n_K von Knoten pro Stufe ein Mehraufwand von

$$n_{overhead,\,i} = \left(10 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_K - 1) \qquad (4\text{-}13)$$

beziehungsweise bei Anwendung der Optimierung im 2-Phasen-Commit

$$n_{overhead,\,i,\,opt} = \left(8 + \left\lfloor \frac{t_{ST,i}}{t_p} \right\rfloor\right) \cdot (n_K - 1) \qquad (4\text{-}14)$$

Der Mehraufwand setzt sich zusammen aus $\lfloor t_{ST,i}/t_p \rfloor \cdot (n_K - 1)$ (kurzen) *I_AM_ALIVE*(..)-Nachrichten, $4 \cdot (n_K - 1)$ (kurzen) Nachrichten des Votierprotokolles, $n_K - 1$ Migrationsnachrichten, $n_K - 1$ (kurzen) Bestätigungsnachrichten und $4 \cdot (n_K - 1)$ (kurzen) Nachrichten des 2PC-Protokolles (bzw. $2 \cdot (n_K - 1)$ Nachrichten bei der optimierten Version). Auch hier kann man, zumindest bei "größeren" Agenten, die durch den zusätzlich transportierten Stufen-Record verursachte Zunahme der Größe der Migrationsnachricht vernachlässigen, da der Stufen-Record für sinnvolle Stufengrößen von 3-7 Knoten im Vergleich zu Code und Datenzustand des Agenten vergleichsweise klein ist.

4.5.2 Möglichkeiten zur Reduktion des Kommunikationsaufwandes

Eine sehr offensichtliche Möglichkeit zur Reduktion des Kommunikationsaufwandes des blockierungsfreien Protokolles ergibt sich direkt aus Gleichung (4-13) bzw. Gleichung (4-14): je weniger Knoten eine Stufe besitzt, desto geringer ist der Nachrichtenmehraufwand. Für eine Stufe mit nur einem Knoten entspricht das blockierungsfreie Protokoll weitgehend dem Basisprotokoll und besitzt in diesem Fall auch keinen Nachrichtenmehraufwand im Vergleich zum Basisprotokoll. Wie sich allerdings in Abschnitt 4.6 zeigen wird, bedeutet eine Verringerung der Anzahl von Knoten pro Stufe (präziser: die Verringerung der Knotenanzahl pro Stufe um 2) eine Erhöhung der Wahrscheinlichkeit, daß der Agent blockiert wird – und widerspricht somit der Zielsetzung des Protokolles. Ebenfalls offensichtlich ergibt sich aus den genannten Gleichungen, daß sich die Anzahl der Nachrichten durch die Wahl einer langen Periodendauer t_p zwischen zwei *I_AM_ALIVE*(..)-Nachrichten in gewissem Umfang reduzieren läßt – wodurch allerdings die Zeit erhöht wird, bis auf einen Fehler reagiert wird. Der Anwender hat jedoch die Möglichkeit, durch Adaption der beiden Parameter “Anzahl der Knoten pro Stufe” und “Periodendauer t_p” den gewünschten Grad der Fehlertoleranz für seine Anwendung anzupassen: Benötigt eine Anwendung nur einen geringen Grad an Fehlertoleranz, so wird der Anwender keinen zu großen Mehraufwand akzeptieren und deshalb sollte in diesem Fall vom Agent bzw. der Anwendung eine kleine Anzahl Knoten pro Stufe (eventuell nur ein Knoten) und eine lange Periodendauer t_p gewählt werden. Ist dem Benutzer jedoch für die Ausführung einer Anwendung ein hoher Grad an Fehlertoleranz wichtig, so wird er dafür auch einen größeren Mehraufwand akzeptieren (den er eventuell zu bezahlen hat), wodurch der Agent bzw. die Anwendung die Möglichkeit besitzt, eine größere Anzahl an Knoten pro Stufe und eine kurze Periodendauer t_p zu wählen.

Steht die Anzahl der Knoten pro Stufe und die Periodendauer t_p fest, kann beim Beobachtungsprotokoll und beim Votierprotokoll der Aufwand nicht weiter reduziert werden. Es besteht jedoch die Möglichkeit, durch entsprechende Auswahl der Knoten einer Stufe den für die Migration des Agenten notwendigen Kommunikationsaufwand zu reduzieren. Ist beispielsweise wie in Abbildung 4-20 ein Knoten K_2 Mitglied zweier aufeinanderfolgender Stufen S_i und S_{i+1}, so kann offensichtlich bei der Versendung des Agenten auf die Knoten der Stufe S_{i+1} bei der zu K_2 versendeten Migrationsnachricht auf den Code des Agenten verzichtet werden, da dieser ja noch aus Stufe S_i dort vorhanden ist. Hierdurch reduziert sich allerdings nur die Menge der zu versendenden Daten, nicht jedoch die Anzahl der zu versendenden Nachrichten.

Eine Reduzierung der zu versendenden Nachrichten kann erreicht werden, wenn der Arbeiter der Stufe S_i auch Mitglied der Stufe S_{i+1} ist. Besitzt die Stufe S_{i+1} insgesamt $n_{K,i+1}$ Knoten, so muß der Agent in diesem Fall nur auf $n_{K,i+1}$-1 Knoten übertragen werden, wodurch eine Migrationsnachricht, deren Bestätigung und 4 (bzw. 2) Nachrichten des 2PC-Protokolles wegfallen. Abbildung 4-21 zeigt ein Szenario, in dem der Knoten K_1 in Stufen S_i und S_{i+1} teilnimmt und in der Stufe S_i den Agenten ausführt. Um den Agent für Stufe S_{i+1} in die Eingangswarteschlange

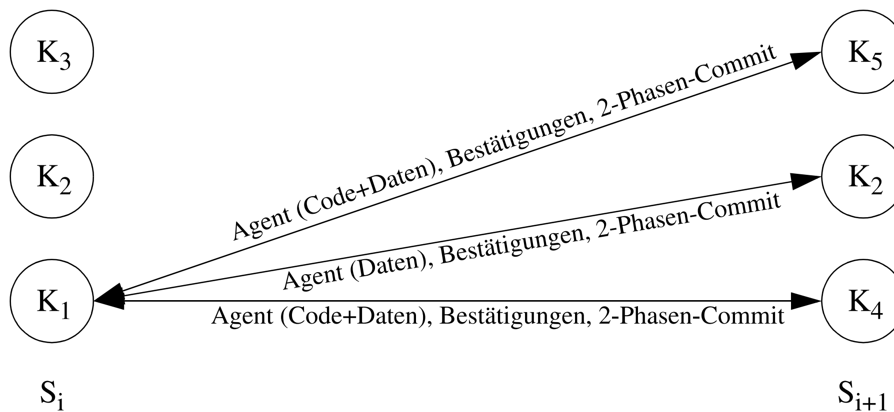


Abbildung 4-20. Einsparung von Code-Transport durch Teilnahme desselben Knotens in zwei aufeinanderfolgenden Stufen

von K_1 zu schreiben müssen in diesem Fall keine Nachrichten über das Netzwerk verschickt werden – die Kommunikation geschieht nur lokal. Der Kommunikationsmehraufwand im Vergleich zum Basisprotokoll wird jedoch nur bedingt verringert. Der genannte Fall, daß der Arbeiter der Stufe S_i auch Mitglied der Stufe S_{i+1} ist, kann nämlich in zwei Situationen auftreten, von denen nur eine eine tatsächliche Reduktion des Kommunikationsmehraufwandes darstellt. In der ersten Situation verfügt die Stufe S_{i+1} nicht über genügend reguläre Knoten. In diesem Falle wird der Arbeiter der Stufe S_i als Ausnahmebehandlungsknoten in die Stufe S_{i+1} aufgenommen, wodurch eine tatsächliche Reduktion des Kommunikationsmehraufwandes erreicht wird. In der zweiten Situation ist der Arbeiter der Stufe S_i regulärer Knoten der Stufe S_{i+1}. Dieser Fall – der Agent führt zwei Schritte hintereinander auf demselben Knoten aus – wurde bei

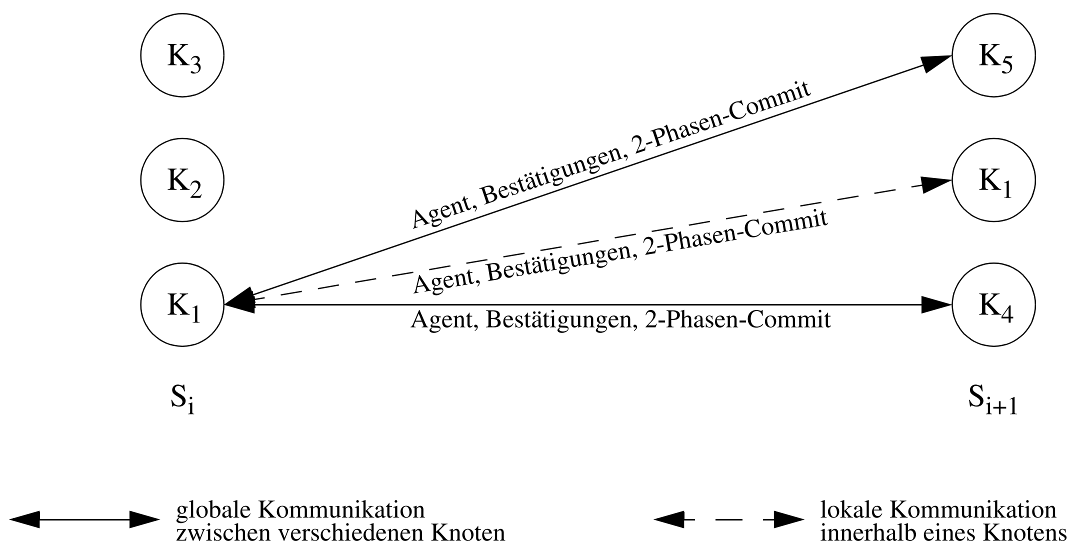


Abbildung 4-21. Einsparung von Nachrichten durch Teilnahme des Arbeiters einer Stufe in der darauffolgenden Stufe

den obigen Betrachtungen explizit ausgeschlossen. Wird der Agent mittels des Basisprotokolles ausgeführt, werden in diesem Falle natürlich auch beide Schritte hintereinander auf demselben Knoten ausgeführt. Insofern ergibt sich in diesem Fall keine Reduktion des durch das blockierungsfreie Protokoll eingeführten Kommunikationsmehraufwandes.

Weitere Reduktionen des Kommunikationsaufwandes sind nur durch Abänderung des Protokolles zu erreichen. So könnte beispielsweise das Votierprotokoll dahingehend geändert werden, daß in einem ersten Schritt nur von so vielen Knoten Voten angefordert werden, daß, unter der Voraussetzung, daß diese alle mit *YES* antworten, schon eine Mehrheit an Voten erreicht würde. Nur wenn nicht genug *YES*-Voten ankommen (z.B. weil einige dieser Knoten ausgefallen sind), müßten von weiteren Knoten der Stufe Voten angefordert werden. Treten keine Fehler auf, würde hierdurch für die knappe Hälfte der Knoten einer Stufe die erste Stufe des Votierprotokolles entfallen. Bei n_K Knoten in einer Stufe würde dies einer Einsparung von $\lfloor (n_K - 1)/2 \rfloor$ (kurzer) Votiernachrichten pro Stufe entsprechen. Allerdings verzögert sich hierdurch jedoch auch der Abschluß der Transaktion, falls nicht alle zur Abgabe einer Stimme aufgeforderten Knoten ihre Stimme abgeben. Wie schon in Abschnitt 4.4.3.2 erwähnt ist es weiterhin möglich, durch Wahl eines anderen Algorithmus den Nachrichtenaufwand für das Auswahlprotokoll zu reduzieren – allerdings auch auf Kosten der Zeit.

4.5.3 Algorithmus zur Stufenkonstruktion

Der in diesem Abschnitt beschriebene Algorithmus hat zum Ziel, die Knoten einer Stufe derart auszuwählen, daß der Kommunikationsmehraufwand so weit wie möglich reduziert wird. Dabei wird auf die im letzten Abschnitt erlangten Erkenntnisse zurückgegriffen. Die grundlegende Idee des Algorithmus ist, so viele reguläre Knoten wie möglich zur Konstruktion der Stufe zu verwenden und dabei – sofern Freiheiten in der Wahl der Knoten bestehen – sicherzustellen, daß aufeinanderfolgende Stufen möglichst viele Knoten gemeinsam haben. Als Eingabeparameter erhält der Algorithmus die Anzahl n der Knoten, aus denen die nächste Stufe S_{i+1} bestehen soll, die Reiseroute des Agenten und die Menge der Knoten der aktuellen Stufe S_i. Der Algorithmus wird auf dem Arbeiter W_i der Stufe S_i ausgeführt und bestimmt die Knoten der Stufe S_{i+1} nebst deren Priorität. Zur Vereinfachung bezeichnen im folgenden S_i und S_{i+1} jeweils auch die Menge der Knoten der Stufen S_i und S_{i+1} (aus Kontext ersichtlich).

In einem ersten Schritt **(1)**[1] bestimmt der Algorithmus mittels der Reiseroute die Menge $Next_i$. $Next_i$ enthält jene Knoten, auf denen der Agent laut Reiseroute als nächstes einen Schritt ausführen kann. Die hierzu notwendige Funktionalität hat die Reiseroute zur Verfügung zu stellen. Der Inhalt dieser Menge ist im allgemeinen davon abhängig, welcher Knoten der Stufe (hier der Arbeiter W_i) den Agent (und damit den Algorithmus zur Stufenkonstruktion der folgenden Stufe) ausführt. Sowohl die Menge $Next_i$ als auch S_i kann Knoten enthalten, von denen W_i weiß,

1. Die Zahlen in Klammern beziehen sich auf die Numerierung in Algorithmus 4-9

```
buildStage(stageSize, S_i, itinerary){
    next = itinerary.getNextPossibleNodes() (1)
    remove non-available nodes from S_i
      and next                              (2)
    S_i+1 = {}
    W = current worker
    if (|next| == stageSize){ (3)
        S_i+1 = next
    } else if (|next| > stageSize){ (4)
        S_i+1 = S_i ∩ next
        if (S_i+1 > stageSize){ (5)
            determine priority of nodes in S_i+1
            remove |S_i+1|-stageSize nodes with
              lowest priorities from S_i+1
            if (W ∈ next AND W ∉ S_i+1){ (6)
                remove node with lowest priority
                  from S_i+1
                add W to S_i+1
            }
        } else if (S_i+1 < stageSize){ (7)
            determine priority of nodes in next\S_i+1
            add stageSize-|S_i+1| nodes with
                highest priorities from next\S_i+1 to
                S_i+1
    } }
    else { // |next| < stageSize (8)
        S_i+1 = next (9)
        if (W ∉ S_i+1){
            add W to S_i+1 (10)
        }
        m = stageSize - |S_i+1|
        if (m≤|S_i\S_i+1|){ (11)
            determine priorities of nodes in S_i\S_i+1
            add m nodes from S_i\S_i+1 with highest
              priorities to S_i+1
        }else{ (12)
            add S_i\S_i+1 to S_i+1
            add stageSize-|S_i+1| arbitrary available
              nodes to S_i+1
        }
        determine priorities of nodes in S_i+1
}
```

Algorithmus 4-9. Stufenkonstruktion

daß sie momentan nicht verfügbar sind – z.B. weil die Migration eines anderen Agenten dorthin gerade fehlschlug. Ist dies der Fall, so werden diese Knoten aus $Next_i$ und S_i vor der Ausführung der weiteren Schritte des Algorithmus entfernt **(2)**.

Im zweiten Schritt werden dann die Knoten in S_{i+1} bestimmt. Hierbei kann man drei verschiedene Fälle unterscheiden. In den ersten beiden Fällen enthält $Next_i$ ausreichend Knoten, sodaß alle Knoten in S_{i+1} reguläre Knoten sind. Während im ersten Fall die Anzahl der Knoten in $Next_i$ genau der Anzahl der Knoten in der nächsten Stufe entspricht und somit die Knoten der nächsten Stufe feststehen, muß im zweiten Fall (mehr Knoten in $Next_i$ als für nächste Stufe notwendig) aus $Next_i$ eine Untermenge bestimmt werden. Dies geschieht, indem man versucht, die Schnittmenge zwischen aktueller Stufe und der nächsten möglichst groß zu machen (kein Code-Transport auf die Knoten der aktuellen Stufe notwendig) und vor allem möglichst den aktuellen Arbeiter mit einzubeziehen (kein Transport des Agenten notwendig). Im dritten Fall enthält $Next_i$ zu wenig Knoten für die nächste Stufe und man muß Ausnahmebehandlungsknoten suchen. Hier werden aus den oben genannten Gründen bevorzugt der momentane Arbeiter und die Knoten der aktuellen Stufe verwendet, bevor auf andere Knoten zurückgegriffen wird. Die folgenden Absätze beschreiben diese drei Fälle detailliert:

Fall 1: (3) Im einfachsten Fall ist die Kardinalität von $Next_i$ ($|Next_i|$) gleich der gewünschten Anzahl von Knoten in S_{i+1}. In diesem Fall können die Knoten in $Next_i$ direkt als Knoten in S_{i+1}

verwendet werden, d.h. S_{i+1}=$Next_i$. Somit enthält S_{i+1} nur reguläre Knoten. Die Zuordnung von Prioritäten zu diesen Knoten wird weiter unten beschrieben.

Fall 2: (4) Ist die Kardinalität von $Next_i$ größer als die gewünschte Anzahl n von Knoten in S_{i+1}, so besteht auch hier die resultierende Stufe nur aus regulären Knoten. Um die aus $Next_i$ für die Bildung von S_{i+1} zu verwendenden Knoten zu bestimmen, wird zuerst $S_{i+1}=S_i \cap Next_i$ berechnet. Hierdurch wird sichergestellt, daß S_i und S_{i+1} möglichst viel gemeinsame Knoten enthalten, wodurch der Transport vom Agentencode auf diese Knoten eingespart werden kann. Falls S_{i+1} danach noch zu viele Knoten enthält **(5)**, d.h. falls $|S_{i+1}|>n$, werden die Prioritäten der Knoten bestimmt (siehe weiter unten) und die n Knoten mit der höchsten Priorität verbleiben dann in S_{i+1}. Ist jedoch $|S_{i+1}|<n$ **(7)**, müssen aus den verbleibenden Knoten $Next_i \backslash S_i$ noch die n-$|S_{i+1}|$ Knoten mit der höchsten Priorität (siehe unten) zu S_{i+1} hinzugefügt werden. Falls die Menge $Next_i$ auch den Knoten W_i enthält, ist es wünschenswert, daß W_i in S_{i+1} enthalten ist, da dadurch Nachrichten eingespart werden. Es ist also zu beachten, daß W_i ungeachtet der errechneten Priorität auf jeden Fall S_{i+1} angehören muß. Für den Fall $|S_i \cap Next_i|>n$ bedeutet dies, daß nur die n-1 Knoten mit der höchsten Priorität und zusätzlich W_i in S_{i+1} aufgenommen werden, falls W_i nicht in den n Knoten mit der höchsten Priorität enthalten ist **(6)**. Die letztendliche Zuordnung der Prioritäten zu den Knoten wird weiter unten beschrieben.

Fall 3: (8) Ist die Kardinalität von $Next_i$ kleiner als n (d.h. $|Next_i|<n$), dann enthält die Stufe S_{i+1} neben den (regulären) Knoten aus $Next_i$ **(9)** noch zusätzlich $m=n-|Next_i|$ Ausnahmebehandlungsknoten. Gute Kandidaten für die Auswahl der Ausnahmebehandlungsknoten sind – soweit sie nicht bereits in $Next_i$ enthalten sind – die Knoten der Stufe S_i im allgemeinen und der Arbeiter der Stufe S_i im besonderen, da durch ihre Auswahl wie schon weiter oben dargelegt sowohl die Anzahl der notwendigen Agentencode-Transporte (wenn Knoten aus Stufe S_i auch in Stufe S_{i+1} sind) als auch die Anzahl der notwendigen Nachrichten (wenn der Arbeiter aus Stufe S_i auch in Stufe S_{i+1} ist) reduziert werden kann.

Ist $m \leq |S_i \backslash Next_i|$ **(11)**, dann können die Ausnahmebehandlungsknoten komplett aus $S_i \backslash Next_i$ entnommen werden, wobei der Arbeiter W_i (falls nicht schon in $Next_i$ enthalten) auf jeden Fall einer jener Ausnahmebehandlungsknoten wird **(10)**. Kann den Knoten aus $S_i \backslash Next_i$ dabei nach einer der weiter unten beschriebenen Strategien eine Priorität zugeordnet werden, so nimmt man als Ausnahmebehandlungsknoten neben W_i (falls dieser nicht in $Next_i$ schon enthalten ist) m-1 Knoten aus $S_i \backslash Next_i \backslash W_i$. Ist $m>|S_i \backslash Next_i|$ **(12)**, werden alle Knoten aus $S_i \backslash Next_i$ als Ausnahmebehandlungsknoten verwendet. Für die Auswahl der jetzt noch fehlenden m-$|S_i \backslash Next_i|$ Ausnahmebehandlungsknoten können entweder beliebige Knoten oder Knoten, die als zukünftige (potentielle) Ziele des Agenten in der Reiseroute stehen, verwendet werden. Insgesamt enthält die Stufe S_{i+1} im Falle $|Next_i|<m$ also die Knoten aus $Next_i$ als reguläre Knoten und die wie gerade beschrieben ausgewählten Ausnahmebehandlungsknoten. Die letztendliche Zuordnung der Prioritäten zu den Knoten wird weiter unten beschrieben. Es ist in diesem Falle jedoch zu beachten, daß die Ausnahmebehandlungsknoten eine geringere Priorität als die regulären Knoten erhalten.

Das weiter oben beschriebene Vorgehen für den Fall 2 ($Next_i$>n) reduziert zwar möglicherweise die Anzahl der Nachrichten und die Menge der zu übertragenden Daten, es sorgt aber dafür, daß die in einer Reiseroute möglicherweise definierten Prioritäten nicht eingehalten werden. Um die Prioritäten der Reiseroute möglichst einzuhalten, müßten für diesen Fall die n Knoten mit der höchsten Priorität aus $Next_i$ für S_{i+1} genommen werden – ohne Rücksicht auf S_i. Sollte die Einhaltung der Prioritäten wichtiger sein als die Reduktion des Kommunikationsaufwandes, so darf also für den Fall $Next_i$>n nicht nach der oben beschriebenen Weise vorgegangen werden, sondern es müssen für die Bildung der Stufe die n Knoten mit der höchsten Priorität aus $Next_i$ verwendet werden.

In einem letzten Schritt werden die Prioritäten für die Knoten der Stufe S_{i+1} endgültig festgelegt. Hierbei wird prinzipiell so vorgegangen, daß zuerst die Prioritäten der Ausnahmebehandlungsknoten festgelegt werden und danach die Prioritäten der regulären Knoten so festgelegt werden, daß die Priorität der regulären Knoten höher ist als die Priorität der Ausnahmebehandlungsknoten. Es muß hierbei auf jeden Fall darauf geachtet werden, daß jeder Knoten der Stufe eine eindeutige Priorität besitzt (d.h. es darf keine 2 Knoten mit derselben Priorität in einer Stufe geben). Bei der Festlegung von Prioritäten sind im Kontext dieses Algorithmus mehrere Strategien möglich. Legt die Reiseroute Prioritäten zwischen den regulären Knoten der Stufe fest, so muß sich dies in den Stufenprioritäten der regulären Knoten widerspiegeln, d.h. sind Knoten x und Knoten y reguläre Knoten der Stufe und spezifiziert die Reiseroute, daß x eine höhere Priorität als y besitzt, so muß x in der Stufe eine höhere Priorität haben als y. Spezifiziert die Reiseroute zwischen zwei Knoten keine Priorität bzw. für zwei Knoten dieselbe Priorität, so kann man die Priorität der beiden Knoten mit einer der folgenden Strategien festlegen.

Eine mögliche Strategie zur Festlegung der Prioritäten der Ausnahmebehandlungsknoten bzw. der Prioritäten von regulären Knoten, zwischen denen die Reiseroute keine Prioritäten festlegt, ist die Verwendung zufälliger Prioritäten. Eine effizientere Strategie ist möglich, wenn Wissen über die Zuverlässigkeit von Knoten verfügbar ist. In diesem Falle bekommen Knoten mit höherer Verfügbarkeit eine höhere Priorität (die Priorität zwischen Knoten mit gleicher Zuverlässigkeit kann dann wieder zufällig festgelegt werden). Bei Verwendung dieser Heuristik werden Knoten mit höherer Zuverlässigkeit für die Ausführung des Agenten bevorzugt.

Die durch den Algorithmus erreichbare Reduktion des Kommunikationsmehraufwandes zeigt Tabelle 4-1. Bei einer Stufengröße von n=3 Knoten reduziert sich beispielsweise bei der Verwendung von nur einem Knoten der aktuellen Stufe als Knoten der nächsten Stufe die Anzahl der notwendigen Agentencode-Transporte um ein Drittel.

Die in Abbildung 4-22 abgebildete (einfache) Beispielreiseroute, welche mit einer konstanten Stufengröße von n=3 Knoten abgearbeitet werden soll, zeigt einen optimalen Fall für den Algorithmus zur Stufenkonstruktion. Die erste Stufe enthält die Knoten N_1 bis N_3 als reguläre Knoten sortiert nach ihrer Priorität. Zum Start des Agenten wird der Stufen-Record, der Code des Agenten und eventuell Datenzustand mittels 3 Migrationsnachrichten und 3 Bestätigungen auf die

	Code-Transporte	Migrationsnachrichten + Bestätigungen	Nachrichten für 2PC
$S_i \cap S_{i+1} = \varnothing$	n	$2n$	$4n$
$S_i \cap S_{i+1} \neq \varnothing \wedge W_i \notin S_{i+1}$	$n-\lvert S_i \cap S_{i+1}\rvert$	$2n$	$4n$
$S_i \cap S_{i+1} \neq \varnothing \wedge W_i \in S_{i+1}$	$n-\lvert S_i \cap S_{i+1}\rvert$	$2(n-1)$	$4(n-1)$

Tabelle 4-1. Reduktion des Kommunikationsaufwandes durch den Algorithmus zur Stufenkonstruktion

Knoten der ersten Stufe transportiert. Für den Abschluß der Starttransaktion werden 4*3 Nachrichten für das 2PC-Protokoll benötigt. In der zweiten Stufe wird der Arbeiter der ersten Stufe (z.B. N_1) als Ausnahmebehandlungsknoten verwendet, N_2 und N_3 sind reguläre Knoten. In diesem Fall sind nur zwei Migrationsnachrichten (auf N_2 und N_3) und deren Bestätigung notwendig, wobei hier jedoch kein Transport des Agentencodes mehr notwendig ist. Auch die zur Beendigung der Schritt-Transaktion der ersten Stufe notwendigen Nachrichten des 2PC-Protokolles reduzieren sich auf 4*2. In der dritten Stufe werden N_1 und der Arbeiter der zweiten Stufe (z.B. N_2) als Ausnahmebehandlungsknoten verwendet, N_3 ist der (einzige) reguläre Knoten. Auch hier fallen wieder nur zwei Migrationsnachrichten (auf N_1 und N_3) und deren Bestätigung sowie 4*2 Nachrichten des 2PC-Protokolles an. Insgesamt ergibt sich für Migration und 2PC-Protokoll also ein Aufwand von 7 Migrationsnachrichten, davon nur 3 Nachrichten mit Agentencode, 7 Bestätigungsnachrichten und 4*7=28 Nachrichten für das 2PC-Protokoll. Im Vergleich zum Basisprotokoll ist das ein Mehraufwand von 4 Migrationsnachrichten, deren 4 Bestätigungen und 16 Nachrichten für das 2PC-Protokoll. Da die 4 zusätzlichen Migrationsnachrichten jeweils keinen Agentencode enthalten, entsteht kein überflüssiger Code-Transport. Ohne die Flexibilität in der Reiseroute und ohne die Optimierung des Algorithmus zur Stufenkonstruktion würde sich (bei Auswahl von jeweils 2 beliebigen Ausnahmebehandlungsknoten pro Stufe) für die Migration und das 2PC-Protokoll ein Aufwand von 9 Migrationsnachrichten (jeweils inklusive Agentencode), deren 9 Bestätigungen und 36 Nachrichten für das 2PC-Protokoll ergeben.

Auch für das in Abbildung 4-23 nochmals abgebildete Beispiel aus Abschnitt 3.1 liefert der Algorithmus gute Ergebnisse. Die erste Stufe (n=3) besteht aus den zwei für die Kinos zuständigen

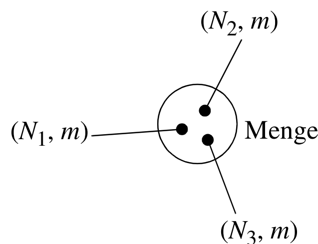


Abbildung 4-22. Einfache Beispielreiseroute

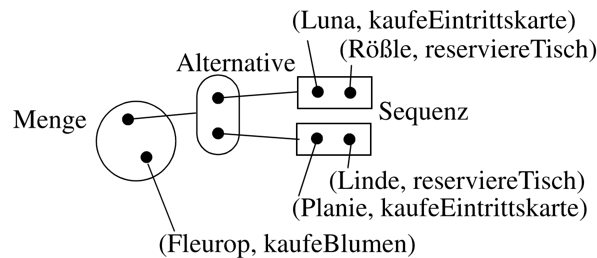


Abbildung 4-23. Beispiel aus Abschnitt 3.1

Knoten und dem Fleurop-Knoten (3 Migrationsnachrichten, 3 Bestätigungen, 4*3 2PC-Nachrichten). Wird der Agent in der ersten Stufe auf dem Knoten des Blumenladen ausgeführt, so besteht die zweite Stufe aus den zwei für die Kinos zuständigen Knoten (reguläre Knoten) und dem Fleurop-Knoten als Ausnahmebehandlungsknoten (zwei Migrationsnachrichten ohne Agentencode, 3 Bestätigungen, 4*2 2PC-Nachrichten). Die dritte Stufe besteht dann aus einem der Restaurant-Knoten (abhängig davon, welcher der Kino-Knoten verwendet wurde), dem Arbeiter der zweiten Stufe und einem weiteren Knoten der zweiten Stufe (zwei Migrationsnachrichten, davon eine ohne Agentencode, 2 Bestätigungen, 4*2 2PC-Nachrichten). Dies ergibt einen Gesamtaufwand von 7 Migrationsnachrichten (Mehraufwand im Vergleich zum Basisprotokoll: 4 Nachrichten), 7 Bestätigungen (Mehraufwand: 4 Nachrichten) und 28 2PC-Nachrichten (Mehraufwand: 16 Nachrichten). Im Gegensatz zum vorhergehenden Beispiel enthielten hier jedoch nur 3 Migrationsnachrichten keinen Agentencode, sodaß hier ein Mehraufwand von einem Codetransport entstand. Wird der Agent in der ersten Stufe anstatt auf dem Fleurop-Rechner auf einem der Restaurant-Knoten ausgeführt, so kann es einen zusätzlichen Code-Transport geben, wenn der Fleurop-Knoten nicht in der zweiten Stufe enthalten ist, da $Next_2$ nur einen Restaurant-Knoten enthält und daher der Algorithmus eventuell die zwei Kino-Knoten als Ausnahmebehandlungsknoten verwendet. In diesem Falle muß der Agentencode in der dritten Stufe erneut auf den Fleurop-Knoten transportiert werden.

Es zeigt sich also, daß die intelligente Zusammensetzung der Knoten einer Stufe den Kommunikationsmehraufwand deutlich senken kann. Eine Implementation des Algorithmus in Kombination mit der von BUSCHLE (1999) entwickelten Reiseroute findet man in PAPOULIDIS (1999).

4.6 Analytische Bewertung der Fehlertoleranz

Die Bewertung der durch ein Protokoll gewonnenen Fehlertoleranz kann prinzipiell auf zwei Arten geschehen: durch Messung aufgrund der Implementierung (entweder in einer realen Umgebung oder einer Simulationsumgebung) des Protokolles oder durch analytische Bewertung. Da in der analytischen Bewertung der Zusammenhang zwischen den Elementen des Protokolles und deren Auswirkung auf die Fehlertoleranz offensichtlicher zu Tage tritt als in der Bewertung mittels Messungen, wird in diesem Abschnitt der analytische Ansatz verfolgt. Der Abschnitt 4.7

befaßt sich dann mit der Messung des durch das Protokoll erzeugten Mehraufwandes.

Um die entwickelten Protokolle analytisch untersuchen zu können ist ein Verfahren notwendig, das die Modellierung der Protokollausführung so exakt wie möglich erlaubt. Hierzu ist die einfache Wahrscheinlichkeitsrechnung, welche i.a. die Unabhängigkeit der betrachteten Ereignisse voraussetzt, nicht geeignet. Es ist daher notwendig, auf die Theorie der stochastischen Prozesse zurückzugreifen. Für die Beschreibung technischer Probleme eignen sich bevorzugt Markov-Modelle (vgl. z.B. SCHNEEWEISS (1992)). Der folgende Abschnitt bietet eine kurze Einführung in die Grundlagen der Markov-Modelle. Anschließend erfolgt die Analyse der entwickelten Protokolle. Die Analyse basiert auf der von FRIEDEL (1998) durchgeführten Diplomarbeit.

4.6.1 Markov-Modelle

Im Gegensatz zu booleschen Modellen, welche die Komponenten eines Systems nur mit den Zuständen “intakt” und “defekt” beschreiben (vgl. SCHNEEWEISS (1973)), erlauben Markov-Modelle die Beschreibung von Systemen mit beliebig großen diskreten oder kontinuierlichen Zustandsmengen. Neben der Berechnung von Ausfallwahrscheinlichkeiten ermöglichen Markov-Modelle auch noch die Berechnung weiterer, für die vorzunehmende Analyse interessanter Kenngrößen (z.B. die mittlere Aufenthaltsdauer in Zustandsmengen).

Eine Einschränkung in der Anwendbarkeit der Markov-Modelle ergibt sich aus der Grundvoraussetzung, daß die betrachteten Lebensdauern und Reparaturzeiten voneinander unabhängig und exponentiell verteilt sein müssen. Eine exponentielle Verteilung der Lebensdauer einer Komponente bedeutet, daß deren Ausfallwahrscheinlichkeit zu jedem Zeitpunkt gleich hoch ist. Die betrachteten Zufallsgrößen besitzen also eine Verteilungsfunktion der Form

$$F(t) = \begin{cases} 0 & \text{für } t < 0 \\ 1 - e^{-\lambda t} & \text{für } t \geq 0, \lambda > 0 \end{cases} \tag{4-15}$$

Für eine λ-exponentiell verteilte Zufallsgröße ergibt sich der Erwartungswert $E(T)$ von T zu

$$E(T) = \frac{1}{\lambda} \tag{4-16}$$

Diese Voraussetzung trifft jedoch auf technische Systeme, also auch auf das hier betrachtete System, weitgehend zu und stellt deshalb keine wesentliche Einschränkung dar.

In diesem Abschnitt werden nun die Grundbegriffe und einige wesentliche Sätze aus dem Bereich der Markov-Modelle basierend auf GAEDE (1977) und HÖFLE-ISPHORDING (1978) eingeführt.

Definition 4-2: *Markov Prozeß*

Ein diskreter stochastischer Prozeß $\{X(t), t \in T\}$ mit der Wertemenge $E=\{0,1,2,...\}$ heißt ein (diskreter) *Markov Prozeß*, wenn für beliebige $t_1 < t_2 < ... < t_n \in T$ und für beliebige Zahlen $i_1, i_2, ..., i_{n-2}, i, j \in E$ folgende Bedingung erfüllt ist:

$$\begin{aligned} &p(X(t_n) = j \mid X(t_{n-1}) = i, X(t_{n-2}) = i_{n-2}, \ldots, X(t_1) = i_1) = \\ &p(X(t_n) = j \mid X(t_{n-1}) = i) \end{aligned} \quad (4\text{-}17)$$

Die Gleichung (4-17) heißt *Markov-Eigenschaft*, die bedingten Wahrscheinlichkeiten heißen *Übergangswahrscheinlichkeiten*.

Anschaulich bedeutet die Markov-Eigenschaft, daß die Wahrscheinlichkeit, sich zum Zeitpunkt t_n in einem Zustand j zu befinden, nur davon abhängt, in welchem Zustand sich das System zum Zeitpunkt t_{n-1} befunden hat ($X(t)$ bezeichnet den Zustand des Systems zum Zeitpunkt t).

Definition 4-3: *Homogenität von Markov Prozessen*

Ein (diskreter) Markov Prozeß heißt *homogener Markov Prozeß*, wenn für ihn weiterhin die folgende Bedingung erfüllt ist:

$$\begin{aligned} &p(X(t+s) = j \mid X(t_0+s) = i) = p(X(t) = j \mid X(t_0) = i) = p_{ij}(t) \\ &\forall i, j \in E, \forall s \geq 0, t \geq 0, t_0 \geq 0 \end{aligned} \quad (4\text{-}18)$$

Anschaulich bedeutet dies, daß bei einem homogenen Markov Prozeß die Übergangswahrscheinlichkeiten $p_{ij}(t)$ nur von der Differenz t_n-t_{n-1} (und nicht von der absoluten Zeit t_n) abhängen. In diesem Falle gilt dann für die Übergangswahrscheinlichkeiten auch die Beziehung

$$\sum_{j \in E} p_{ij}(t) = 1, (i \in E; t \geq 0) \quad (4\text{-}19)$$

Definition 4-4: *Absolute Wahrscheinlichkeit $P_k(t)$*

Die absolute Wahrscheinlichkeit $P_k(t) = p(X(t)=k)$ bezeichnet die Wahrscheinlichkeit, daß sich das System zum Zeitpunkt t im Zustand k befindet.

Satz 4-1:

Ist $\{X(t),\ t \in T\}$ ein diskreter homogener Markov Prozeß mit den Zuständen 1, 2, ..., n und sind die Übergangswahrscheinlichkeiten $p_{ij}(t)$ stetig an der Stelle t=0 und differenzierbar für alle t≥0, dann gilt für die zeitlichen Ableitungen $\dot{p}_{ik}$, $\dot{P}_k$:

$$\dot{p}_{ik}(t) = \sum_{j=1}^{n} a_{kj} p_{ij}(t) \qquad \forall i, k \tag{4-20}$$

$$\dot{P}_k(t) = \sum_{i=1}^{n} a_{ki} P_i(t) \qquad \forall k \tag{4-21}$$

Hierbei ist a_{ji} die Ableitung von $p_{ij}(t)$ an der Stelle t=0 und heißt Übergangsrate von Zustand i nach Zustand j. Da die betrachteten Zufallsgrößen exponentiell verteilt sind, ergibt sich der Erwartungswert für die Übergangszeit von Zustand i in Zustand j zu $1/a_{ji}$. Der Beweis für Gleichung (4-20) und Gleichung (4-21) findet man in HÖFLE-ISPHORDING (1978). Gleichung (4-21) läßt sich mittels Vektoren und Matrizen einfacher darstellen:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \\ \dots & & & \\ a_{n1} & & & a_{nn} \end{bmatrix}, \quad P(t) = \begin{bmatrix} P_1(t) \\ P_2(t) \\ \dots \\ P_n(t) \end{bmatrix} \tag{4-22}$$

$$\dot{P}(t) = A \cdot P(t) \tag{4-23}$$

Die Matrix A wird im weiteren auch als *Übergangsmatrix* bezeichnet.

Das Differentialgleichungssystem kann mittels der Laplace-Transformation einfach gelöst werden.

Bezeichne $\widehat{P}(s) = \begin{bmatrix} \widehat{P}_1(s) \\ \dots \\ \widehat{P}_n(s) \end{bmatrix}$ die Laplace-Transformierte von $P(t) = \begin{bmatrix} P_1(t) \\ \dots \\ P_n(t) \end{bmatrix}$.

Für die Lösung des Gleichungssystems kann man weiterhin die folgende Eigenschaft der Laplace-Transformation für differenzierbare f(t) ausnutzen:

$$\mathrm{L}\left(\frac{d}{dt}f(t)\right) = s \cdot \widehat{f}(s) - f(0)$$

Damit läßt sich Gleichung (4-23) umformen zu

$$s \cdot \widehat{P}(s) - P(0) = A \cdot \widehat{P}(s) \qquad (4\text{-}24)$$

Nach $\widehat{P}(s)$ aufgelöst ergibt sich (mit *n*-reihiger quadratischer Einheitsmatrix *I*)

$$\widehat{P}(s) = (s \cdot I - A)^{-1} \cdot P(0) \qquad (4\text{-}25)$$

Für größere *n* wird die Berechnung der Lösung dieser Gleichung durch die aufwendige Matrizeninversion recht komplex.

Definition 4-5: Stationärer Zustand, stationäre Lösung

Existiert der Grenzwert $\lim_{t \to \infty} P(t)$, so existiert ein *stationärer Zustand*, dem der Prozeß zustrebt. In diesem Fall wird $\lim_{t \to \infty} P(t) = P$ als *stationäre Lösung* bezeichnet.

Satz 4-2:

Ist für einen homogenen Markov Prozeß mit endlich vielen Zuständen die Bedingung

$$\exists t_0 \in T, \text{ so daß gilt } p_{ij}(t_0) > 0 \forall i, j \qquad (4\text{-}26)$$

erfüllt, dann existieren die Grenzwerte $\lim_{t \to \infty} p_{ij}(t)$ und $\lim_{t \to \infty} P_j(t)$ und es gilt $\lim_{t \to \infty} p_{ij}(t) = \lim_{t \to \infty} P_j(t)$

Den Beweis für diesen Satz findet man in LAHRES (1964). Vereinfacht sagt der Satz aus, daß es einen stationären Zustand gibt, falls zu einem beliebigen Zeitpunkt t_0 jeder Zustand von jedem anderen Zustand aus erreichbar ist. Existiert ein stationärer Zustand, so gilt weiter $\dot{P}(t) = 0$ für $t \to \infty$. Dadurch läßt sich Gleichung (4-23) im Grenzwert vereinfachen zu

$$0 = A \cdot P \qquad (4\text{-}27)$$

Weiterhin gilt

$$\sum_{i=1}^{n} \tilde{P}_i = 1, \text{ für die Zustandsmenge } E = \{1, 2, \ldots, n\} \tag{4-28}$$

Definition 4-6: *absorbierender Zustand*

Ein Zustand i heißt absorbierender Zustand wenn gilt $p_{ii}(t) = 1, \forall t \geq 0$

Ein absorbierender Zustand wird also, sobald er einmal erreicht wird, nicht wieder verlassen.

Satz 4-3:

Ist I eine Zustandsmenge, und sind alle Elemente $\notin$ I absorbierend und zum Zeitpunkt t=0 liegt ein Zustand aus I vor, so ist

$$R(t) = \sum_{i \in I} P_i(t) \tag{4-29}$$

die Wahrscheinlichkeit, daß bis zum Zeitpunkt t ein Zustand aus I vorliegt. Bedeutet weiter T_I die Aufenthaltsdauer in der Zustandsmenge I, und existiert der Erwartungswert μ_1 von T_I, so gilt

$$\mu_1 = \lim_{s \to 0} \widehat{R}(s) \tag{4-30}$$

Wegen der Additivität der Laplace-Transformation gilt damit auch:

$$\mu_1 = \sum_{i \in I} \lim_{s \to 0} \widehat{P}_i(s) \tag{4-31}$$

Definition 4-7: *Verfügbarkeit, Dauerverfügbarkeit*

Die Wahrscheinlichkeit, ein System zum Zeitpunkt t im Zustand “intakt” anzutreffen wird als die *Verfügbarkeit V(t)* des Systems zum Zeitpunkt t bezeichnet:
$V(t) = P$[System ist zum Zeitpunkt t intakt]
Im allgemeinen interessiert man sich bei einem System vor allem für die *Dauerverfügbarkeit* $V = \lim_{t \to \infty} V(t)$ die angibt, mit welcher Wahrscheinlichkeit das System zu einem beliebigen Zeitpunkt sich im Zustand “intakt” befindet.

4.6.2 Einschränkung des Fehlermodells

Für die Modellierung der zu untersuchenden Protokolle muß das in Abschnitt 4.1.3 festgelegte Fehlermodell eingeschränkt werden. Bei den Knoten ist keine Einschränkung des Fehlermodells notwendig. Knoten unterliegen Crash-Fehlern, d.h. bei einem Knotenfehler hält der Knoten die Ausführung aller Programme an; der Ausfall einer echten Teilmenge der Prozesse eines Knotens ist ausgeschlossen.

Netzwerkfehler, auch nicht eingeschränkt auf Netzwerkpartitionen, können bei der Untersuchung nicht berücksichtigt werden. Bei Markov-Modellen müssen die zu untersuchenden Zufallsgrößen voneinander unabhängig sein. Dies trifft im Falle der Netzwerkpartitionierung (welche laut Abschnitt 4.1.3 der einzige betrachtete Netzwerkfehler ist) jedoch weder in realen Netzwerken noch in dem in Abschnitt 4.1.2 eingeführten Netzwerkmodell (je ein Kommunikationskanal zwischen zwei Rechnern) zu: entsteht eine Netzwerkpartition, so müssen mehrere Verbindungen gleichzeitig (d.h. abhängig) ausfallen. Neben den Markov-Modellen gibt es zwar weitere, noch wesentlich komplexere Modelle, welche auch mit solchen Fällen zurecht kommen. Mathematisch ist dies aber bei einem derart komplexen Problem kaum noch erfaßbar. Außerdem könnte auch eine Anwendung dieser komplexeren Modelle keine allgemeingültigen Ergebnisse liefern, da die Wahrscheinlichkeiten für Netzwerkpartitionierungen – und damit die Wahrscheinlichkeiten für den Ausfall der Kommunikationskanäle zwischen den Knoten – stark von der Topologie des die Knoten verbindenden Netzwerkes abhängen. Es wäre also jeweils nur eine Aussage für eine bestimmte Ausprägung der Netzwerktopologie möglich.

4.6.3 Verfügbarkeit eines Knotens

Die Verfügbarkeit eines einzelnen Knotens wird in den folgenden Abschnitten immer wieder benötigt und daher an dieser Stelle berechnet. Laut Fehlermodell befindet sich ein Knoten in exakt einem der Zustände “intakt” oder “defekt”.

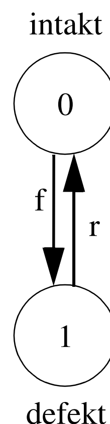


Abbildung 4-24. Modell eines Knotens

Die zwei Zustände des Knoten und die Übergänge zwischen diesen Zuständen zeigt Abbildung 4-24. Mit der Übergangsrate *f* (*Ausfallrate*) geht ein Knoten vom Zustand "intakt" in den Zustand "defekt" über. Der Erwartungswert für diesen Übergang (d.h. die mittlere Zeit bis von Zustand "intakt" in den Zustand "defekt" übergegangen wird) beträgt $1/f$. Ist der Knoten im Zustand "defekt", so wird er repariert und geht mir der Übergangsrate *r* (*Reparaturrate*) in den Zustand "intakt" über. Auch hier beträgt der Erwartungswert für diesen Übergang $1/r$. Die beiden Erwartungswerte werden i.a. auch mit *MTTF* (*Mean Time To Failure, mittlere Zeit bis zum Ausfall*) und *MTTR* (*Mean Time To Repair, mittlere Reparaturzeit*) bezeichnet: $MTTF = 1/f$, $MTTR = 1/r$. Im weiteren wird davon ausgegangen, daß *r* und *f* für alle Knoten gleich sind.

Die Verfügbarkeit $V_K(t)$ des Knotens entspricht der Wahrscheinlichkeit, den Knoten im Zustand 0 ("intakt") vorzufinden, d.h. $V_K(t) = P_0(t)$. Die Übergangsmatrix für das System ergibt sich zu

$$A = \begin{bmatrix} -f & r \\ f & -r \end{bmatrix} \tag{4-32}$$

Für diesen Fall trifft Satz 4-2 zu, somit kann V_K durch lösen des Gleichungssystems (4-27) unter Berücksichtigung von Gleichung (4-28) berechnet werden:

$$0 = \begin{bmatrix} -f & r \\ f & -r \end{bmatrix} \cdot \tilde{P}$$

$$0 = f \cdot \tilde{P}_0 - r \cdot \tilde{P}_1$$

$$1 = \tilde{P}_0 + \tilde{P}_1$$

Hieraus ergibt sich für die Verfügbarkeit V_k:

$$V_K = \tilde{P}_0 = \frac{r}{r+f} \tag{4-33}$$

Durch Einsetzen von $MTTF = 1/f$ und $MTTR = 1/r$ und anschließender Umformung erhält man die bekanntere Formel

$$V_K = \frac{MTTF}{MTTF + MTTR} \tag{4-34}$$

4.6.4 Systemverfügbarkeit und Blockierwahrscheinlichkeit

Ein wichtiges Maß bei der Untersuchung von Fehlertoleranz ist die Verfügbarkeit eines Systems. Bei den in diesem Kapitel vorgestellten Protokollen ist jedoch die Definition, was unter Verfügbarkeit des Systems zu verstehen ist, nicht auf den ersten Blick klar. Die Unterteilung der Verarbeitung des Agenten in abgeschlossene, innerhalb einer Schritt-Transaktion ausgeführte

Schritte legt jedoch die Festlegung nahe, daß das System genau dann verfügbar ist, wenn die zum Abschluß einer Schritt-Transaktion notwendigen Knoten verfügbar sind.

4.6.4.1 Basisprotokoll

Bei der Ausführung eines mobilen Agenten mittels des in Abschnitt 4.3 vorgestellten Basisprotokolls sind während der Ausführung einer Schritt-Transaktion zwei Knoten beteiligt: der Knoten, auf dem der aktuelle Schritt ausgeführt wird und, da der Agent noch innerhalb der Transaktion migriert, der Knoten, auf dem der folgende Schritt ausgeführt werden soll. Der Beitrag dieser beiden Knoten zum Gelingen einer Schritt-Transaktion ist jedoch sehr verschieden: Der Knoten, auf dem der aktuelle Schritt ausgeführt wird ist unverzichtbar, d.h. wenn dieser ausfällt, ist das System für den auszuführenden Agent auf jeden Fall nicht verfügbar. Unter der Annahme, daß für den nächsten auszuführenden Schritt ausreichend viele alternative Knoten zur Verfügung stehen, kann noch vor der Migration getestet werden, welcher dieser Knoten momentan verfügbar ist und einer der verfügbaren Knoten als Ziel der Migration verwendet werden. Dennoch kann natürlich der gewählte Knoten nach diesem Test noch ausfallen. Dieser Ausfall kann jedoch nur zum (verzögerten) Rücksetzen der Schritt-Transaktion (und nicht zum Blockieren) führen, da der Koordinator des 2PC-Protokolles der Schritt-Transaktion auf dem Knoten sitzt, auf dem der aktuelle Schritt ausgeführt wird. In diesem Fall kann die Schritt-Transaktion einfach wiederholt werden – das System steht also immer noch zur Ausführung des Agenten zur Verfügung. Daher läßt sich beim Basisprotokoll die Verfügbarkeit V_s des Systems auf die Verfügbarkeit des Knotens, der den aktuellen Schritt ausführt, reduzieren:

$$V_s = V_k = \frac{r}{r+f} \quad (4\text{-}35)$$

Die Blockierwahrscheinlichkeit B_s bei der Ausführung eines Schrittes, d.h. die Wahrscheinlichkeit daß der Agent während der Ausführung eines Schrittes blockiert wird, berechnet sich mittels

$$B_s = 1 - V_s \quad (4\text{-}36)$$

4.6.4.2 Blockierungsfreies Protokoll

Bei der Ausführung eines mobilen Agenten mittels des in Abschnitt 4.4 vorgestellten blockierungsfreien Protokolls sind während der Ausführung einer Schritt-Transaktion sowohl die Knoten der aktuellen Stufe als auch die Knoten der nächsten Stufe beteiligt. Eine analog dem vorherigen Abschnitt geführte Argumentation erlaubt auch hier, die Betrachtung der Verfügbarkeit auf die Knoten der aktuellen Stufe einzuschränken.

Während für die Ausführung des Schrittes als solchem die Verfügbarkeit eines einzelnen Knotens der Stufe ausreicht, ist für eine erfolgreiche Durchführung des Votierens (und damit den Abschluß der Schritt-Transaktion) die Verfügbarkeit der Mehrheit der Knoten der Stufe zwingend notwendig. Bei einer Stufengröße von *n* Knoten müssen also mindestens $\left\lceil \frac{n+1}{2} \right\rceil$ Knoten der Stufe verfügbar sein (die Gauß-Klammer $\lceil\ \rceil$ bedeutet hierbei, daß zur nächst größeren ganzen Zahl aufgerundet wird). Abbildung 4-25 zeigt das Markov-Modell einer Stufe. Im Zustand *i* der Stufe sind genau *i* Knoten der Stufe intakt. Da im Zustand *i* also *i* Knoten ausfallen können, erfolgt der Übergang zum Zustand *i*-1 (ein weiterer Knoten ausgefallen) mit der *i*-fachen Ausfallrate *if*. Analoges gilt für die Reparaturrate: im Zustand *i* sind *n-i* Knoten ausgefallen, also erfolgt der Übergang zum Zustand *i*+1 mit der Reparaturrate *(n-i)r.*

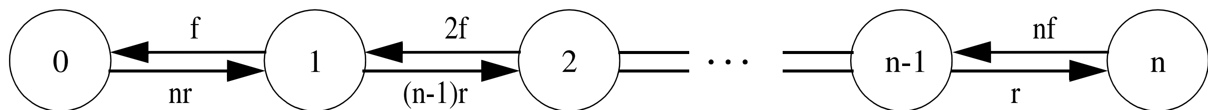


Abbildung 4-25. Modell zur Berechnung der Verfügbarkeit einer Stufe

Aus dem Modell ergibt sich dann (nach Gleichung (4-21)) das dazugehörige Differentialgleichungssystem:

$$\dot{P}_0(t) = -n \cdot r \cdot P_0(t) + f \cdot P_1(t)$$

$$\dot{P}_i(t) = r(n-i+1)P_{i-1}(t) - (i \cdot f + r(n-i))P_i(t) + (i+1) \cdot f \cdot P_{i+1}(t) \qquad i \in \{1, 2, \ldots n-1\}$$

$$\dot{P}_n(t) = r \cdot P_{n-1}(t) - n \cdot f \cdot P_n(t)$$

Da das vorliegende System die Bedingung (4-26) aus Satz 4-2 erfüllt, gilt Gleichung (4-27) und man erhält zusammen mit Gleichung (4-28) für den stationären Fall

$$0 = -n \cdot r \cdot \tilde{P}_0 + f \cdot \tilde{P}_1$$

$$0 = r(n-i+1)\tilde{P}_{i-1} - (i \cdot f + r(n-i))\tilde{P}_i + (i+1) \cdot f \cdot \tilde{P}_{i+1} \qquad i \in \{1, 2, \ldots n-1\}$$

$$0 = r \cdot \tilde{P}_{n-1} - n \cdot f \cdot \tilde{P}_n$$

$$\sum_{i=0}^{n} \tilde{P}_i = 1$$

Die Lösung dieser Gleichung ergibt für die Aufenthaltswahrscheinlichkeit im Zustand i (d.h. Wahrscheinlichkeit, daß genau i Knoten der Stufe verfügbar sind)

$$\tilde{P}_i = \binom{n}{i}\left(\frac{r}{f}\right)^i\left(\frac{f}{f+r}\right)^n, \qquad i \in \{0, 1, \ldots n\} \tag{4-37}$$

Herleitung und Beweis dieser Lösung findet man in HÖFLE-ISPHORDING (1978). Diese Lösung läßt sich unter Verwendung von Gleichung (4-33) umformen zu

$$\tilde{P}_i = \binom{n}{i}\left(\frac{r}{f+r}\right)^i\left(1-\frac{r}{f+r}\right)^{n-i} = \binom{n}{i}(V_K)^i(1-V_K)^{n-i} \tag{4-38}$$

Für p=(Wahrscheinlichkeit, daß ein Knoten verfügbar ist)=V_K ist dies die Binomialverteilung (vgl. HUGHES UND GRAWOIG (1971)) welche die Wahrscheinlichkeit berechnet, daß aus n Elementen genau i Elemente gezogen werden.

Die Verfügbarkeit $V_{s,n}$ eines Systems mit n Knoten pro Stufe ergibt sich dann zu

$$V_{s,n} = P\left[\text{"System in einem der Zustände } \left\lceil\frac{n+1}{2}\right\rceil \ldots n\text{"}\right] = \sum_{i=\left\lceil\frac{n+1}{2}\right\rceil}^{n} \tilde{P}_i \tag{4-39}$$

Für den Spezialfall n=1 ergibt sich aus dieser Formel erwartungsgemäß die Verfügbarkeit des Basisprotokolles.

Die Blockierwahrscheinlichkeit $B_{s,n}$ einer Stufe mit n Knoten berechnet sich zu

$$B_{s,n} = 1 - V_{s,n} \tag{4-40}$$

Die relative Blockierwahrscheinlichkeit $B_{r,n}$ einer Stufe mit n Knoten berechnet sich mittels

$$B_{r,n} = \frac{B_{s,n}}{B_s} \tag{4-41}$$

Eine relative Blockierwahrscheinlichkeit $B_{r,n}$=0.4 bedeutet beispielsweise, daß die Wahrscheinlichkeit, daß ein Agent in einer Stufe mit n Knoten blockiert wird nur 40% der Wahrscheinlichkeit beträgt, daß ein Agent bei der Ausführung mittels des Basisprotokolles blockiert wird.

n	V_k		
	0.75	0.9	0.99
1	100%	100%	100%
2	175%	190%	199%
3	62%	28%	3%
4	105%	52%	6%
5	41%	9%	~0%
6	68%	16%	~0%
7	28%	3%	~0%

Tabelle 4-2. Relative Blockierwahrscheinlichkeit einer Stufe

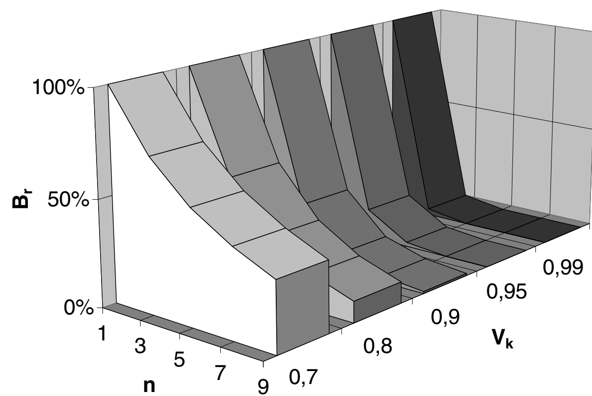


Abbildung 4-26. Relative Blockierwahrscheinlichkeit einer Stufe

Tabelle 4-2 und Abbildung 4-26 zeigen die relative Blockierwahrscheinlichkeit B_r abhängig von der Anzahl der Knoten n und der Einzelverfügbarkeit V_k eines Knotens. Auf ersten Blick fällt auf, daß der Wert für die Blockierwahrscheinlichkeit für ungerade n immer kleiner ist als der Wert für das größere n+1, sich die Blockierwahrscheinlichkeit also bei der Erhöhung der Knotenzahl von einem ungeraden n auf ein gerades n+1 erhöht. Dies ist jedoch verständlich, da für eine Stufe mit einer ungeraden Knotenzahl n für das erfolgreiche Votieren relativ gesehen weniger Knoten ((n/2)+0.5) vorhanden sein müssen als bei einer Stufe mit einer geraden Knotenanzahl ((n/2)+1). Insgesamt zeigt sich, daß ab einer Knotenanzahl von 3 Knoten pro Stufe die relative Blockierwahrscheinlichkeit wesentlich verringert wird und daß die Verwendung einer ungeraden Anzahl von Knoten pro Stufe empfehlenswert ist.

4.6.5 Verweildauer in einer Stufe

Die Ergebnisse des letzten Abschnittes sind nur begrenzt aussagekräftig, da sie auf der Betrachtung von Dauerverfügbarkeiten basieren, die über den zeitlichen Verlauf der Verfügbarkeit wenig aussagen. Die Dauerverfügbarkeit hängt jedoch nur vom Verhältnis zwischen Ausfallrate und Reparaturrate ab, d.h. ein Knoten hat auch dann eine hohe Dauerverfügbarkeit, wenn die Ausfallrate zwar recht hoch ist, die Reparaturrate aber entsprechend kurz ausfällt. Bei hoher Ausfallrate sinkt die Zeit zwischen Knotenausfällen. Wird diese Zeit kürzer als die durchschnittliche Zeit zur Ausführung eines Schrittes eines Agenten, so sinkt die Wahrscheinlichkeit, daß ein Schritt eines Agenten vollständig ausgeführt werden kann gegen null – der Agent wird trotz hoher Verfügbarkeit blockiert.

Ein besseres Maß zur Bewertung der gewonnenen Fehlertoleranz, welches auch die Problematik des Verhältnisses zwischen Knotenausfallzeit und Zeit zur Ausführung eines Schrittes berücksichtigt, besteht darin, die durchschnittliche Verweildauer eines Agenten in einer Stufe zu berechnen. Diese Zeit umfaßt den Zeitraum von der Ankunft des Agenten in der Stufe bis zum

Commit der Schritt-Transaktion und berücksichtigt auch zusätzliche Zeiten, die durch Ausfälle von Knoten entstehen können.

Einen Ansatz zur Berechnung dieser durchschnittlichen Verweildauer bietet Satz 4-3. Während der Verarbeitung eines Agenten in einer Stufe tritt nie ein absorbierender Zustand ein, nur der Abschluß der Schritt-Transaktion mündet in einen absorbierenden Zustand. Da die bei der Berechnung der Verweildauer des blockierungsfreien Protokolles entstehenden Übergangsmatrizen sehr groß werden, ist es sinnvoll, das Problem soweit als möglich in kleinere Teilprobleme zu zerlegen. Die 2-Phasigkeit des Commit-Protokolles bietet eine einfache Möglichkeit der Unterteilung in 2 Abschnitte: Der erste Abschnitt beginnt mit dem Eintreffen des Agenten in der Stufe und endet mit der Commit-Entscheidung des Koordinators. Sobald der Koordinator sich für ein Commit entschieden hat, ist sichergestellt, daß die Transaktion und damit auch die Stufe vollends beendet werden kann - ein Zurücksetzen der Transaktion (und somit ein Übergang in den ersten Abschnitt) ist dann nicht mehr möglich. Der zweite Abschnitt umfaßt lediglich die Verteilung der Commit-Entscheidung an die beteiligten Ressourcen.

Die gesamte Verweildauer $T_{V,b}$ eines Agenten in einer Stufe beim Basisprotokoll ist dann die Summe der Verweildauer $T_{W,b}$ des ersten Abschnitts und der Verweildauer $T_{C,b}$ des zweiten Abschnitts:

$$T_{V,b} = T_{W,b} + T_{C,b} \tag{4-42}$$

4.6.5.1 Basisprotokoll

Erster Abschnitt:

Bei der Ausführung des ersten Abschnittes sind insgesamt zwei Knoten beteiligt: Der den Agenten ausführende Knoten und der Zielknoten der Migration. Um die Verweildauer zu berechnen macht jedoch eine Modellierung, in der die Zustände direkt mit der Anzahl der verfügbaren Knoten korreliert sind (d.h. ein Modell des physischen Systems), wenig Sinn. Vielmehr müssen Zustände des Protokolles selbst modelliert werden. Abbildung 4-27 zeigt das Modell zur Berechnung der Verweildauer im ersten Abschnitt.

Die Verarbeitung des ersten Abschnittes gliedert sich in drei verschiedene Zustände und einen zusätzlichen Endzustand. Sobald ein Agent bei einem Knoten eintrifft, befindet er sich im Zustand “Ausführen”. Hier wird die Schritt-Transaktion begonnen und der auszuführende Schritt abgearbeitet. Danach geht die Verarbeitung in den Zustand “Vorbereiten” über, in dem der Agent in die Eingangswarteschlange des nächsten Knotens geschrieben wird und in dem der Koordinator der Transaktion *rm_prepare* auf allen beteiligten Ressourcen aufruft. Neben der lokalen Eingangswarteschlange und der Eingangswarteschlange des nächsten Knotens sind dies noch jene lokalen Ressourcen, auf die der Agent während seiner Ausführung zugegriffen hat.

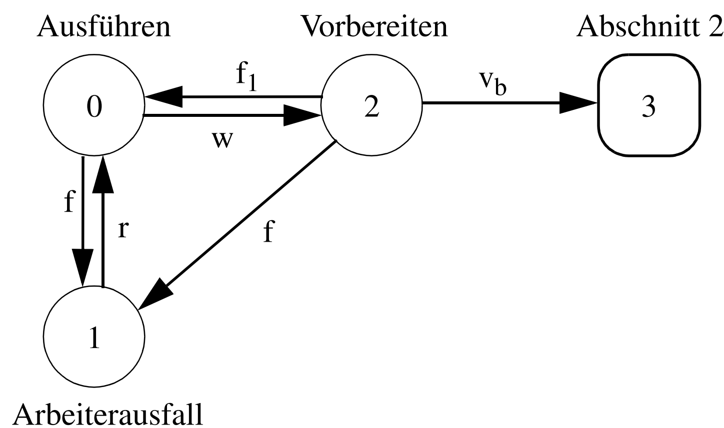


Abbildung 4-27. Modell zur Berechnung der Verweildauer des ersten Abschnitts des Basisprotokoll

Fällt der ausführende Knoten aus während sich das System in den Zuständen “Ausführen” oder “Vorbereiten” befindet, so ist die Ausführung des Agenten blockiert und geht in den Zustand “Arbeiterausfall” über. Fällt der Zielknoten der Migration im Zustand “Vorbereiten” aus, so wird die Transaktion abgebrochen und der Agent erneut im Zustand “Ausführen” ausgeführt. Entscheidet der Transaktionskoordinator mit “Commit”, so wird in den 2. Abschnitt des Protokolles übergegangen. Entscheidet der Transaktionskoordinator aus irgendwelchen Gründen mit “Abort” (z.B. Probleme in Ressourcen, Deadlocks,...), so ist die Transaktion ebenfalls abgebrochen und es wird in den Zustand “Ausführen” übergegangen.

Die Übergänge zum Zustand “Arbeiterausfall” finden mit der Ausfallrate f eines einzelnen Knotens statt. Die Übergangsrate f_1 vom Zustand “Vorbereiten” setzt sich zusammen aus der Ausfallrate f eines Knotens und einer Ausfallrate, welche sich aus der Wahrscheinlichkeit ergibt, mit der eine Transaktion vom Koordinator abgebrochen wird. Da dies nur sehr selten geschieht und quantitativ nur schlecht erfaßt werden kann (da abhängig von den Ressourcen, auf die der Agent während seiner Ausführung zugreift), wird f_1 durch f angenähert. Die mittlere Ausführungszeit $1/w$ setzt sich zusammen aus der Zeit $1/b$ zum Starten einer Transaktion, der mittleren Zeit $1/l$ zum Lesen des Agenten aus der Eingangswarteschlange und der mittleren Zeit $1/e$ zur (fehlerfreien) Ausführung des auszuführenden Schrittes. Es gilt

$$\frac{1}{w} = \frac{1}{b} + \frac{1}{l} + \frac{1}{e}.$$

Geht man davon aus, daß die Zeiten zum Start einer Transaktion und zum Lesen des Agenten aus der Eingangswarteschlange wesentlich kleiner als die Zeit zur Ausführung des Schrittes sind, so ist $1/w$ ungefähr die mittlere Dauer der Ausführung eines Schrittes. Die mittlere Ausführungszeit $1/v_b$ setzt sich zusammen aus der mittleren Ausführungszeit $1/s$ zum Schreiben des Agenten in die Warteschlange des nächsten Knotens und der mittleren Ausführungszeit $2*1/p_s$

der ersten Phase des 2PC-Protokolles (2*mittlere Zeit zum Aufruf von *rm_prepare* auf eine Eingangswarteschlange). Hier gilt

$$\frac{1}{v_b} = \frac{1}{s} + 2\frac{1}{p_s}.$$

Als Übergangsmatrix für das System erhält man

$$A = \begin{bmatrix} -(f+w) & r & f & 0 \\ f & -r & f & 0 \\ w & 0 & -(2f+v_b) & 0 \\ 0 & 0 & v_b & 0 \end{bmatrix}$$

Die Laplace-Transformierte berechnet sich nach Gleichung (4-25) zu

$$\widehat{P}(s) = (s \cdot I - A)^{-1} \cdot P(0) = \begin{bmatrix} s+f+w & -r & -f & 0 \\ -f & s+r & -f & 0 \\ -w & 0 & s+2f+v_b & 0 \\ 0 & 0 & -v_b & s \end{bmatrix}^{-1} \cdot P(0)$$

Da der Agent sich nach dem Eintreffen auf dem Knoten nur im Zustand “Ausführen” befinden kann (im Zustand “Arbeiterausfall” hätte er gar nicht in die Eingangswarteschlange geschrieben werden können), ist die Anfangsbedingung für die Aufenthaltswahrscheinlichkeiten

$$P(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Nach Berechnung der Inversen der Laplace-Transformierten und dem Einsetzen von P(0) kann man dann den Grenzwert von $\widehat{P}(s)$ für $s \to 0$ berechnen:

$$\lim_{s \to 0} \widehat{P}(s) = \begin{bmatrix} \frac{2f+v_b}{v_b w} \\ \frac{f(2f+v_b+w)}{r v_b w} \\ \frac{1}{v_b} \\ \infty \end{bmatrix}$$

Nach Satz 4-3, Gleichung (4-31) erhält man den Erwartungswert T_W der Aufenthaltsdauer in der Zustandsmenge {0,1,2}, und damit die Verweildauer in Abschnitt 1 des Basisprotokolles, mittels

$$T_{W,b} = \sum_{i=0}^{2} \widehat{P}(0) = \frac{(2f + w + v_b)(f + r)}{wrv_b} \qquad (4\text{-}43)$$

Zweiter Abschnitt:

Im zweiten Abschnitt des Basisprotokolles muß nur noch die zweite Phase des 2PC-Protokolles abgehandelt werden, genauer gesagt es muß die Commit-Entscheidung des Transaktionskoordinators an die beteiligten Ressourcen übermittelt werden (beim Abbruch der Transaktion würde der erste Abschnitt nicht verlassen). Beteiligte Ressourcen sind die lokale Eingangswarteschlange, die Eingangswarteschlange des nachfolgenden Knotens und weitere lokale Ressourcen, auf die der Agent während der Schritt-Transaktion zugegriffen hat.

Die Commit-Entscheidung kann den Ressourcen seriell oder parallel mitgeteilt werden. Bei der seriellen Mitteilung wird einer Ressource die Commit-Entscheidung mitgeteilt und auf deren Antwort gewartet, bevor einer weiteren Ressource die Commit-Entscheidung mitgeteilt wird. Beim parallelen Fall werden an alle Ressourcen die Commit-Entscheidungen verschickt und dann auf die Antworten gewartet. An dieser Stelle wird der zeitlich ungünstigere Fall, die serielle Mitteilung, betrachtet. Es wird hierbei angenommen, daß, sobald eine Ressource die Commit-Mitteilung bestätigt hat, die Ressource die Commit-Entscheidung auch bei auftretenden Fehlern nicht erneut übermittelt bekommt. Somit kann man die Commit-Mitteilungen der einzelnen Ressourcen getrennt betrachten.

Abhängig davon, wo der Koordinator der Transaktion sitzt und wo sich die zu benachrichtigende Ressource befindet, müssen unterschiedliche Fehlermodelle betrachtet werden. Sitzen Koordinator und Ressource auf demselben Knoten (lokale Ressource), so trifft das Modell aus Abbildung 4-28 zu.

In diesem Falle kommt es nur darauf an, ob der Knoten, auf dem sich der Transaktionskoordinator und die Ressource befinden, intakt ist oder nicht. Fällt der Knoten aus, während bei der lokalen Ressource der Commit durchgeführt wird, so wird der Ressource nach Reparatur des Knotens erneut die Commit-Entscheidung mitgeteilt. Die mittlere Commit-Dauer im fehlerfreien Fall, und somit die mittlere Zeit bis zum Übergang in den (End-)Zustand 2, beträgt $1/c_l$.

Im hier betrachteten Basisprotokoll sitzt der Transaktionskoordinator i.a. auf dem Knoten, auf dem die Schritt-Transaktion durchgeführt wird. Lokale Ressourcen sind in diesem Falle die Eingangswarteschlange des Knotens und jene Ressourcen, auf die der Agent während der Ausführung des Schrittes zugegriffen hat. Da diese für den allgemeinen Fall nicht bekannt sind, wird hier nur die Zeit des lokalen Commit für die Eingangswarteschlange (mit mittlerer Commit-Dauer $1/c_l$) betrachtet. Die Berechnung der Zeiten für weitere lokale Ressourcen erfolgt analog.

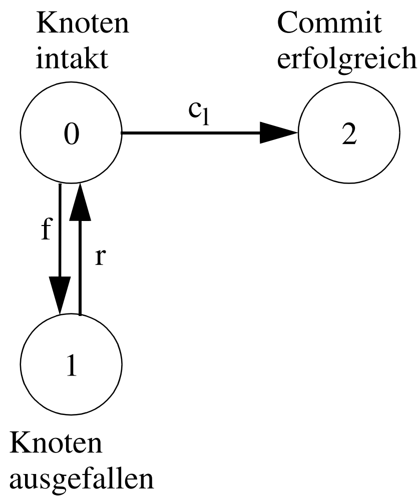


Abbildung 4-28. Modell für lokales Commit

Da für den Beginn dieser Phase der den Agenten ausführende Knoten intakt sein muß, ergibt sich die Anfangsbedingung zu

$$P(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Die Übergangsmatrix ergibt sich wie folgt:

$$A = \begin{bmatrix} -(f + c_l) & r & 0 \\ f & -r & 0 \\ c_l & 0 & 0 \end{bmatrix}$$

Analog zum letzten Abschnitt wird die mittlere Aufenthaltsdauer $T_{cl,b}$ in der Zustandsmenge {0,1} berechnet:

$$\widehat{P}(s) = (s \cdot I - A)^{-1} \cdot P(0) = \begin{bmatrix} s + f + c_l & -r & 0 \\ -f & s + r & 0 \\ -c_l & 0 & s \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\lim_{s \to 0} \widehat{P}(s) = \begin{bmatrix} \frac{1}{c_l} \\ \frac{f}{c_l r} \\ \infty \end{bmatrix}$$

$$T_{cl,b} = \sum_{i=0}^{1} \widehat{P}(0) = \frac{r+f}{c_l r} \tag{4-44}$$

Befinden sich der Koordinator der Transaktion und die Ressource (hier: Eingangswarteschlange des Knotens, auf den Agent migriert) auf unterschiedlichen Knoten, so trifft das Modell aus Abbildung 4-29 zu. Hier spielen sowohl der Knoten des Transaktionskoordinators als auch der Knoten der Ressource eine Rolle.

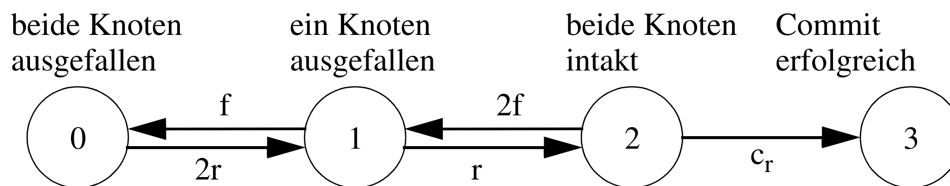


Abbildung 4-29. Modell für entferntes Commit

Nur wenn diese beide Knoten intakt sind, kann die Commit-Entscheidung der Ressource mitgeteilt und von der Ressource verarbeitet werden, daher ist ein Übergang in den (End-)Zustand 3 nur aus Zustand zwei möglich. Die Ausfallrate im Zustand “beide Knoten intakt” entspricht dem doppelten der normalen Ausfallrate *f*, dasselbe gilt für die Reparaturrate, wenn beide Knoten ausgefallen sind.

Die Anfangsbedingungen sind auch hier einfach zu bestimmen. Wird diese Phase des Protokolles erreicht, dann ist der Knoten des Transaktionskoordinators auf jeden Fall nicht ausgefallen. Somit kann also nur ein Knoten (der Knoten der Zielwarteschlange) oder kein Knoten ausgefallen sein. Die Wahrscheinlichkeit, daß der Knoten der Zielwarteschlange verfügbar ist erhält man analog Gleichung (4-33). Da sich das System anfangs nicht im Endzustand befinden kann, erhält man die Anfangsbedingungen

$$P(0) = \begin{bmatrix} 0 \\ 1 - \frac{r}{r+f} \\ \frac{r}{r+f} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{f}{r+f} \\ \frac{r}{r+f} \\ 0 \end{bmatrix}$$

Die Übergangsmatrix des Modells ergibt sich zu

$$A = \begin{bmatrix} -2r & f & 0 & 0 \\ 2r & -(f+r) & 2f & 0 \\ 0 & r & -(2f+c_r) & 0 \\ 0 & 0 & c_r & 0 \end{bmatrix}$$

Wie zuvor kann man hieraus nun T_{cr} als mittlere Aufenthaltsdauer in der Zustandsmenge {0,1,2,3} berechnen:

$$\widehat{P}(s) = (s \cdot I - A)^{-1} \cdot P(0) = \begin{bmatrix} s+2r & -f & 0 & 0 \\ -2r & -s+f+r & (-2)f & 0 \\ 0 & -r & s+2f+c_r & 0 \\ 0 & 0 & -c_r & s \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ \frac{f}{r+f} \\ \frac{r}{r+f} \\ 0 \end{bmatrix}$$

$$\lim_{s \to 0} \widehat{P}(s) = \begin{bmatrix} \frac{1}{2}\frac{f^2(2f+c_r+2r)}{r^2 c_r(r+f)} \\ \frac{f(2f+c_r+2r)}{rc_r(r+f)} \\ \frac{1}{c_r} \\ \infty \end{bmatrix}$$

$$T_{cr,b} = \sum_{i=0}^{2} \widehat{P}(0) = \frac{1}{2}\frac{f(2f+c_r+2r)(f+2r)}{r^2 c_r(r+f)} \tag{4-45}$$

Die Gesamtzeit $T_{C,b}$ für den Commit ergibt sich also zu

$$T_{C,b} = T_{cl,b} + T_{cr,b} = \frac{r+f}{c_l r} + \frac{1}{2}\frac{f(2f+c_r+2r)(f+2r)}{r^2 c_r(r+f)} \tag{4-46}$$

Zusammenfassung:

Als Verweildauer eines Agenten in einer Stufe ergibt sich beim Basisprotokoll der Wert

$$T_{V,b} = T_{W,b} + T_{C,b} = \frac{(2f+w+v_b)(f+r)}{wrv_b} + \frac{1}{2}\frac{f(2f+c_r+2r)(f+2r)}{r^2 c_r(r+f)} + \frac{r+f}{c_l r} \tag{4-47}$$

4.6.5.2 Blockierungsfreies Protokoll

Da das blockierungsfreie Protokoll wesentlich komplexer als das Basisprotokoll ist, wird auch die Modellierung des Protokolles wesentlich komplexer. Um die Komplexität in einem akzeptablen Rahmen zu halten, werden deshalb an einigen Stellungen Abschätzungen nach oben vorgenommen, d.h. es wird meist der jeweils schlechteste Fall angenommen. Für die Verweildauer bedeutet dies, daß der reale Wert günstiger (d.h. kleiner) ausfällt als der hier berechnete Wert. Auch hier wird wieder dieselbe Unterteilung wie beim Basisprotokoll in zwei Protokollabschnitte vorgenommen. Für die Berechnungen wird der ungünstigere Fall angenommen, daß die Knotenmengen zweier aufeinanderfolgender Stufen disjunkt sind.

Erster Abschnitt:

Bei der Aufstellung des in Abbildung 4-30 dargestellten Modells wurde, wie schon im vorhergehenden Abschnitt, der "Verarbeitungszustand" einer Stufe als Basis für die Zustände des Modells genommen.

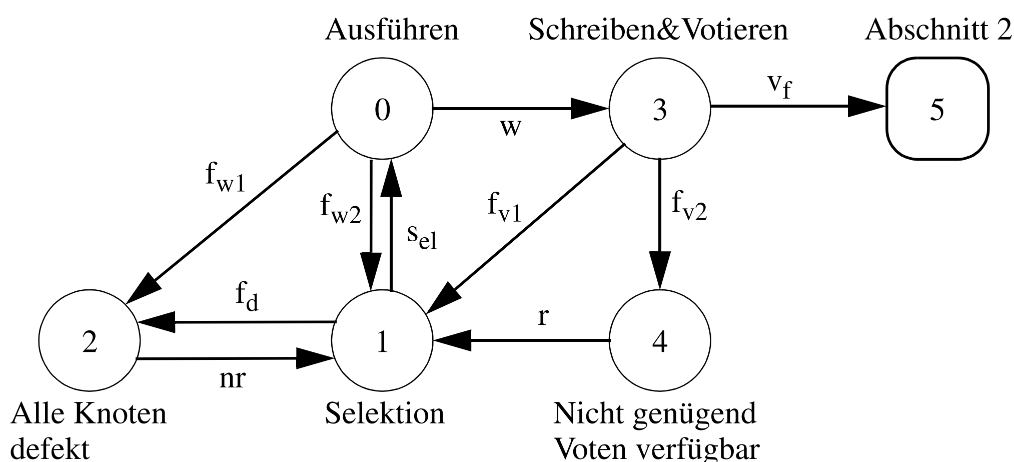


Abbildung 4-30. Modell zur Berechnung der Verweildauer des ersten Abschnitts des blockierungsfreien Protokolles

Nachdem der Agent in der Stufe angekommen ist, wird er vom Arbeiter (i.a. der Knoten mit der höchsten Priorität) ausgeführt, befindet sich im Modell also im Zustand 0. Die Ausführung umfaßt den Start der Schritt-Transaktion, das Lesen des Agenten aus der Eingangswarteschlange, die Ausführung des Schrittes und die Bestimmung der Knoten der nachfolgenden Stufe. Hierbei kann davon ausgegangen werden, daß immer genug Knoten vorhanden sind, um eine Stufe mit n Knoten zu bilden. Danach geht das Modell mit der Übergangsrate w in den Zustand 3, "Schreiben&Votieren", über. Die mittlere Ausführungszeit $1/w$ setzt sich zusammen aus der Zeit $1/b$ zum Starten einer Transaktion, der mittleren Zeit $1/l$ zum Lesen des Agenten aus der Eingangswarteschlange, der mittleren Zeit $1/e$ zur (fehlerfreien) Ausführung des auszuführenden Schrit-

tes (inklusive der Berechnung der Knoten der nächsten Stufe). Es gilt

$$\frac{1}{w} = \frac{1}{b} + \frac{1}{l} + \frac{1}{e}.$$

Da die Zeit zur Ausführung eines Schrittes i.a. wesentlich höher ist als die Zeit zur Ermittlung der Knoten der nächsten Stufe, kann für $1/e$ die mittlere Zeit zur Ausführung eines Schrittes angenommen werden.

Im Zustand "Schreiben&Votieren" wird zuerst der Agent in die Eingangswarteschlangen der Knoten der nächsten Stufe geschrieben und dann die erste Phase des 2PC-Protokolles durchgeführt. Diese schließt das Votieren ein, bei dem der Orchestrator von allen Votierern der Stufe ein Votum eintreibt. Hat der Orchestrator eine Mehrheit der Voten erhalten und haben alle Ressourcen dem Abschluß der Transaktion zugestimmt, geht die Verarbeitung in den zweiten Abschnitt über. Die Übergangsrate in den zweiten Abschnitt berechnet sich aus der für das Schreiben des Agenten in die nächste Stufe benötigten Zeit $n*1/s$ (für n Knoten in der nächsten Stufe) und der für die Durchführung der ersten Phase des 2PC-Protokolles. In der ersten Phase des 2PC-Protokolles wird *rm_prepare* bei den beteiligten Ressourcenmanagern aufgerufen. Dies betrifft in diesem Falle die Eingangswarteschlangen der Knoten der nächsten Stufe, die Eingangswarteschlange des Arbeiters, den Orchestrator und die Ressourcen, auf die der Agent während dem Schritt zugegriffen hat. Nimmt man für den Aufruf von *rm_prepare* auf eine Eingangswarteschlange eine mittlere Ausführungszeit von $1/p_s$ an, so ergibt sich eine mittlere Ausführungszeit für die erste Phase des 2PC-Protokolles von

$$T_{2pc1} = (n+1)\frac{1}{p_s} + T_v$$

und damit gilt für die Übergangsrate von Zustand 3 in Zustand 5

$$\frac{1}{v_f} = n\frac{1}{s} + (n+1)\frac{1}{p_s} + T_v \qquad (4\text{-}48)$$

Hierbei wird der ungünstigere Fall des sequentiellen Aufrufes von *rm_prepare* auf die Eingangswarteschlangen angenommen. T_v bezeichnet die vom Orchestrator benötigte Zeit für den *rm_prepare*-Aufruf und umfaßt vor allem die für das Votieren benötigte Zeit. Die Berechnung von T_v hängt von der Verfügbarkeit der Knoten der aktuellen Stufe ab und ist ziemlich komplex. Aus Gründen der Übersichtlichkeit erfolgt die Berechnung deshalb weiter unten.

Treten Knotenausfälle auf, so ist die Auswirkung des Ausfalls stark von dem Zustand abhängig, in dem sich die Verarbeitung befindet. Im Zustand "Ausführen" wird das System nur dann beeinflußt, wenn der Arbeiter, d.h. der den Agent ausführende Knoten, ausfällt. Fällt der Arbeiter aus, sind zwei Alternativen denkbar. Solange noch mindestens ein Knoten der Stufe verfügbar ist, geht das System in den Zustand "Selektion" über. War der Arbeiter der letzte verfügbare Knoten der Stufe, so geht das System in den Zustand "Alle Knoten defekt" über. Die Ausfallrate

des Arbeiters ist f (Ausfall eines Knotens). Diese Rate muß nun aufgeteilt werden zwischen dem Übergang in den Zustand “Selektion” und dem Übergang in “Alle Knoten defekt”. Fällt der Arbeiter aus, so ist die Wahrscheinlichkeit, daß er in den Zustand “Alle Knoten defekt” übergeht gleich der Wahrscheinlichkeit P_{w1}, daß alle n-1 restlichen Knoten der Stufe ausgefallen sind. Diese Wahrscheinlichkeit berechnet sich nach Gleichung (4-38) zu

$$P_{w1} = \left(\frac{f}{f+r}\right)^{n-1} \tag{4-49}$$

Für die Übergangsrate f_{w1} vom Zustand “Ausführen” in den Zustand “Alle Knoten defekt” erhält man dann

$$f_{w1} = P_{w1} \cdot f = f\left(\frac{f}{f+r}\right)^{n-1} \tag{4-50}$$

Die Übergangsrate f_{w2} vom Zustand “Ausführen” in den Zustand “Selektion” ergibt sich dann zu

$$f_{w2} = f - f_{w1} = f\left(1 - \left(\frac{f}{f+r}\right)^{n-1}\right) \tag{4-51}$$

Im Zustand “Schreiben&Votieren” wirkt sich (beinahe) jeder Ausfall eines der an der Verarbeitung der Stufe beteiligten Knoten auf das System aus. Fällt einer der Knoten der nächsten Stufe aus, so wird die Schritt-Transaktion zurückgesetzt und das System geht in den Zustand “Selektion” über. Fallen Beobachter der Stufe aus, so kann das Auswirkungen auf die zum Votieren benötigte Zeit T_v (siehe oben) haben, falls nicht genug Knoten für das Erlangen einer Voten-Mehrheit verfügbar sind. Fällt der Arbeiter selbst aus, dann hat der Orchestrator zu diesem Zeitpunkt schon von 0,...,n Knoten eine Stimme bekommen. Hat der Orchestrator schon die benötigte Stimmenzahl erhalten, so kann kein weiterer Orchestrator die notwendige Anzahl an Stimmen erhalten. In diesem Fall ist das System blockiert, bis der Arbeiter-Knoten neu startet (und die erhaltenen Voten zurückgeben kann). Hat der Arbeiter jedoch noch nicht genug Stimmen bekommen, dann könnte das System entweder in den Zustand “Selektion” oder in den Zustand “Alle Knoten defekt” übergehen. Problem hier ist, daß einerseits nur sehr schlecht eine Aussage getroffen werden kann, mit welcher Wahrscheinlichkeit nach Ausfall eines Arbeiters noch genügend Stimmen vorhanden sind und andererseits umfaßt das Modell nicht die Möglichkeit, daß eventuell mehrere “ehemalige” (d.h. während dem Votieren ausgefallene) Arbeiter Teile der Stimmen blockieren. Aus diesem Grunde wird der für die Ausführungszeit ungünstigere Fall angenommen, daß bei Ausfall des Arbeiters prinzipiell nicht mehr genügend Voten vorhanden sind, sodaß hier immer in den Zustand “Nicht genügend Voten” übergegangen wird. Die Rate f_{v1} für den Übergang vom Zustand “Votieren” in den Zustand “Selektion” hängt demnach nur

vom Ausfall eines der Knoten der nachfolgenden Stufe ab und ergibt sich daher zu

$$f_{v1} = nf \quad (4\text{-}52)$$

Dabei wurde (der ungünstigere Fall) angenommen, daß der Ausfall eines Knotens der nächsten Stufe auch dann den Abbruch der Transaktion bewirkt, wenn die Eingangswarteschlange bereits mit *rm_yes* auf das *rm_prepare* geantwortet hat. Die Rate f_{v2} für den Übergang vom Zustand "Votieren" in den Zustand "Nicht genügend Voten verfügbar" hängt nach obiger Festlegung nur vom Ausfall des Arbeiters ab und ergibt sich deshalb zu

$$f_{v2} = f \quad (4\text{-}53)$$

Befindet sich das System im Zustand "Nicht genügend Voten verfügbar", dann wird in der Realität nach der Wahl eines neuen Arbeiters bereits mit der Ausführung des Agenten begonnen.Im Modell wird jedoch (der ungünstigere Fall) angenommen, daß die Auswahl des neuen Arbeiters erst dann beginnt, wenn der alte Arbeiterknoten wieder verfügbar ist. Die Übergangsrate von "Nicht genügend Knoten verfügbar" in den Zustand "Selektion" entspricht daher der Reparaturrate eines einzelnen Knotens *r*.

Vom Zustand "Selektion" aus kann das System in zwei Nachfolgezustände übergehen. Wird ein neuer Arbeiter gewählt, so geht das System in den Zustand "Ausführen" über. Die Übergangsrate hierfür sei s_{el}. Bei der Wahl dieser Rate wird auch berücksichtigt, daß der Knoten mit der höchsten Priorität verfügbar sein kann (und in diesem Fall die Selektionsphase sehr kurz ist). Fällt im Zustand "Selektion" ein Knoten aus ändert dies nichts – es sei denn, alle anderen (n-1) Knoten der Stufe sind schon ausgefallen. In diesem Falle geht das System in den Zustand "Alle Knoten defekt" über. Da dies dieselbe Situation ist wie beim Übergang vom Zustand "Ausführen" in den Zustand "Alle Knoten defekt" ergibt sich für den Übergang "Selektion" nach "Alle Knoten defekt" die Übergangsrate analog zu Gleichung (4-50):

$$f_d = f_{w1} = f\left(\frac{f}{f+r}\right)^{n-1} \quad (4\text{-}54)$$

Der Zustand "Alle Knoten defekt" wird in Richtung "Selektion" verlassen, sobald einer der n Knoten repariert ist. Die Übergangsrate für diesen Übergang ist nr (d.i. die n-fache Reparaturrate eines einzelnen Knotens).

Bevor nun die Verweildauer des Protokolles im ersten Abschnitt berechnet werden kann, muß noch die mittlere Zeit ausgerechnet werden, die der Orchestrator zum Votieren benötigt. Sind von der aktuellen Stufe ausreichend (d.h. $\lceil (n+1)/2 \rceil$) Knoten verfügbar (blockierungsfreier Fall), so bewegt sich die Dauer T_{vn} für das Votieren in der Größenordnung der Round-Trip-

Time (Zeit, um Datenpaket zu einem Knoten hin und wieder zurück zu transportieren) zwischen den Knoten der Stufe. Sind jedoch nicht ausreichend Knoten vorhanden ist der Agent blockiert und muß warten, bis ausreichend Knoten für eine Stimmenmehrheit verfügbar sind.

Die mittlere Dauer T_v des Votierens ergibt sich aus der mittleren Dauer T_{vn} im blockierungsfreien Fall gewichtet mit dessen Wahrscheinlichkeit P_n und der mittleren Dauer T_{vb} im Fehlerfall gewichtet mit dessen Wahrscheinlichkeit P_b:

$$T_v = P_n T_{vn} + P_b T_{vb} \tag{4-55}$$

Für die Berechnung von P_n und P_b kann davon ausgegangen werden, daß der Arbeiter immer verfügbar ist (fällt der Arbeiter aus wird in einen Zustand übergegangen, der momentan nicht Gegenstand der Betrachtung ist). Das Protokoll blockiert folglich nicht, wenn von den n-1 Beobachtern mindestens $\lceil (n-1)/2 \rceil$ Beobachter verfügbar sind. Der Wert für P_n ergibt sich direkt analog zu Gleichung (4-39) und mit Gleichung (4-38) zu

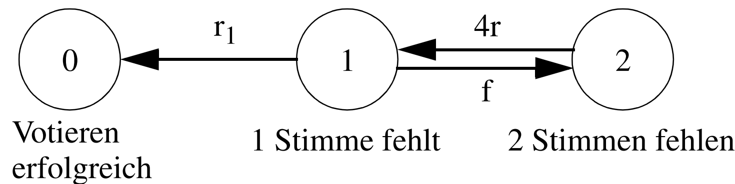
$$P_n = \sum_{i=\left\lceil \frac{n-1}{2} \right\rceil}^{n-1} \left(\frac{r}{f}\right)^i \binom{n-1}{i} \left(\frac{f}{f+r}\right)^{n-1} = \left(\frac{f}{f+r}\right)^{n-1} \sum_{i=\left\lceil \frac{n-1}{2} \right\rceil}^{n-1} \left(\frac{r}{f}\right)^i \binom{n-1}{i} \tag{4-56}$$

Die Wahrscheinlichkeit P_b ergibt sich dann einfach zu

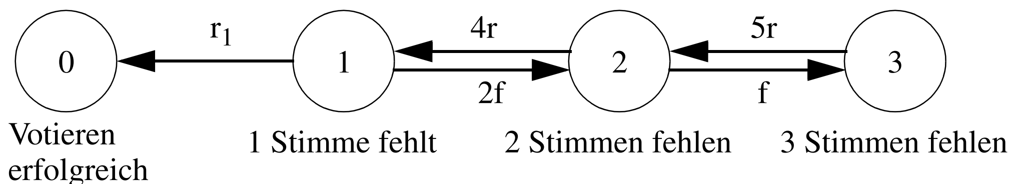
$$P_b = 1 - P_n \tag{4-57}$$

Die Berechnung der mittleren Zeit für das Votieren bei einer Blockierung hängt vor allem von der Anzahl der Knoten in einer Stufe ab. Abbildung 4-31, welche Modelle für das Votieren für unterschiedliche Anzahlen von Knoten pro Stufe zeigt, läßt vermuten, daß hier nur sehr schwer eine geschlossene Form zur Berechnung der mittleren Zeit zu finden ist. Die Zustände im Modell modellieren jeweils die Anzahl der für ein vollständiges Votum noch fehlenden Stimmen. In einer Stufe mit 5 Knoten werden neben der Stimme des Arbeiters (die in diesem Fall vorausgesetzt werden kann, siehe oben) noch die Stimmen von mindestens zwei Beobachtern benötigt (also insgesamt drei Stimmen). Fehlen beide Beobachter-Stimmen bedeutet dies, daß von den 4 Beobachtern momentan keiner verfügbar ist. Fehlt nur eine Stimme, ist ein Beobachter verfügbar. Die Übergangsrate von “2 Stimmen fehlen” nach “1 Stimme fehlt” ist demnach die 4-fache Reparaturrate eines Knotens, die Gegenrichtung ist die Ausfallrate eines Knotens (des einen verfügbaren Beobachters). Der Übergang von “1 Stimme fehlt” zu “Votieren erfolgreich” setzt sich aus mehreren Komponenten zusammen. Für eine Stufe mit 5 Knoten ist dies zum einen die 3-fache Reparaturrate eines Knotens. Sind alle Knoten verfügbar muß jedoch erst die Aufforderung zum Votieren bei den Knoten eintreffen. Diese wird periodisch vom Orchestrator verschickt und trifft im Schnitt nach der halben Periodendauer bei den Votierern ein. Die Rate in

5 Knoten:



6 Knoten:



7 Knoten:

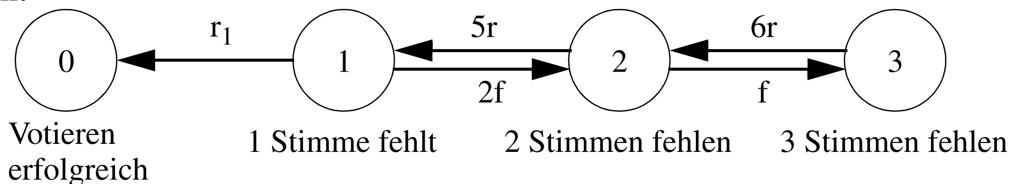


Abbildung 4-31. Modell des Votierens für Stufen mit 5, 6 und 7 Knoten bei Blockierung

den Zustand "Votieren erfolgreich" setzt sich bei einer Stufe mit fünf Knoten daher wie folgt zusammen:

$$r_1 = \frac{1}{\frac{T_{resend}}{2} + \frac{1}{3r}} \qquad (4\text{-}58)$$

wobei T_{resend} die Zeit zwischen dem Verschicken von 2 Votier-Aufforderungen ist. Im Modell wird hierbei vom ungünstigeren Fall ausgegangen, daß die Beobachter ihre Stimme erst dann abgeben, wenn wirklich genug Beobachter vorhanden sind.

Besitzt eine Stufe 6 oder 7 Knoten, so werden zusätzlich zur Stimme des Arbeiters noch 3 Stimmen von Beobachtern benötigt, wodurch diese Modelle einen Zustand mehr haben als das Modell für 5 Knoten. Obwohl beide Modelle dieselbe Anzahl von Zuständen besitzen unterscheiden sich die Übergangsraten zwischen den Zuständen.

Die mittlere Dauer für das Votieren im Fall des Blockierens ergibt sich, indem die mittlere Aufenthaltsdauer in der Zustandsmenge $\{1, 2, ..., \lceil (n-1)/2 \rceil\}$ berechnet wird. Da keine geschlossene Form für diese Aufenthaltsdauer gefunden wurde, wird im folgenden die Berechnung des Wertes anhand einer Stufe mit 5 Knoten demonstriert, Tabelle 4-3 zeigt die Werte für Stufen der Größe 2 bis 7.

Für eine Stufe mit 5 Knoten ergibt sich aus Abbildung 4-31 die folgende Übergangsmatrix

$$A_v = \begin{bmatrix} 0 & r_1 & 0 \\ 0 & -(r_1+f)_1 & 4r \\ 0 & f & -4r \end{bmatrix}$$

Die Laplace-Transformierte berechnet sich dann zu

$$\widehat{P}(s) = (s \cdot I - A_v)^{-1} \cdot P(0) = \begin{bmatrix} s & -r_1 & 0 \\ 0 & s + r_1 + f & -4r \\ 0 & -f & s + 4r \end{bmatrix}^{-1} \cdot P(0)$$

Die Anfangsbedingung *P(0)* erhält man aus den folgenden Überlegungen: Da das Modell nur für den Fall gilt, daß das Votieren tatsächlich blockiert, kann es sich am Anfang nicht im Zustand 0 (“Votieren erfolgreich”) befinden:

$$P_0(0) = 0 \tag{4-59}$$

Die Wahrscheinlichkeiten für die anderen Zustände sind bedingte Wahrscheinlichkeiten:

$$P_1(0) = P(\text{genau 1 Beobachter verfügbar} \mid \text{höchstens 1 von 4 Beobachtern verfügbar})$$
$$= \frac{P(\text{genau 1 Beobachter verfügbar})}{P(\text{höchstens 1 von 4 Beobachtern verfügbar})}$$

und

$$P_2(0) = P(\text{kein Beobachter verfügbar} \mid \text{höchstens 1 von 4 Beobachtern verfügbar})$$
$$= \frac{P(\text{kein Beobachter verfügbar})}{P(\text{höchstens 1 von 4 Beobachtern verfügbar})}$$

Mit Gleichung (4-38) und analog Gleichung (4-39) ergibt sich dann

$$P_1(0) = \frac{\binom{4}{1}\left(\frac{r}{f}\right)^1\left(\frac{f}{f+r}\right)^4}{\sum_{i=0}^{1}\binom{4}{i}\left(\frac{r}{f}\right)^i\left(\frac{f}{f+r}\right)^4} = 4\frac{r}{4r+f} \tag{4-60}$$

$$P_2(0) = \frac{\binom{4}{0}\left(\frac{r}{f}\right)^0\left(\frac{f}{f+r}\right)^4}{\sum_{i=0}^{1}\binom{4}{i}\left(\frac{r}{f}\right)^i\left(\frac{f}{f+r}\right)^4} = \frac{f}{4r+f} \tag{4-61}$$

Nach Berechnung der Inversen der Laplace-Transformierten und dem Einsetzen von P(0) kann man dann den Grenzwert von $\widehat{P}(s)$ für $s \to 0$ berechnen:

$$\lim_{s \to 0} \widehat{P}(s) = \begin{bmatrix} \infty \\ \frac{1}{r_1} \\ \frac{1}{4}\frac{f(4r + r_1 + f)}{r_1 r(f + 4r)} \end{bmatrix}$$

woraus sich dann direkt die gesuchte mittlere Aufenthaltsdauer T_{vb} in den Zuständen {1,2} ergibt:

$$T_{vb} = \sum_{i=1}^{2} \widehat{P}(0) = \frac{1}{r_1} + \frac{1}{4}\frac{f(4r + r_1 + f)}{r_1 r(f + 4r)} = \frac{1}{4}\frac{16r^2 + 8rf + fr_1 + f^2}{r_1 r(f + 4r)} \quad (4\text{-}62)$$

Mit Gleichung (4-58) erhält man schließlich

$$T_{vb} = \frac{32r^2 + 48r^3 T_{resend} + 22fr + 24r^2 fT_{resend} + 2f^2 + 3f^2 rT_{resend}}{24(f + 4r)r^2} \quad (4\text{-}63)$$

Die Werte für Stufengrößen von 2 bis 7 Knoten zeigt Tabelle 4-3.

Stufen-größe	T_{vb}
2	$\frac{2 + rT_{resend}}{2r}$
3	$\frac{1 + rT_{resend}}{2r}$
4	$\frac{9r^2 + 9r^3 T_{resend} + 8rf + 6r^2 fT_{resend} + f^2 + f^2 rT_{resend}}{6(f + 3r)r^2}$
5	$\frac{32r^2 + 48r^3 T_{resend} + 22fr + 24r^2 fT_{resend} + 2f^2 + 3f^2 rT_{resend}}{24(f + 4r)r^2}$

Tabelle 4-3. Mittlere Dauer der Blockierung des Votierens für verschiedene Stufengrößen

Stufen-größe	T_{vb}
6	$\frac{300r^4fT_{resend}+135r^3f^2T_{resend}+132r^2f^2+3f^4rT_{resend}+30r^2f^3T_{resend}}{60(r^3(f^2+5rf+10r^2))}+\frac{275r^3f+300r^5T_{resend}+200r^4+2f^4+23rf^3}{60(r^3(f^2+5rf+10r^2))}$
7	$\frac{360r^4fT_{resend}+132r^3f^2T_{resend}+100r^2f^2+2f^4rT_{resend}+24r^2f^3T_{resend}}{60(r^3(f^2+6rf+15r^2))}+\frac{252r^3f+450r^5T_{resend}+225r^4+f^4+14rf^3}{60(r^3(f^2+6rf+15r^2))}$

Tabelle 4-3. Mittlere Dauer der Blockierung des Votierens für verschiedene Stufengrößen

Nachdem nun alle Übergangsraten für den ersten Abschnitt des blockierungsfreien Protokolles bestimmt wurden, kann die Übergangsmatrix für das Modell aus Abbildung 4-30 aufgestellt werden:

$$A = \begin{bmatrix} -(f_{w1}+f_{w2}+w) & sel & 0 & 0 & 0 & 0 \\ f_{w2} & -(sel+f_d) & nr & nf & r & 0 \\ f_{w1} & f_d & -nr & 0 & 0 & 0 \\ w & 0 & 0 & -(nf+f+v_f) & 0 & 0 \\ 0 & 0 & 0 & f & -r & 0 \\ 0 & 0 & 0 & v_f & 0 & 0 \end{bmatrix} \tag{4-64}$$

Da die Knoten der Stufe die Commit-Entscheidung der Schritt-Transaktion der vorherigen Stufe sequentiell bekommen ist es möglich, daß der Transaktionskoordinator dieser Transaktion während dem Versenden der Commit-Entscheidung ausfällt und somit nur ein Teil der Knoten der Stufe den Agent schon aus ihrer Eingangswarteschlange lesen können. Da der Zustand “Selektion” auch schon den Fall beinhaltet, daß der Knoten mit der höchsten Priorität verfügbar ist, wird daher die (für den fehlerfreien Fall ungünstigere) Annahme getroffen, daß sich das System zu Beginn im Zustand “Selektion” befindet. Die Anfangsbedingung lautet demnach

$$P(0) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4-65}$$

Für die Bestimmung der Aufenthaltsdauer muß nun zuerst wieder die Laplace-Transformierte nach Gleichung (4-25) berechnet werden:

$$\widehat{P}(s) = \begin{bmatrix} s+f_{w1}+f_{w2}+w & -s_{el} & 0 & 0 & 0 & 0 \\ -f_{w2} & s+s_{el}+f_d & -nr & -nf & -r & 0 \\ -f_{w1} & -f_d & s+nr & 0 & 0 & 0 \\ -w & 0 & 0 & s+nf+f+v_f & 0 & 0 \\ 0 & 0 & 0 & -f & s+r & 0 \\ 0 & 0 & 0 & -v_f & 0 & s \end{bmatrix}^{-1} \cdot P(0)$$

Nach Berechnung der Inversen der Laplace-Transformierten und dem Einsetzen von P(0) kann man dann den Grenzwert von $\widehat{P}(s)$ für $s \to 0$ und daraus dann die mittlere Aufenthaltsdauer $T_{W,f}$ in der Zustandsmenge {0,1,2,3,4} berechnen:

$$\lim_{s \to 0} \widehat{P}(s) = \begin{bmatrix} \frac{v_f + (n+1)f}{v_f w} \\ \frac{(f_{w1}+f_{w2}+w)((n+1)f+v_f)}{v_f w s_{el}} \\ \frac{((f_{w1}+f_{w2}+w)f_d + f_{w1}s_{el})(v_f+(n+1)f)}{nrv_f w s_{sel}} \\ \frac{1}{v_f} \\ \frac{f}{rv_f} \\ \infty \end{bmatrix}$$

$$T_{W,f} = \sum_{i=0}^{4} \widehat{P}(0) = \frac{v_f + (n+1)f}{v_f w} + \frac{(f_{w1}+f_{w2}+w)((n+1)f+v_f)}{v_f w s_{el}} + \frac{1}{v_f} + \frac{f}{rv_f} + \frac{((f_{w1}+f_{w2}+w)f_d + f_{w1}s_{el})(v_f+(n+1)f)}{nrv_f w s_{el}}$$

mit Gleichung (4-51) und Gleichung (4-54) ergibt sich

$$T_{W,f} = \sum_{i=0}^{4} \widehat{P}(0) = \frac{(v_f+(n+1)f)((s_{el}+w+f)(nr+f_{w1}))}{nrv_f w s_{el}} + \frac{1}{v_f} + \frac{f}{rv_f} \tag{4-66}$$

Zweiter Abschnitt:

Im zweiten Abschnitt muß, wie auch beim Basisprotokoll, nur noch die zweite Phase des 2PC-Protokolles abgehandelt werden, genauer gesagt es muß die Commit-Entscheidung des Transaktionskoordinators an die beteiligten Ressourcen übermittelt werden (beim Abbruch der Transaktion würde der erste Abschnitt nicht verlassen). Beteiligte Ressourcen sind die lokale Eingangswarteschlange, der lokale Orchestrator, die Eingangswarteschlange der Knoten der nächsten Stufe und weitere lokale Ressourcen, auf die der Agent während der Schritt-Transaktion zugegriffen hat.

Ziel der gesamten Betrachtung ist zu berechnen, wie lange ein Agent in einer Stufe benötigt. Sobald irgend ein Knoten der nächsten Stufe die Commit-Entscheidung erhalten hat, beginnt für den Agent schon die nächste Stufe (im Zustand "Selektion"). Dies bedeutet für diesen Abschnitt zweierlei. Erstens wird beim Orchestrator die Zeit, die für Versand der *FORGET*-Nachrichten und Empfang der Bestätigungen benötigt wird, hier nicht eingerechnet, da die *FORGET*-Phase erst nach Zurücksenden der Bestätigung an den Transaktionskoordinator beginnt (und daher den restlichen Verlauf der zweiten Phase des 2PC-Protokolles nicht beeinflußt). Zweitens muß nur die Zeit eingerechnet werden, bis einer der Knoten der nächsten Stufe die Commit-Entscheidung erhalten hat.

Bei der Berechnung der Commit-Zeiten kann man auf die Ergebnisse aus Abschnitt 4.6.5.1 zurückgreifen. Geht man davon aus, daß der Orchestrator für die zweite Phase des 2PC-Protokolles nicht länger braucht als die lokale Eingangswarteschlange (die Zeit ist sogar eher kürzer, da, wie oben erwähnt, die *FORGET*-Phase hier nicht berücksichtigt werden muß) und daß die entfernten Ressourcen erst nach den lokalen Ressourcen die Commit-Entscheidung erhalten, dann berechnet sich die Zeit $T_{cl,f}$ für den Commit der lokalen Ressourcen zu

$$T_{cl,f} = 2T_{cl,b} = 2\frac{r+f}{c_l r} \tag{4-67}$$

Da die nächste Stufe bereits (im Zustand "Selektion") beginnt, nachdem der erste Knoten der nächsten Stufe die Commit-Entscheidung erfahren hat, ergibt sich

$$T_{cr,f} = T_{cr,b} = \frac{f(2f+c_r+2r)(f+2r)}{2r^2 c_r(r+f)} \tag{4-68}$$

Die mittlere Zeit $T_{C,f}$ für die zweite Phase berechnet sich also zu

$$T_{C,f} = T_{cl,f} + T_{cr,f} = 2\frac{r+f}{c_l r} + \frac{f(2f+c_r+2r)(f+2r)}{2r^2 c_r(r+f)} \tag{4-69}$$

Gesamtzeit:

Die mittlere Verweildauer $T_{V,f}$ eines Agenten in einer Stufe beim blockierungsfreien Protokoll ergibt sich dann aus der Summe von Gleichung (4-66) und Gleichung (4-69):

$$T_{V,f} = \frac{(v_f + (n+1)f)((s_{el} + w + f)(nr + f_{w1}))}{nrv_f w s_{el}} + \frac{1}{v_f} + \frac{f}{rv_f} + 2\frac{r+f}{c_l r} + \frac{f(2f + c_r + 2r)(f + 2r)}{2r^2 c_r (r+f)} \tag{4-70}$$

4.6.5.3 Vergleich der Protokolle

Um die beiden Protokolle vergleichen zu können wird zunächst das Verhältnis Q_V zwischen Verweildauer T_V in einer Stufe und der Ausführungszeit $1/e$ des Agenten berechnet:

$$Q_V = T_V / \frac{1}{e} = eT_V \tag{4-71}$$

Der Wert (Q_V - 1)*100 gibt dann den Mehraufwand in Prozent an, der durch das Protokoll, durch Knotenausfälle und durch die Mobilität des Agenten entsteht. Mit den Werten $Q_{V,b}=eT_{V,b}$ für das Basisprotokoll und $Q_{V,f} = eT_{V,f}$ für das blockierungsfreie Protokoll kann man dann den durch das blockierungsfreie Protokoll erreichten Gewinn G berechnen:

$$G = \frac{Q_{V,b} - 1}{Q_{V,f} - 1} \tag{4-72}$$

Der Gewinn G ist das Verhältnis von Mehraufwand Basisprotokoll zu Mehraufwand blockierungsfreies Protokoll. Ist der Gewinn G größer 1.0, so ist der Mehraufwand des blockierungsfreien Protokolles kleiner als der des Basisprotokolles. Ein Gewinn von G=2.0 bedeutet beispielsweise, daß der Mehraufwand des Basisprotokolles doppelt so hoch ist wie der des blockierungsfreien Protokolles.

Abbildung 4-32 zeigt Gewinn G des blockierungsfreien Protokolles in Abhängigkeit von der Verfügbarkeit der Knoten und der Anzahl der Knoten pro Stufe. Für die Berechnung des Gewinnes wurden die Parameter verwendet, welche sich aus den Messungen in Abschnitt 4.7 ergeben: Rate “Transaktionsstart” b=1/0.005 (5ms), Rate “Lesen aus Eingangswarteschlange” l=1/0.02 (20 ms), Rate “Schreiben in Eingangswarteschlange” s=1/0.04 (40ms), Rate “Prepare Eingangswarteschlange” p_s=1/0.07 (7ms), Rate “Commit lokal” c_l = 1/0.005 (5ms), Rate “Commit entfernt” c_r=1/0.007 (7ms), Rate “Selektion” s_{el}=1/.05 (50ms), Zeit zwischen wiederholtem Versenden von Votier-Aufforderungen T_{resend}=50ms und Dauer des Votierens im fehlerfreien

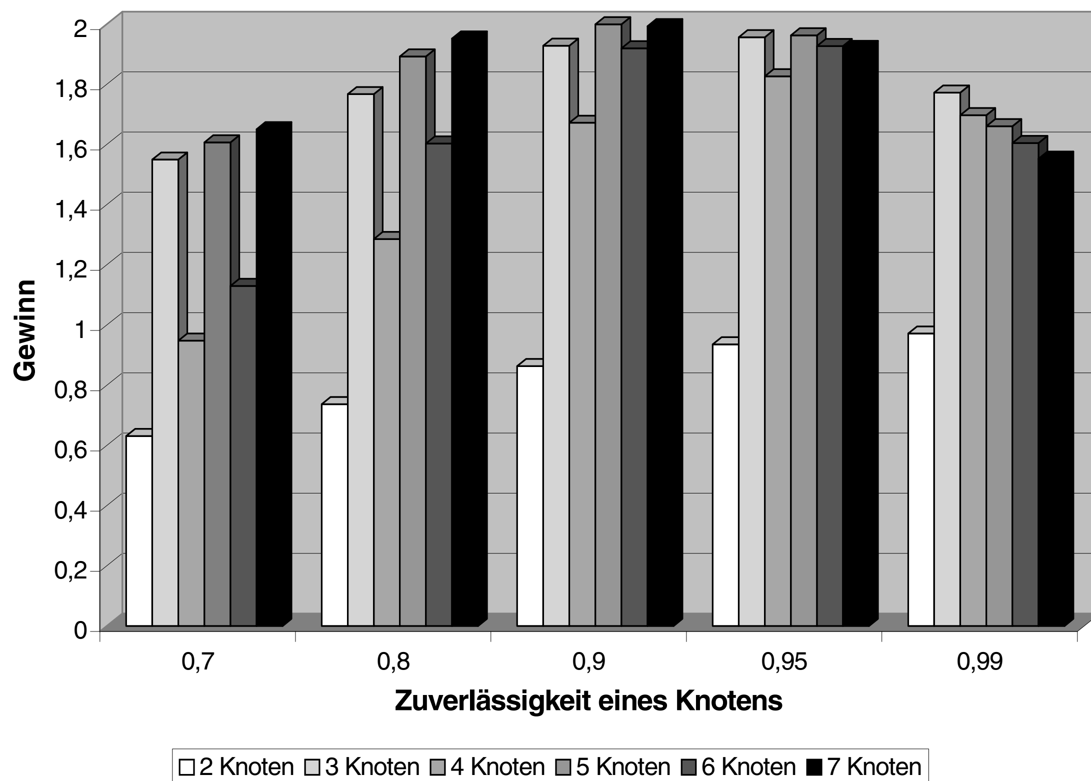


Abbildung 4-32. Gewinn des blockierungsfreien Protokolles in Abhängigkeit von der Verfügbarkeit eines Knotens und der Anzahl der Knoten pro Stufe

Fall T_{vn}=4ms. Für die Reparaturrate r wurde ein Wert von 1/120 gewählt, was einer mittleren Reparaturdauer von 2 Minuten entspricht, die Ausfallrate f berechnet sich dann aus der Knotenverfügbarkeit und der Reparaturrate. Die mittlere Ausführungszeit eines Agenten beträgt 128 Sekunden.

Anhand der Graphik ergeben sich mehrere Beobachtungen. Aus demselben Grund wie schon in Abschnitt 4.6.4 fällt auch bei der Verweildauer das Ergebnis für ungerade Knotenanzahlen n (zumindest teilweise) schlechter aus als das Ergebnis bei der jeweilig zugehörigen geraden Knotenzahl n-1. Allerdings zeigt die Abbildung auch, daß dies für höhere Knotenverfügbarkeiten nicht mehr gilt, ja daß mit steigender Knotenverfügbarkeit der Gewinn mit zunehmender Anzahl an Knoten pro Stufe sogar generell abnimmt. In Abbildung 4-32 beispielsweise ist bei einer Knotenverfügbarkeit von 0,99 der Gewinn für 3 Knoten maximal und nimmt dann mit jedem weiteren Knoten ab. Weiterhin zeigt sich, daß der Gewinn für einen gegebenen Satz an Eingangsparametern ab einer bestimmten Einzelknotenverfügbarkeit mit steigender Einzelknotenverfügbarkeit wieder abnimmt. In Abbildung 4-32 beispielsweise ist der Gewinn bei einer Verfügbarkeit von 0,99 niedriger als bei einer Verfügbarkeit von 0,95. Dieses Verhalten erklärt sich durch die folgenden Überlegungen: Durch die konstante Reparaturrate werden Fehler eines einzelnen Knotens mit steigender Verfügbarkeit eines Knotens sehr selten sodaß sich Knotenfehler zunehmend weniger auf den Erwartungswert der Verweildauer beim Basisprotokoll auswirken.

Da der durch das blockierungsfreie Protokoll eingeführte Mehraufwand (Verteilung auf mehrere Knoten, Votieren) nicht von der Verfügbarkeit eines Knotens sondern nur von der Anzahl der Knoten pro Stufe abhängt, wirkt sich dieser konstante Mehraufwand zunehmend negativ auf den Gewinn aus.

Der geringere Gewinn bei geringerer Einzelknotenverfügbarkeit ergibt sich vor allem dadurch, daß hier die Wahrscheinlichkeit einer Blockierung des Votierens höher ist und dadurch die Rate v_f relativ klein ausfällt (dies läßt sich zeigen, indem man bei der Berechnung von v_f eine konstante mittlere Zeit für das Votieren verwendet). Beide Faktoren zusammen bestimmen, wieviele Knoten pro Stufe bei gegebener Knotenverfügbarkeit den höchsten Gewinn ergeben.

Eine weitere Eigenschaft des blockierungsfreien Protokolles zeigt Abbildung 4-33. Die Abbildung zeigt den Gewinn abhängig von der Ausführungszeit des Agenten (in einer Stufe) und der Anzahl der Knoten pro Stufe. Die Parameter wurden wie oben gewählt, für die Verfügbarkeit eines einzelnen Knotens wurde ein Wert von 95% angenommen. Der Abbildung kann man entnehmen, daß sich nennenswerte Gewinne nur bei langen Ausführungszeiten eines Agenten in einer Stufe erreichen lassen. Wird die Zeit, die ein Agent zur Erledigung seiner Aufgabe auf einem Knoten braucht, sehr kurz, dann ist die mittlere Verweildauer in der Stufe beim blockierungsfreien Protokoll länger als beim Basisprotokoll. Analog zu oben läßt sich dies dadurch erklären, daß die Wahrscheinlichkeit eines Knotenausfalls während der Ausführung bei kürzeren Ausführungszeiten geringer ist und daher den Erwartungswert der Verweildauer beim Basispro-

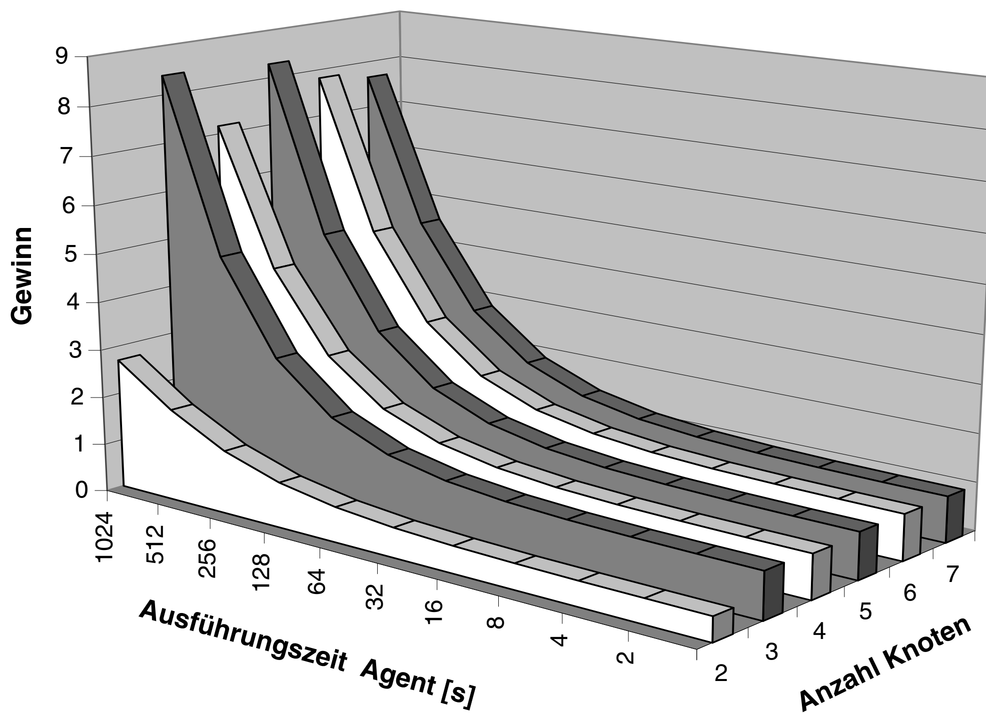


Abbildung 4-33. Gewinn des blockierungsfreien Protokolles in Abhängigkeit von der Ausführungszeit eines Agenten und der Anzahl der Knoten pro Stufe

tokoll weniger beeinflußt während der Protokoll-Mehraufwand beim blockierungsfreien Protokoll konstant ist.

Es zeigt sich also, daß mit dem blockierungsfreien Protokoll unter realen Bedingungen – Verfügbarkeit eines einzelnen Knotens mindestens 99% bei administrierten Systemen (vgl. z.B. GRAY UND REUTER (1993)) und Ausführungszeiten eines Agenten auf einem Knoten im Bereich von mehreren hundert Millisekunden bis wenigen Sekunden – ein Gewinn laut obiger Definition nicht zu erreichen ist, d.h. die mittlere Verweildauer in einer Stufe und damit die mittlere Ausführungszeit eines Agenten steigt durch die Verwendung des blockierungsfreien Protokolles.

Aus diesen Beobachtungen zu schließen, daß der Einsatz des blockierungsfreien Protokolles nutzlos ist, wäre allerdings falsch, da in diesem Abschnitt Mittelwerte betrachtet wurden. Ziel des Protokolles ist jedoch nicht, den Agenten im Mittel so schnell wie möglich auszuführen sondern den Agenten zuverlässig auszuführen und Blockierungen der Ausführung durch Systemfehler zu vermeiden. Abschnitt 4.4.4.2 zeigt, daß das Protokoll den Agenten zuverlässig ausführt und Abschnitt 4.6.4 zeigt, daß die Blockierwahrscheinlichkeit des Agenten durch das Protokoll drastisch verringert wird. Für Anwendungen, bei denen eine blockierungsfreie, zuverlässige Ausführung des Agenten notwendig ist, ist die Verwendung des Protokolles – auf Kosten der durchschnittlichen Ausführungszeit – unumgänglich. Ist für eine Anwendung jedoch eine zuverlässige, im Durchschnitt möglichst schnelle Ausführung notwendig, bei der auch seltene, längerandauernde Blockierungen toleriert werden können, dann empfiehlt sich für diese Anwendung die Verwendung des Basisprotokolles.

4.7 Leistungsmessungen

In diesem Abschnitt wird untersucht, wieviel Mehraufwand durch den Einsatz des in diesem Kapitel entwickelten blockierungsfreien Protokolles entsteht. Hierzu wird zuerst ein sehr grober Überblick über die Protokollimplementation gegeben, der vor allem die Verteilung der Komponenten des Protokolles auf verschiedene Prozesse und die Kommunikationsmechanismen zwischen diesen Prozessen beschreibt. Nach Einführung der Messmethodik werden die Messergebnisse präsentiert und diskutiert.

4.7.1 Protokollimplementation

Für die Implementation des Protokolles wurde der am Institut verwendete CORBA-ORB von Iona und dessen Transaktionsservice verwendet. Dieser Transaktionsservice erlaubt in der verwendeten Version 1.0 die Implementation von Servern lediglich in C++, Klienten dieser transaktionalen Server können sowohl in C++ als auch in Java implementiert werden. Da das Protokoll in das am Institut implementierte Agentensystem Mole integriert werden sollte, welches in

Java implementiert wurde, war eine Trennung von in C++ implementierten Teilen (transaktionale Ressourcen) und in Java implementierten Teilen unumgänglich.

Eine erste Implementation des Protokolles erfolgte durch FRIEDEL (1998), eine effiziente transaktionale Nachrichtenwarteschlange wurde von MESSNER (1999) realisiert, der Algorithmus zur Stufenkonstruktion und die Integration des Protokolles wurden von PAPOULIDIS (1999) vorbereitet. Um klarere Schnittstellen zwischen den in Java und C++ implementierten Teilen des Protokolles zu erhalten und die Leistung zu erhöhen, wurde die Protokollimplementation nochmals komplett neu gemacht. Einen groben Überblick über die resultierende Architektur gibt Abbildung 4-34.

Der Orchestrator und die Nachrichtenwarteschlange (Message Queue) sind in C++ implementiert und werden auf jedem Knoten in jeweils einem eigenen Prozeß ausgeführt. Der Rest des Protokolles ist in Java implementiert und wird zusammen mit der Agenten-Ausführungsumgebung Mole in einer Java Virtual Machine (in einem Prozeß) ausgeführt. CORBA wird nur dann zur Kommunikation verwendet, wenn dies zum Austausch von Transaktionskontext-Informationen notwendig ist, beschränkt sich also auf das Lesen von Agenten aus bzw. das Schreiben von Agenten in die Nachrichtenwarteschlange und auf die Registrierung der Schritt-Transaktion beim Orchestrator. Alle andere Kommunikation wird aus Effizienzgründen mittels *UDP* (*User Datagram Protocol*) erledigt. Die Implementierung des blockierungsfreien Protokolles enthält als Spezialfall das Basisprotokoll.

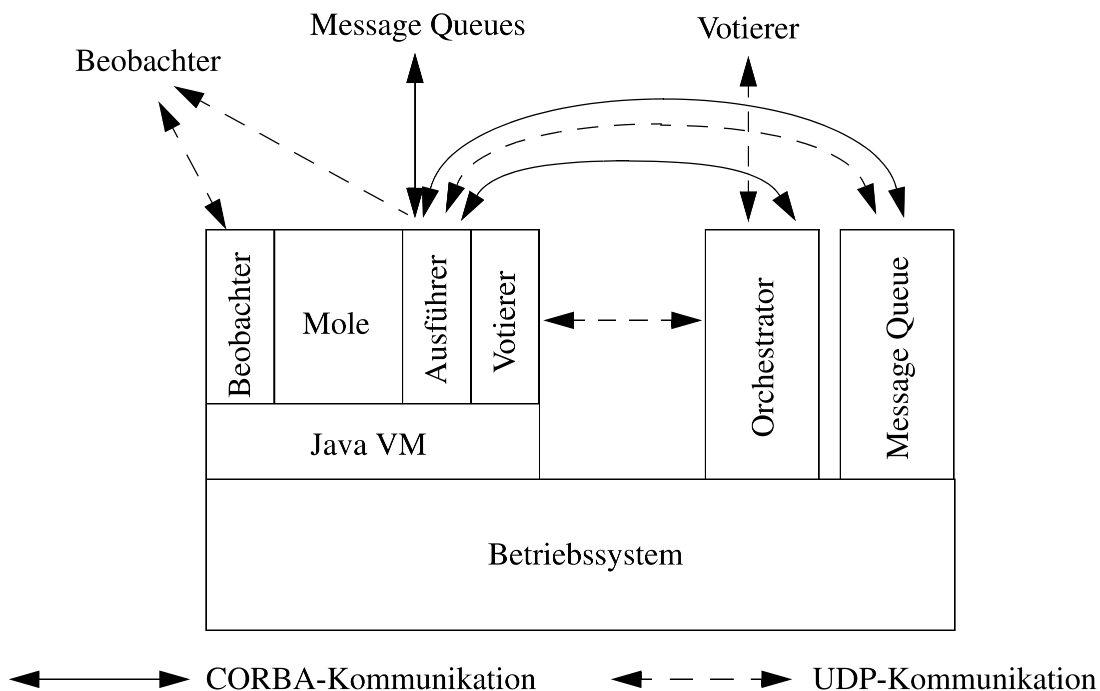


Abbildung 4-34. Architektur der Implementierung

4.7.2 Messungen

Ziel der Messungen ist es, eine Aussage über den durch das blockierungsfreie Protokoll erzeugten Mehraufwand bei fehlerfreiem System (keine Ausfälle) zu treffen und zu untersuchen, wie sich dieser auf die verschiedenen Komponenten des Protokolles verteilt. Für die Messungen wurde der Code des Protokolles instrumentiert, um Messwerte über die zur Verarbeitung eines Agenten in einer Stufe notwendigen Aktionen zu erhalten.

Für die Messungen wurde ein Meß-Agent geschrieben, welcher zwischen zwei Knoten insgesamt 50 Migrationen durchführt und hierbei insgesamt 51 Schritte ausführt – 26 Schritte auf dem Knoten, auf dem der Agent gestartet wurde und 25 Schritte auf dem anderen Knoten. Um die Transportgröße des Agenten während der Ausführungszeit des Agenten konstant auf 12KB zu halten, enthält jeder Schritt einige wenige (kurz laufende) Code-Anweisungen, die für einen Ausgleich der Transportgröße sorgen. Dieser Ausgleich ist notwendig, da sich durch die Verwendung des von BUSCHLE (1999) implementierten Reiseroutenalgorithmus die Größe des Agenten bei jeder Migration ändert, selbst wenn sonst keinerlei Änderungen an den Daten vorgenommen werden. Zusätzlich ermittelt der Agent bei der Ausführung des ersten Schrittes die lokale Zeit T_{start}, bei der Ausführung des letzten Schrittes (der ebenfalls auf dem Startknoten ausgeführt wird) wird die lokale Zeit T_{stop} genommen. Ansonsten enthalten die Schritte keinen zusätzlichen Code. Die durchschnittliche Zeit zur Ausführung einer Stufe ergibt sich dann durch (T_{stop}-T_{start})/50. Die Division durch 50 und nicht durch 51 ergibt sich deshalb, weil die Zeiten während des ersten und des letzten Schrittes genommen wurden und nicht vor dem ersten Schritt und nach dem letzten Schritt

Während der gesamten Ausführung des Meß-Agenten werden durch den instrumentierten Code des Protokolles jeweils auf dem Start-Knoten die Zeiten der einzelnen Aktionen des Protokolles gemessen und nach vollständiger Ausführung der *statistische Mittelwert* und die *statistische Standardabweichung* berechnet. Um hierbei die erhaltenen Werte nicht zu verfälschen, werden die Werte des letzten ausgeführten Schrittes nicht mehr berücksichtigt, da hier keine weitere Migration durchgeführt wird.

Die Messungen wurden auf 5 Ultra10-Rechnern (440MHz CPU-Taktfrequenz, 256 MB Hauptspeicher) in einem lokalen Netzwerk durchgeführt. Der Start-Knoten und die drei Rechner, welche nur als Beobachter zum Einsatz kamen, sind mit einem 100Mb-Ethernet (über einen Switch) verbunden. Der Knoten, welcher neben dem Start-Knoten ebenfalls zur Ausführung des Meß-Agenten herangezogen wurde, ist über einen Hub mit einem 10Mb-Ethernet an den Switch angeschlossen, über den die restlichen 4 Knoten verbunden sind. Mit dem Programm *ping* gemessene *Rundlaufzeiten* (engl.: *round-trip-time*) liegen zwischen allen Rechnern unter 1ms. Sowohl die Rechner als auch das Netzwerk waren während den Messungen anderweitig nur unwesentlich (durch Systemprozesse und -nachrichten) belastet.

Die Messungen wurden so angelegt, daß der Agent die 51 Schritte jeweils mit Stufengrößen von

1 Knoten (Basisprotokoll), 2 Knoten, 3 Knoten, 4 Knoten und 5 Knoten ausführt. Diese Messungen wurden zur Kontrolle insgesamt 3 mal ausgeführt.

Die durch die Messungen mit dem Meß-Agenten erhaltenen durchschnittlichen Ausführungszeiten für eine Stufe/einen Schritt zeigen Tabelle 4-4 und Abbildung 4-35

Nr. Knoten	1	2	3	4	5
T [ms]	206	269	347	411	497

Tabelle 4-4. Durchschnittliche Ausführungszeit pro Stufe

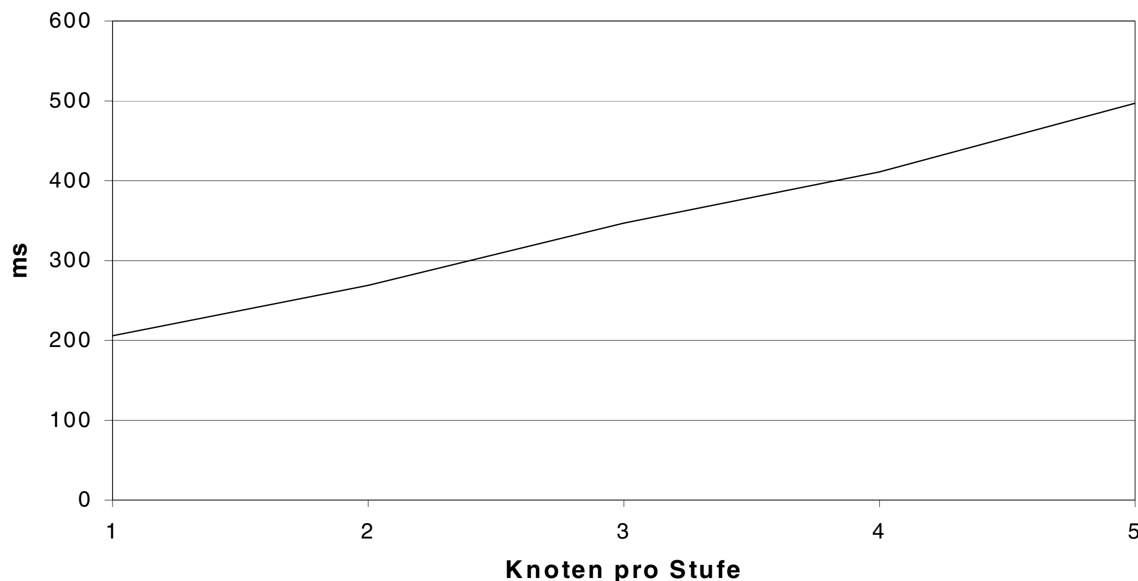


Abbildung 4-35. Durchschnittliche Ausführungszeit pro Stufe

Es zeigt sich, daß die Ausführungszeiten annähernd linear mit der Anzahl der Knoten pro Stufe ansteigen, wobei die Zunahme pro Knoten (zwischen 63ms und 86 ms) sehr deutlich unter der Ausführungszeit beim Basisprotokoll (206ms) ist, d.h. doppelte Anzahl Knoten bedeutet nicht doppelte Zeit. Wie sich diese Zeiten zusammensetzen zeigen Tabelle 4-5 und Abbildung 4-36.

Es zeigt sich, daß bei allen Phasen des Protokolles, in denen der Mehraufwand von der Anzahl der Knoten in der Stufe abhängig ist, die Ausführungszeiten annähernd linear mit der Anzahl der Knoten pro Stufe anwachsen. In der Phase "Ausführen & Stufenkonstruktion" geht der Zuwachs alleine auf das Konto des Stufenkonstruktionsalgorithmus, hält sich jedoch mit rund 7-10ms pro zusätzlichem Knoten in Grenzen. Der stärkste Zuwachs erfolgt durch das Schreiben des Agenten in die Eingangswarteschlangen der Knoten der nachfolgenden Stufe. Dies ist vor allem darauf zurückzuführen, daß bei der Protokollimplementation auf Portabilität auf andere Transaktions-Middleware geachtet wurde und deshalb das Schreiben des Agenten in die Eingangswarteschlangen der nächsten Stufe seriell erfolgt (nicht jedes Transaktionssystem erlaubt parallelen Zugriff auf verschiedene Ressourcen innerhalb derselben Transaktion). Messungen in der Arbeit von FRIEDEL (1998) zeigen, daß durch paralleles Schreiben in die Eingangswarteschlangen der nächsten Stufe diese Zeit nochmals drastisch reduziert werden kann. Die für das

Anzahl Knoten	1	2	3	4	5
Beginn Transaktion	4.6 (0.74)	4.64 (0.67)	5.52 (1.43)	5.82 (0.58)	7.52 (9.09)
Lesen und Deserialisieren Agent	71.56 (11.20)	67.92 (7.31)	67.92 (5.95)	68.24 (4.37)	72.64 (11.56)
Ausführen & Stufenkonstruktion	11.16 (7.92)	18.64 (11.57)	25.00 (14.47)	35.88 (18.94)	42.44 (24.97)
Serialisieren und Schreiben Agent	65.48 (29.19)	98.88 (40.60)	148.68 (84.75)	183.8 (89.32)	235.52 (126.62)
Registrierung Orchestrator	0	7.84 (10.07)	5.92 (3.38)	5.96 (3.58)	6.16 (3.38)
Commit Transaktion	27.12 (3.49)	39.36 (6.15)	52.56 (7.62)	64.96 (8.77)	81.84 (13.88)

Tabelle 4-5. Ausführungszeiten in ms (Standardabweichungen in ms) der verschiedenen Abschnitte bei der Ausführung einer Stufe

Commit der Transaktion benötigte Zeit steigt ebenfalls nur leicht mit zunehmender Anzahl an Knoten pro Stufe (ca. 12ms pro zusätzlichem Knoten). Messungen beim Orchestrator haben gezeigt, daß dieser für das Votieren unabhängig von der Anzahl der Knoten pro Stufe zwischen 3ms und 4 ms benötigt. Der zusätzliche Zeitaufwand beim Commit ist also alleine darauf zurückzuführen, daß zusätzliche Eingangswarteschlangen im 2PC-Protokoll teilnehmen.

Auffällig an den Werten in Tabelle 4-5 sind die teilweise sehr hohen statistischen Standardab-

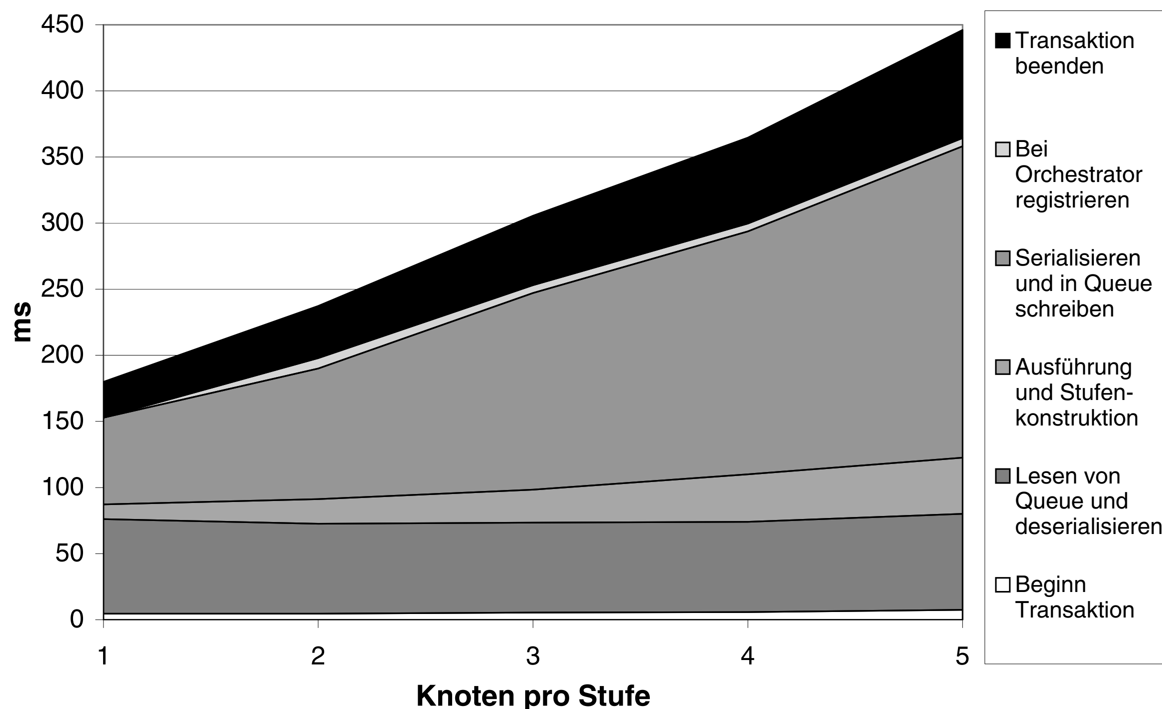


Abbildung 4-36. Ausführungszeiten der verschiedenen Abschnitte bei der Ausführung einer Stufe

weichungen. Diese sind aber leicht nachvollziehbar, wenn man bedenkt, daß die Messungen in einem Multitaskingsystem und über ein Netzwerk hinweg erfolgen und daher einzelne Messwerte je nach aktuellem System- und Netzwerkzustand stark voneinander abweichen können.

Zusammenfassend kann man sagen, daß die Messungen gezeigt haben, daß der Mehraufwand annähernd linear zur Anzahl der Knoten zunimmt:

$$\text{Zeit für Stufe mit n Knoten} \approx \text{Zeit für Basisprotokoll} + (n\text{-}1)(\text{Zeit für Stufe mit 2 Knoten} - \text{Zeit für Basisprotokoll})$$

Diese lineare Zunahme der Zeit läßt sich bei allen Phasen des Protokolles, bei denen der Mehraufwand von der Anzahl der Knoten abhängig ist, beobachten. Einen wesentlichen Beitrag zur benötigten Zeit steuert die Serialisierung des Agenten und (vor allem) der Transport des serialisierten Agenten in die Warteschlangen der nachfolgenden Stufe bei. Die für den Transport des Agenten in die nächste Stufe benötigte Zeit kann in einer optimierten Implementation durch paralleles Schreiben in die Warteschlangen der nachfolgenden Stufe reduziert werden.

4.8 Verwandte Arbeiten

Im Bereich der mobilen Agenten existieren bisher nur wenige verwandte Arbeiten. In den klassischen Bereichen der Transaktionsverarbeitung und Fehlertoleranz hingegen gibt es eine Vielzahl unterschiedlichster Ansätze zur blockierungsfreien bzw. genau-einmal Ausführung von Programmen. Im folgenden werden zuerst die klassischen Ansätze dargestellt.

4.8.1 Bereiche Transaktionsverarbeitung und Fehlertoleranz

In den Bereichen der Transaktionsverarbeitung und der Fehlertoleranz reicht die Spanne der Ansätze vom Aufbau fehlertoleranter Hardware bis zur replizierten Programmausführung auf mehreren Rechnern.

Die Konstruktion fehlertoleranter Hardware setzt schon auf unterster Ebene im Bereich der Schaltkreise an. Die hier verwendeten Mechanismen haben zu dieser Arbeit keinen wesentlichen Bezug, deshalb sei hierfür auf eine Zusammenfassung in JALOTE (1994) verwiesen. Ein auf höherer Ebene verwendetes Konzept ist das der schon erwähnten *N modular redundancy*. Idee hier ist, ein (beliebiges) Modul *n*-fach auszulegen. Die replizierten Module bekommen dieselben Eingaben und die Ausgaben dieser Module werden über (einen oder mehrere) Votierer zu einer Ausgabe (welche der Mehrzahl der produzierten Ausgaben entspricht) zusammengefasst. Hiermit lassen sich sowohl transiente als auch permanente Hardwaredefekte in einem Modul maskieren. Eine Abwandlung dieses Konzepts, das *N version programming*, dient dazu, Designfehler auf der Softwareebene zu maskieren. Das Problem der Fehlertoleranz gegen Fehler

im Design liegt nicht im Blickpunkt dieser Arbeit liegt und wird deshalb hier nicht weiter ausgeführt.

Im Bereich der Client/Server-Interaktion stellte SPECTOR (1982) schon sehr früh verschiedene Fehlersemantiken des entfernten Prozeduraufrufs (engl.: remote procedure call, kurz: *RPC*) vor. Die hierin definierte *nur-einmal-Typ-2* Ausführung (engl.: only-once-type-2) entspricht laut SCHILL (1992A) der später weit verbreiteten Definition der *genau-einmal* Ausführung (engl.: *exactly-once*). Diese Fehlersemantik garantiert, daß ein Aufruf einer Server-Prozedur auch bei Ausfall und Wiederanlauf des Client- oder Server-Rechners genau einmal durchgeführt wird bzw. daß das System das Resultat genau einer einzigen Ausführung der Prozedur widerspiegelt. Sowohl SPECTOR (1982) als auch SCHILL (1992B) skizzieren Möglichkeiten zur Implementierung dieser Fehlersemantik – unter anderem mittels Transaktionen.

ACID-Transaktionen (vgl. GRAY UND REUTER (1993) für eine Übersicht) sind ein Konzept im Bereich der Datenbanken und der fehlertoleranten Systeme. Sie stellen sicher, daß eine Menge von Operationen atomar ausgeführt wird. Atomare Ausführung bedeutet einerseits, daß entweder alle Operationen erfolgreich ausgeführt werden oder bei einem Abbruch der Transaktion keinerlei Effekte der in der Transaktion ausgeführten Operationen sichtbar werden. Andererseits bedeutet atomar auch, daß die Ausführung der Operationen, von außerhalb der Transaktion betrachtet, wie die Ausführung einer einzelnen Operation aussehen - Zwischenzustände sind nicht sichtbar. In einer Datenbank oder in anderen Ressourcen gespeicherte Ergebnisse von erfolgreich abgeschlossenen Transaktionen sind dauerhaft und gehen selbst bei einem Systemausfall nicht verloren. Der Programmzustand einer eine Transaktion ausführenden Applikation wird hingegen bei gängigen Transaktionssystemen bei Transaktionsende nicht auf stabilen Speicher geschrieben (und geht daher bei einem Rechnerausfall verloren). Daher realisiert das Konzept der Transaktionen zwar von sich aus noch keine genau-einmal Ausführung, es ist jedoch bei vielen Mechanismen zur fehlertoleranten Ausführung – auch bei dem in dieser Arbeit vorgestellten – ein essentieller Bestandteil.

Einen großen Schritt in Richtung der genau-einmal Ausführung von langandauernden Transaktionen geht das in GARCIA-MOLINA UND SALEM (1987) vorgestellte Transaktionsmodell für langandauernde Aktivitäten, die *Sagas*, auf dem auch der in dieser Arbeit vorgestellte Ansatz basiert. Um eine langfristige Blockierung von Ressourcen durch eine Transaktion zu verhindern, wird die langandauernde Aktivität in mehrere Teile, die sogenannten *Schritte* (engl.: *steps*), zerlegt. Jeder Schritt wird innerhalb einer ACID-Transaktion ausgeführt. Das Commitment eines Schrittes beginnt automatisch eine Transaktion für den nächsten Schritt. Für jeden Schritt muß ein Kompensationsschritt spezifiziert werden, der die Auswirkungen eines Schrittes rückgängig machen kann. Die Laufzeitumgebung einer Saga garantiert, daß letztendlich entweder alle Schritte einer Saga erfolgreich beendet werden oder die Saga abgebrochen wird. Bricht ein Schritt ab, wird *Backward-Recovery* durchgeführt, indem für die schon erfolgreich beendeten Schritte die Kompensations-Schritte ausgeführt werden, wobei der abgebrochene Schritt die für einen Transaktionsabbruch übliche Recovery erfährt. Damit nach einem Fehler nicht die

ganze bisher geleistete Arbeit verloren geht, können von der Anwendung zwischen Schritten Rücksetzpunkte geschrieben werden, auf die bei den genannten Fehlerfällen mittels Backward-Recovery zurückgesetzt werden kann und bei denen dann die Saga fortgesetzt werden kann (*Forward-Recovery*). Bei explizitem Abbruch der Saga durch die Anwendung werden natürlich immer alle bisher erfolgreich beendeten Schritte kompensiert. Die in GARCIA-MOLINA ET AL. (1991) vorgestellten Erweiterungen wie *geschachtelte Sagas* (engl.: *nested sagas*) und *non-vital Sub-Sagas* ermöglichen eine flexiblere Definition des Aufbaus einer Saga. Vor allem durch die Möglichkeit Rücksetzpunkte zu setzen realisiert das Saga-Konzept die genau-einmal Ausführung einer Anwendung (sofern die Anwendung Rücksetzpunkte setzt). Allerdings bietet das Saga-Konzept keinerlei Vorkehrungen, die auf einem ausgefallenen Rechner zum Ausfallzeitpunkt ausgeführten Anwendungen auf einem anderen Rechner neu aufzusetzen, d.h. die Anwendungen sind so lange blockiert, bis der ausgefallene Rechner neu startet und die *Recovery* für die laufenden Sagas durchführt.

Zwei Konzepte zur fehlertoleranten Implementierung von Servern, die nach einem Server-Ausfall möglichst schnell wieder zur Verfügung stehen sollen, sind das *Warm-Backup* (*warmes Backup*) und das *Hot-Backup* (*heißes Backup*). BERNSTEIN UND NEWCOMER (1997) bieten einen guten Überblick über die Konzepte. Bei beiden Konzepten gibt es neben dem eigentlichen Server, genannt *Primär-Server* (engl.: *primary server*), noch den *Backup-Server.* Solange keine Fehler auftreten ist der Primär-Server für die Bearbeitung der Client-Anfragen zuständig. Beim Warm-Backup überwacht der Backup-Server nur den Primär-Server. Sobald der Primär-Server ausfällt führt der Backup-Server anhand des Logs Recovery durch, d.h. er stellt bei sich einen Zustand her der dem letzten konsistenten und ins Log geschriebenen Zustand des Primär-Servers entspricht. Anschließend übernimmt der Backup-Server die Aufgaben des Primärservers. Beim Hot-Backup – auch *Prozeßpaare* (engl.: *process pairs*) genannt, vgl. GRAY UND REUTER (1993) – wird der Zustand des Backup-Servers konstant mit dem Zustand des Primär-Servers konsistent gehalten. Dies geschieht entweder, indem der Backup-Server laufend die neuen Log-Einträge des Primär-Servers mitliest und damit seinen Zustand aktualisiert (andauerndes Recovery), oder indem die Client-Anfragen auch an den Backup-Server geschickt wird und dieser die Anfragen ebenfalls bearbeitet, aber keine Antworten an den Client schickt. Sowohl Warm-Backup als auch Hot-Backup setzen absolut zuverlässige Kommunikation (ohne Ausfall) und daher physikalische Nähe zwischen Primär- und Backup-Server voraus. Vorteil des Warm-Backup ist, daß ein Server als Backup für mehrere Primär-Server verwendet werden kann, da er nur im Falle eines Ausfalles eines Primärservers dessen Arbeit übernehmen muß. Um gegen den gleichzeitigen Ausfall mehrerer Primär-Server gewappnet zu sein sind entsprechend mehr Backup-Server vorzusehen. Es entsteht also ein relativ geringer Mehraufwand. Nachteil des Warm-Backup ist, daß der Backup-Server nach Ausfall des Primär-Servers erst einen konsistenten Zustand herstellen muß und er daher die Bearbeitung von Client-Anfragen nur mit Verzögerung aufnehmen kann. Beim Hot-Backup hingegen wird, vorausgesetzt daß die Primär-Server ausgelastet sind, für jeden Primär-Server ein Backup-Server benötigt, da der

Backup-Server andauernd damit beschäftigt ist, auf dem gleichen Stand zu sein wie der Primär-Server. Der hierdurch eingeführte Mehraufwand ist erheblich. Allerdings kann hier der Backup-Server unverzüglich die Arbeit der Primär-Servers bei dessen Ausfall übernehmen. Bei der Entscheidung Warm-Backup versus Hot-Backup gilt es folglich abzuwägen, ob die schnelle Übernahme der Arbeit den durch Hot-Backup eingeführten Mehraufwand für den Einsatzzweck rechtfertigt. Warm-Backup und Hot-Backup führen jedoch nur Fehlertoleranz für den Server ein, eine fehlertolerante Ausführung der Client-Anwendungen ist hiermit nicht gegeben.

Das Prinzip des Hot-Backup findet man in Erweiterungen in einigen Arbeiten vor. YAP, JALOTE UND TRIPATHI (1988) stellen einen fehlertoleranten RPC vor, bei dem die Anfrage der Clients an einen Primär-Server und *mehrere* Backup-Server verschickt wird. Alle Server bearbeiten die Anfrage, nur der Primär-Server schickt die Antwort an den Client. Fällt der Primär-Server aus, übernimmt einer der Backup-Server. Eine Übertragung dieses Prinzips auf das modernere Programmierparadigma der objektorientierten Programmierung führen BEEDUBAIL ET AL. (1995) durch. Auch hier hat jedes Primär-Objekt mehrere Backup-Objekte. Die Methodenaufrufe auf das Primär-Objekt werden gleichzeitig an die Backup-Objekte verteilt und von diesen bearbeitet. Auch hier ist eine fehlertolerante Ausführung der Client-Anwendungen nicht gegeben.

Eine für den Anwendungsprogrammierer transparente fehlertolerante Ausführung der Anwendung präsentiert der auf dem Konzept des Hot-Backup basierende Ansatz von BRESSOUD (1998). Die Ausführung einer Anwendung erfolgt repliziert auf mehreren Rechnern, wobei eine Ausführungsinstanz die Primär-Instanz ist, die anderen Ausführungsinstanzen sind Backup-Instanzen. Die transparente Replikation wird erreicht, indem einerseits eine Zwischenschicht zwischen Applikation und Betriebssystem eingefügt wird und andererseits transparente Änderungen am Code der Anwendung zur Laufzeit durchgeführt werden. Die Zwischenschicht zwischen Applikation und Betriebssystem bietet der Applikation dabei dieselbe Schnittstelle wie das Betriebssystem. Die Zwischenschicht und die Codemodifikationen garantieren, daß Betriebssystemaufrufe und die Auslieferung von *Ausnahmen* (engl.: *exceptions*) auf allen Replikaten in derselben Reihenfolge ausgeführt werden, wobei nichtdeterministische Betriebssystemaufrufe (z.B. Uhrzeit auslesen) auf der Primärinstanz ausgeführt und die Ergebnisse den Replikaten zugeschickt werden. Fällt die Primärinstanz aus, wird eines der anderen Replikate zur Primärinstanz gewählt. Neben dem bei Hot-Backup typischen hohen Mehraufwand hat der Ansatz noch die Nachteile, daß die Transparenz im Fehlerfalle nicht garantiert wird und daß der Mechanismus keine Netzwerkpartitionierungen toleriert.

WÄCHTER UND REUTER (1992) und REUTER, SCHNEIDER UND SCHWENKREIS (1997) stellen mit dem *ConTract* Modell einen dem in dieser Arbeit vorgestellten Ansatz im Ziel sehr ähnlichen Ansatz vor. Das ConTract Modell hat ebenfalls die fehlertolerante genau-einmal Ausführung langandauernder Berechnungen (z.B. Workflows) in einer verteilten Umgebung zum Ziel. Eine Anwendung besteht aus einer Menge von Schritten, deren Ablaufreihenfolge sehr flexibel durch ein *Script* definiert wird. Im Gegensatz zu unserem Ansatz ist hier auch die parallele Abarbeitung von Schritten möglich. Ein Schritt selbst ist ein Programm das einen Teil der Anwendungs-

logik implementiert. Es wird im allgemeinen in einer Transaktion ausgeführt. Der Zustand eines Scripts, welcher den Gesamtzustand der Anwendung repräsentiert, wird in persistenten Variablen, den sogenannten *Kontext-Variablen*, gehalten. Dies ermöglicht den einfachen Neustart eines Schrittes im Falle eines (System-)Fehlers während der Ausführung des Schrittes. Schwerpunkt der Forschung beim ConTract Modell ist vor allem die Wahrung der Konsistenz bei der Ausführung eines Scripts und weniger die Problematiken der Tolerierung von Netzwerkpartitionierungen oder der Weiterführung einer Anwendung bzw. eines Scripts auf anderen Rechnern im Falle eines Rechnerausfalles. Beispielsweise wird im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz die Koordination der Ausführung eines Scripts von einer stationären ConTract Engine ausgeführt, welche die Ausführung des Scriptes bei Rechnerausfall erst nach Neustart des Rechners fortsetzt. Einen Überblick über eine prototypische Implementation bieten SCHNEIDER (1997B) und SCHNEIDER (1998).

4.8.2 Mobile Agenten

Bei den Ansätzen aus dem Bereich der mobilen Agenten reicht die Spanne von Algorithmen, die nur einzelne Teile der Ausführung mobiler Agenten, beispielsweise die Migration, mit mehr Fehlertoleranz versehen bis zu Algorithmen zur Erkennung bzw. Vermeidung byzantinischer Fehler bei der Ausführung der Agenten. Alle Ansätze bis auf den zuletzt vorgestellten berücksichtigen bei Knoten nur Crash-Fehler, Netzwerkfehler bzw. die Partitionierung von Netzwerken werden oft nicht toleriert.

Ein Ansatz der vor allem die fehlerfreie Migration von Agenten sicherstellt wird sowohl in VOGLER, KUNKELMANN UND MOSCHGATH (1997A) als auch in VOGLER, KUNKELMANN UND MOSCHGATH (1997B) beschrieben. Hierbei wird die Migration eines Agenten – ähnlich wie in dem in dieser Arbeit entwickelten Ansatz – innerhalb einer verteilten Transaktion durchgeführt. Dies stellt sicher, daß der Agent entweder erfolgreich auf dem Zielrechner der Migration ankommt oder noch auf dem Ursprungsrechner der Migration ist, d.h. daß der Agent bei der Migration nicht verloren gehen kann. Bricht ein Rechner wegen eines Fehlers zusammen, so wird ein auf diesem Rechner zum Zeitpunkt des Zusammenbruchs ausgeführter Agent nach Neustart des Rechners erneut gestartet – im selben Zustand, in dem der Agent direkt nach seiner Ankunft auf dem Rechner gestartet wurde. Da keine weiteren Maßnahmen getroffen werden, führt der Agent bei einem Neustart die vor dem Zusammenbruch des Rechners ausgeführten Operationen erneut aus. Gibt der Agent beispielsweise vor dem Rechnerausfall eine Bestellung von Waren auf, so macht er dies nach dem Neustart erneut und hat somit zwei Bestellungen aufgegeben, obwohl nur eine gewünscht ist. Es ist hier also nicht sichergestellt, daß ein Agent die von ihm zu erledigenden Arbeiten nur einmal ausführt. Ähnliche Ansätze werden in DALMEIJER ET AL. (1998) und WALSH, PACIOREK UND WONG (1999) beschrieben. Hier haben die Agenten allerdings die Möglichkeit, *Rücksetzpunkte* (engl.: *savepoints*) zu schreiben und nach einem Neustart beim aktuellsten Rücksetzpunkt mit der Ausführung zu beginnen. Bei entsprechender

(aufwendiger) Programmierung der Agenten ermöglicht dies eine genau-einmal Ausführung der Agenten. Defizit aller dieser Verfahren ist, daß die Ausführung eines Agenten blockiert bleibt, solange sein momentaner Aufenthaltsort (d.h. der ihn momentan ausführende Rechner) wegen eines Fehlers oder einer Rechnerabschaltung außer Betrieb ist. Ist der ausführende Rechner wegen eines Netzwerkfehlers vom Netzwerk abgeschnitten, kann der Agent erst nach Behebung des Netzwerkfehlers auf einen anderen Rechner migrieren.

Einen Ansatz, in dem die fehlertolerante Ausführung mobiler Agenten durch den Einsatz eines hochverfügbaren Systems (Tandem Himalaya Server) erreicht wird beschreibt BADER (1998). Der Ansatz stellt sicher, daß ein Agent seine Aufgaben genau einmal ausführt, verhindert jedoch nicht, daß Agenten durch eine Netzwerkpartitionierung langfristig blockiert werden.

Die Blockierung von Agenten im Falle von Fehlern durch einen Warm-Backup-Ansatz zu verhindern wird von JOHANSEN, VAN RENESSE UND SCHNEIDER (1995) vorgeschlagen. Eine *Nachhut* (engl.: *Rear-Guard*), die bei der Migration zurückgelassen wird, soll die Ausführung des Agenten auf dem Zielrechner der Migration überwachen. In JOHANSEN ET AL. (1999) wird dieser Vorschlag konkretisiert. Sobald ein Agent bei der Migration auf dem Zielrechner ankommt, wird der Agent mittels eines zuverlässigen Broadcast-Protokolls auf eine Menge weiterer Rechner – zum Beispiel einige der Rechner, welche den Agent in der Vergangenheit ausgeführt haben – zusätzlich versendet. Diese Rechner sind dafür zuständig, die Ausführung des Agenten zu überwachen. Sobald der Agent auf diesen Überwachungsrechnern angekommen ist, wird auf dem Zielrechner die ihm zugeordnete *Aktion* gestartet. Als eine Aktion werden von JOHANSEN ET AL. (1999) die auf einem einzelnen Rechner auszuführenden Operationen bezeichnet. Für jede Aktion gibt es eine *Wiederherstellungsaktion* (engl.: *recovery action*). Ein in das Broadcast-Protokoll integriertes, kombiniertes *Auswahl-* (engl.: *election*) und *Überwachungs-Protokoll* (engl.: *monitoring protocol*) stellt sicher, daß bei Abbruch der Aktion (zum Beispiel durch Rechnerabsturz) auf genau einem der Überwachungsrechner die der abgebrochenen Aktion zugeordnete Wiederherstellungsaktion ausgeführt wird. Hierdurch werden jedoch die auf dem ausgefallenen Rechner bereits durchgeführten Operationen nicht automatisch zurückgesetzt. Hat der Agent dort zum Beispiel den Inhalt einer nur lokal zugreifbaren Datenbank geändert, so bleiben diese Änderungen erhalten, da diese von dem Überwachungsrechner aus gar nicht rückgängig gemacht werden können. Es ergibt sich also eine äußerst unklare Semantik der Ausführung eines Agenten. Weiterhin toleriert das Protokoll in der vorgestellten Version keine Netzwerkpartitionierung.

Mit einem auf dem in diesem Kapitel vorgeschlagenen blockierungsfreien Protokoll aufbauenden Protokoll stellen ASSIS SILVA UND POPESCU-ZELETIN (1998) einen weiteren Warm-Backup-Ansatz vor. Das Protokoll basiert ebenfalls auf einem *Stufenkonzept* (engl.: *stage concept*). Bei der Migration wird ein Agent innerhalb einer Transaktion auf mehrere Rechner (die Stufe) transportiert. Einer der Rechner führt den Agent aus, die anderen überwachen die Ausführung. Um die Wiederherstellung eines Agenten nach Rechnerausfall zu unterstützen, schreibt der Agent regelmäßig Rücksetzpunkte - sowohl in kürzeren Abständen lokal auf dem

ausführenden Rechner als auch in längeren Abständen in einer “Verteilten Datenbank”. Die verteilte Datenbank besteht aus Datenreplikaten auf den Rechnern der aktuellen Stufe. Der Zugriff auf die Replikate der verteilten Datenbank wird durch Votieren mit Mehrheitsentscheid geregelt. Fällt der einen Agenten ausführende Rechner aus, so wird nach Neustart des Rechners der Agent beim aktuellsten lokalen oder globalen Rücksetzpunkt aufgesetzt. Fällt der Rechner längerfristig aus, wird dies durch die überwachenden Rechner erkannt. Ein Auswahl-Protokoll stellt sicher, daß nur einer der überwachenden Rechner den Agenten beim aktuellsten globalen Rücksetzpunkt aufsetzt. Der Mechanismus stellt sicher, daß immer nur der zuletzt für einen Agenten ausgewählte Ausführungsrechner auf die verteilte Datenbank schreiben darf. Hierdurch toleriert das Protokoll einerseits Netzwerkpartitionierungen und kann andererseits damit umgehen, daß ein nach einem längerfristigen Ausfall neu startender Rechner einen beim Ausfall des Rechners ausgeführten Agenten neu startet, obwohl dieser Agent zwischenzeitlich längst auf einem der überwachenden Rechner gestartet wurde. Stellt ein Agent auf einem Rechner fest, daß es eine “aktuellere” Ausführungsinstanz gibt (d.h. es wurde zwischenzeitlich ein anderer Rechner als Ausführungsrechner ausgewählt), dann wird der Agent auf den letzten von ihm geschriebenen globalen Rücksetzpunkt zurückgesetzt und die Ausführung auf dem Rechner beendet. Nachteil des Mechanismus ist, daß eine genau-einmal Ausführung eines Agenten nur durch aufwendige Programmierung (Rücksetzpunkte setzen, Rücksetzoperationen zur Herstellung des letzten globalen Rücksetzpunktes) zu erreichen ist. Ein weiterer Nachteil des Protokolls ist der gegenüber dem in diesem Kapitel vorgestellten Protokoll wesentlich höhere Aufwand. Dieser ist einerseits dadurch bedingt, daß bei der Migration relativ zu unserem Protokoll etwa 50% mehr Rechner in der verteilten Transaktion beteiligt sind und daß zur (recht gering ausfallenden) Erhöhung der Fehlertoleranz ein 3-Phasen-Commit-Protokoll verwendet wird.

ASSIS SILVA UND KRAUSE (1997) stellen ein dem unseren Modell sehr ähnliches Modell für verteilte Transaktionen basierend auf mobilen Agenten vor. Ein Agent hat eine Menge von *Aufgaben* (engl.: *tasks*) zu erledigen. Jede Aufgabe wird auf einem Rechner innerhalb einer Transaktion bearbeitet. Es besteht zusätzlich die Möglichkeit, mehrere Aufgaben hintereinander auf mehreren Rechnern innerhalb einer Transaktion zu bearbeiten. Die Ausführung von Agenten wird überwacht, um die sich auf einem längerfristig ausgefallenen Rechner befindlichen Agenten auf anderen Rechnern neu starten zu können. Hierfür wird bei jeder Migration der Zustand des Agenten in einer zentralen Kontextdatenbank geschrieben. Die Überwachung von Agenten geschieht zentral. Wird die Bearbeitung eines Agenten abgebrochen, werden die bisherigen Aktionen des Agenten per *Kompensation* zurückgesetzt. Leider werden in der Veröffentlichung keinerlei Algorithmen zum Erreichen der beschriebenen Semantik angegeben. Die zentrale Kontextdatenbank und Überwachung der Agenten deuten darauf hin, daß das dem beschriebenen Modell zugrundeliegende Fehlermodell keine Netzwerkpartitionierung zuläßt.

Einen wesentlich höheren Grad an Fehlertoleranz bietet der von MINSKY ET AL. (1996) und SCHNEIDER (1997A) vorgestellte Mechanismus. Während die bisher vorgestellten Ansätze davon ausgehen, daß Rechner nur unter Crash-Fehlern leiden, umfaßt das diesem Mechanismus

zugrundeliegende Fehlermodell auch byzantinische Fehler. Auch dieser Mechanismus basiert auf einem Stufenkonzept. Im Gegensatz zu den bisherigen Ansätzen führt hier ähnlich einem Hot-Backup-Ansatz jeder der Rechner einer Stufe den Agent aus. Im Gegensatz zum Hot-Backup sind hier jedoch alle Rechner gleichberechtigt - es wird nicht zwischen Primärrechner und Backup-Rechner unterschieden. Nach der Ausführung des Agenten verschickt jeder Rechner den Agent bei der Migration zu allen Rechnern der nächsten Stufe. Die Rechner jeder Stufe erhalten daher mehrere Agenten (maximal n Agenten wenn die vorhergehende Stufe n Rechner umfaßte; weniger als n Agenten bei Rechner-/Netzwerkausfällen), aus denen sie mittels einer Mehrheitsentscheidung den auszuführenden Agenten ermitteln. Hiermit können bei n Rechnern pro Stufe $n/2$-1 (Rechner- und Netzwerk-) Fehler maskiert werden. Dieser Ansatz ist dem *NMR-Ansatz* (*N modular redundancy*, vgl. JALOTE (1994)) nachempfunden, welcher erfolgreich in fehlertoleranter Hardware angewandt wurde. Obwohl dieser Ansatz den bestmöglichen Grad an Fehlertoleranz bietet, ist er in der Realität aus drei Gründen in nur sehr wenigen Situationen anwendbar. Erstens führt der Mechanismus einen extremen Mehraufwand ein. Bei n Rechnern pro Stufe findet die Berechnung n-fach statt. Zusätzlich muß jeder Rechner noch aus n erhaltenen Agenten den auszuführenden bestimmen. Zweitens setzt der Algorithmus voraus, daß die durch den Agenten genutzten Dienste auf allen Rechnern einer Stufe repliziert sind und diese Replikate auch bei Fehlern konsistent gehalten werden. Und schließlich setzt der Algorithmus voraus, daß die Ausführung eines Agenten auf einem Rechner determiniert ist, d.h. daß bei gleicher Ausgangssituation die gleichen Resultate erzeugt werden. Um dies zu erreichen, müßte zum Beispiel eine Abfrage der Uhrzeit auf den Rechnern einer Stufe den Replikaten eines Agenten dieselbe Uhrzeit liefern. Der Mehraufwand des Mechanismus wird wohl nur für sehr wenige Anwendungsgebiete tolerierbar sein, replizierte Dienste und garantiert determinierte Ausführung auf mehreren Rechnern dürfte nur in den seltensten Anwendungen gegeben sein.

4.9 Diskussion

Ergebnisse

In diesem Kapitel wurden Protokolle entwickelt, welche die in Definition 4-1 festgelegte genau-einmal Ausführung für mobile Agenten implementieren. Der Entwicklung der Protokolle zugrundegelegt wurden hierbei das in Abschnitt 2.2 beschriebene Agentenmodell und die in Abschnitt 4.1 beschriebenen System- und Fehlermodelle. Das zuerst entwickelte Basisprotokoll stellt zwar die genau-einmal Ausführung mobiler Agenten sicher, bietet jedoch keinen Schutz vor Blockierung bei Knoten- oder Netzwerkausfall. Deshalb wurde das Basisprotokoll zu einem Protokoll weiterentwickelt, welches bei Systemfehlern die Blockierung der Ausführung der mobilen Agenten verhindert.

Abschnitt 4.6 zeigt, daß durch das blockierungsfreie Protokoll eine signifikant geringere Blockierwahrscheinlichkeit im Vergleich zum Basisprotokoll erreicht wurde. Jedoch zeigen Abschnitt 4.5 und Abschnitt 4.7, daß diese Verbesserung nur durch einen wesentlichen Mehr-

aufwand bei der Ausführung der mobilen Agenten erkauft wird. Abschnitt 4.6.5.2 zeigt, daß im Mittel die für diesen Mehraufwand benötigte Zeit unter realen Bedingungen größer ist als die durch die Vermeidung von Blockierungen gewonnene Zeit. Durch die Funktionsweise des Protokolls ist es der Anwendung bzw. dem Anwender allerdings möglich, die Fehlertoleranz und damit den erforderlichen Mehraufwand an die Erfordernisse der Anwendung zu adaptieren: Wird ein hoher Grad an Fehlertoleranz erwartet, d.h. ist eine Vermeidung von Blockierungen notwendig, so wird eine große Stufengröße verwendet welche viel Mehraufwand erzeugt. Für einen geringen Grad an Fehlertoleranz reicht eine kleine Stufengröße mit entsprechend geringerem Mehraufwand aus. Hierbei ist das Basisprotokoll der Sonderfall des blockierungsfreien Protokolls mit einer Stufengröße von 1.

Einschränkungen des Agentenmodells

Zwei Besonderheiten des in Abschnitt 2.2 vorgestellten Agentenmodells stehen in direkter Beziehung mit der Funktionsweise der in diesem Kapitel vorgestellten Protokolle. Sowohl die Einschränkung, daß ein mobiler Agent nicht mit einem anderen mobilen Agent direkt kommunizieren darf als auch die Einschränkung, daß ein mobiler Agent keinen neuen mobilen Agent starten bzw. einen Klon von sich selbst erzeugen darf, hängt eng damit zusammen, daß die einzelnen Schritte der mobilen Agenten in einer Transaktion ausgeführt werden. Zu den Eigenschaften einer Transaktion gehört unter anderem, daß die Ergebnisse der Transaktion für andere Transaktionen erst nach erfolgreichem Transaktionsabschluß sichtbar sein dürfen. Kommunizieren zwei mobile Agenten direkt, wird diese Eigenschaft der Transaktionen verletzt, wie die zwei folgenden Beispiele zeigen.

Beispiel 4-1. Zwei Agenten A_1 und A_2 werden beide jeweils in einer Schritt-Transaktion ausgeführt. A_1 liest den Wert einer Ressource R und teilt diesen Wert dem Agenten A_2 mit. Sowohl A_1 als auch A_2 berechnen aufgrund dieses Wertes unterschiedliche neue Werte für R. Da die Ressource von Agent A_1 gesperrt ist, schreibt zuerst A_1 seinen neuen Wert für R und schließt die Schritt-Transaktion ab. Erst dann kann A_2 seinen Wert in R schreiben – und überschreibt damit den von A_1 geschriebenen Wert. Diese Konstellation in Konflikt stehender Operationen wird im Bereich der Datenbanken auch "*verlorengegangene Änderung*" (engl.: *lost update*) genannt (vgl. HÄRDER UND RAHM (1999)).

Beispiel 4-2. Agent A_1 berechnet innerhalb einer Schritt-Transaktion einen Wert und kommuniziert diesen Wert noch innerhalb der Transaktion an A_2. Agent A_2 beendet die Schritt-Transaktion, in der er den Wert von A_1 erhalten hat, und setzt seine Ausführung mit dem nächsten Schritt fort. A_1 bricht danach seine Schritt-Transaktion ab. A_2 rechnet jetzt mit einem Wert, den A_1 logisch gesehen nie berechnet hat (auch "*Zugriff auf schmutzige Daten* (engl.: *dirty read*)" genannt, vgl. ebenfalls HÄRDER UND RAHM (1999)).

Die beiden Beispiele zeigen, daß eine direkte Kommunikation zwischen zwei Agenten bei der Funktionalität der in diesem Kapitel vorgestellten Protokolle nicht möglich ist. Indirekte Kommunikation mittels einer Kommunikationsressource hingegen, beispielsweise mittels einer

transaktionalen Nachrichtenwarteschlange, kann natürlich problemlos realisiert werden. Dies eignet sich jedoch nicht zur Realisierung von Dialogen, da beispielsweise eine von einem Agent A_1 in eine transaktionale Nachrichtenwarteschlange geschriebene Nachricht erst dann von einem anderen Agent gelesen werden kann, wenn A_1 die Schritt-Transaktion, innerhalb der die Nachricht geschrieben wurde, erfolgreich beendet hat. Wie jedoch Kapitel 5 zeigen wird, wird selbst diese Art der Kommunikation problematisch, wenn die Möglichkeit besteht, die Ausführung des Agenten teilweise zurückzusetzen.

Ein sehr ähnliches Problem ergibt sich, wenn ein Agent A_1 einen weiteren Agenten A_2 startet bzw. einen Klon A_2 von sich selbst erzeugt. Bricht nämlich die umgebende Schritt-Transaktion ab, so muß dafür gesorgt werden, daß A_2 auch abgebrochen wird. Hat dieser jedoch zwischenzeitlich den Knoten, auf dem er erzeugt wurde, verlassen, ist dies nicht mehr ohne weiteres möglich. Eine mögliche Lösung ist, A_2 innerhalb der Schritt-Transaktion von A_1 auf die Knoten der ersten Stufe von A_2 zu verteilen. Somit erscheint A_2 auf den Knoten seiner ersten Stufe, sobald A_1 die Schritt-Transaktion, innerhalb der A_2 erzeugt wurde, beendet. Wie Kapitel 5 zeigen wird, ist auch diese Lösung sehr problematisch, wenn die Möglichkeit besteht, die Ausführung eines Agenten teilweise zurückzusetzen.

Kapitel 5

Partielles Rücksetzen

Die im letzten Kapitel vorgestellten Mechanismen realisieren im Fehlerfall eine strikte *Forward Recovery*: Schlägt aus beliebigem Grund die Durchführung einer Schritt-Transaktion fehl, so werden die bereits ausgeführten Schritte des Agenten nicht zurückgesetzt (*Backward Recovery*), sondern die Ausführung des Agenten entweder durch eine Wiederholung des abgebrochenen Schrittes oder durch die Ausführung eines alternativen Schrittes "nach vorne" fortgesetzt. Es gibt jedoch Situationen, in denen der Abbruch und der Neustart eines Schrittes nicht zur Behandlung einer Fehlersituation ausreichen. Stellt der Agent fest, daß die von ihm bisher verfolgte Strategie nicht zum Ziel führt – weil ihm beispielsweise die Zugriffsberechtigung zu einer Ressource fehlt oder weil das bisherige Vorgehen nicht die gewünschten Ergebnisse liefert – so kann es notwendig sein, Teile der bisherigen Ausführung des Agenten rückgängig zu machen.

Das partielle Rücksetzen der Ausführung eines mobilen Agenten kann prinzipiell auf zwei verschiedene Arten realisiert werden. Eine Möglichkeit ist, daß der Entwickler des Agenten die dazu notwendige Funktionalität in den Agenten direkt integriert. In diesem Fall unterscheidet sich für die Agentenplattform das Rücksetzen nicht von der normalen Agentenausführung, der Entwickler eines Agenten muß jedoch – für jeden Agenten erneut – das Rücksetzen des Agenten vollständig ausprogrammieren. Diese Aufgabe ist schon für normale Programme mit (mehr oder weniger) feststehendem Programmablauf nicht ganz einfach. Beinahe unmöglich wird diese Aufgabe jedoch, wenn durch die Verwendung des in Abschnitt 3.2 vorgestellten Reiseroutenkonzeptes die Schritte des Agenten in sehr unterschiedlicher Reihenfolge ausgeführt werden können, da das Rücksetzen von Operationen im allgemeinen von deren Ausführungsreihenfolge abhängt. Die deshalb zu bevorzugende Möglichkeit zum Rücksetzen der Agentenausführung ist, daß die Agentenplattform Mechanismen zur Verfügung stellt, die auf Anforderung das Rücksetzen weitestgehend automatisiert durchführen. Die Entwicklung solcher Mechanismen ist Ziel dieses Kapitels. Erste Versionen der hier entwickelten Mechanismen und Konzepte wurden in STRASSER UND ROTHERMEL (2000) publiziert.

5.1 Problemstellung

Die exakte Problemstellung des partiellen Rücksetzens der Ausführung mobiler Agenten eröffnet sich nur durch die Betrachtung des Ausführungsmechanismus für mobile Agenten. Im Kontext dieser Arbeit wird nur betrachtet, wie das partielle Rücksetzen für die im vorherigen Kapitel vorgestellten Mechanismen zur genau-einmal Ausführung mobiler Agenten realisiert werden kann. Eine direkte Übertragung der dabei entwickelten Konzepte auf andere Ausführungsmechanismen ist nur eingeschränkt möglich.

Es wird im Laufe dieses Kapitels offensichtlich werden, daß das für das partielle Rücksetzen relevante Grundprinzip – nämlich die Ausführung des Agenten in einer Schritt-Transaktion mittels Lesen des Agenten von stabilem Speicher, Ausführen des Agenten und Schreiben des Agenten auf stabilen Speicher – sowohl beim Basisalgorithmus als auch bei der blockierungsfreien Erweiterung dasselbe ist. Deshalb wird der Mechanismus zum partiellen Rücksetzen vorerst anhand des Basisprotokolls zur genau-einmal Ausführung mobiler Agenten entwickelt und dann gezeigt, wie dieser Mechanismus in das erweiterte Protokoll integrierbar ist.

Abbildung 5-1 zeigt die Ausführung der Schritte i, i+1 und i+2 des Agenten A, wobei die Schritte i und i+1 schon vollständig abgeschlossen sind und Schritt i+2 sich noch in Ausführung befindet. Die Ausführung eines Schrittes erfolgt dabei wie in Abschnitt 4.3 beschrieben: Innerhalb einer Transaktion, der Schritt-Transaktion, wird der Agent aus der Eingangswarteschlange des ausführenden Knotens gelesen, ausgeführt und in die Eingangswarteschlange des nachfolgenden Knotens geschrieben. Während der Ausführung ändert der Agent hierbei die lokalen Ressourcen R_k (k=i, i+1, i+2) der ausführenden Knoten. Im Laufe der Ausführung des Schrittes i+2 stellt der Agent fest, daß ein partielles Rücksetzen notwendig ist. Da die meisten heutzutage verfügbaren Transaktionsmanagementsysteme keine Rücksetzpunkte für Ressourcen innerhalb einer Transaktion unterstützen, kann nur auf Zustände zwischen (beziehungsweise vor) den

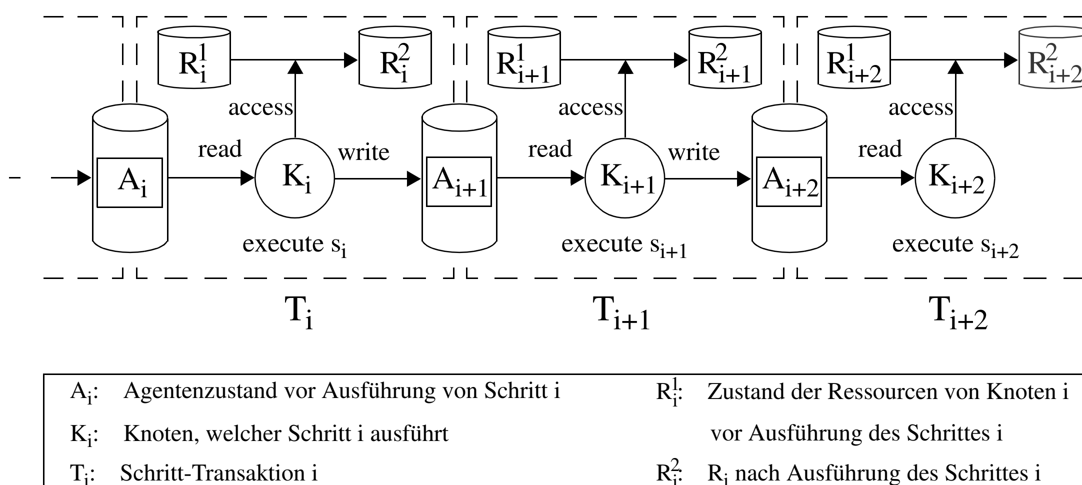


Abbildung 5-1. Ausführung eines Agenten

Schritt-Transaktionen, d.h. zwischen (beziehungsweise vor) Schritten, zurückgesetzt werden. Im vorliegenden Beispiel bedeutet dies, daß der Agent nur auf die Zustände A_{i+2}, A_{i+1}, A_i,..., A_1 zurückgesetzt werden kann.

Muß der Agent nur auf den Zustand A_{i+2} zurückgesetzt werden, ist dies einfach realisierbar, indem die Schritt-Transaktion T_{i+2} abgebrochen wird. Das Transaktionsmanagement sorgt in diesem Falle automatisch dafür, daß der Agent im Zustand A_{i+2} in der Eingangswarteschlange des Knotens K_{i+2} vorliegt und daß die während der Ausführung des Schrittes an den Ressourcen R_{i+2} durchgeführten Änderungen rückgängig gemacht werden. Wesentlich problematischer ist das Rücksetzen des Agenten auf die Zustände A_1, A_2, ..., A_{i+1}. Das Problem hierbei ist, daß nach Abschluß der Schritt-Transaktionen T_1, T_2, ..., T_{i+1} andere Anwendungen auf die durch den Agenten geänderten Ressourcen R_1, R_2, ..., R_{i+1} zugreifen können. Somit ist ein einfaches Rücksetzen der Ressourcen auf den ursprünglichen Zustand durch das Transaktionsmanagement nicht mehr möglich. Aus diesem Grunde muß das Rücksetzen der Ressourcen in diesem Fall mittels *Kompensationstransaktionen* (engl.: *compensation transaction*) erfolgen (vgl. auch KORTH, LEVY UND SILBERSCHATZ (1990)). Wie sich in den folgenden Absätzen zeigen wird, kann nicht einmal der Zustand des Agenten zurückgesetzt werden, indem einfach der alte Datenzustand des Agenten wieder hergestellt wird. Im Gegensatz zu gängigen Ansätzen (z.B. GARCIA-MOLINA UND SALEM (1987)) muß also zusätzlich zum Zustand der Ressourcen auch der Zustand des Agenten mittels Kompensation zurückgesetzt werden. Zum Zwecke der Vereinheitlichung wird in Anlehnung an KORTH, LEVY UND SILBERSCHATZ (1990) der Begriff des erweiterten Zustandsraumes eingeführt:

Definition 5-1: Erweiterter Zustandsraum

Der erweiterte Zustandsraum ist die Vereinigung des Zustandes des Agenten mit dem Zustand der Ressourcen, auf die der Agent zugreift.

Hiermit kann die Ausführung eines Schrittes als Folge von Operationen auf diesem erweiterten Zustandsraum beschrieben werden. Das Zurücksetzen eines bereits abgeschlossenen Schrittes geschieht dann mittels Kompensationsoperationen auf dem erweiterten Zustandsraum.

Im Gegensatz zum Rücksetzen der Ausführung eines sich in Ausführung befindlichen Schrittes durch Abbruch der Schritt-Transaktion werden für Kompensationsoperationen zusätzliche Informationen benötigt. Löscht ein Agent beispielsweise während eines Schrittes Daten, welche er in vorherigen Schritten gesammelt hat, aus seinem Datenzustand, so werden diese verworfenen Daten für die Kompensation dieses Schrittes benötigt. Es ist also notwendig, während der Ausführung eines Schrittes die für die Kompensation des Schrittes notwendigen Informationen zu sammeln und im Falle einer Kompensation des Schrittes wieder zur Verfügung zu stellen.

Der folgende Abschnitt diskutiert das Konzept der Kompensation und dessen Einschränkungen. Anschließend wird das in Abschnitt 2.2 eingeführte Agentenmodell detailliert und um das Rücksetzen von Agenten erweitert. Abschnitt 5.4 präsentiert einen auf diesem erweiterten Mo-

dell basierenden Mechanismus zum Rücksetzen, Abschnitt 5.5 beschreibt mögliche Optimierungen dieses Mechanismus'. Mögliche Erweiterungen des Reiseroutenkonzeptes zur Integration des Rücksetzmechanismus' und eine Diskussion der Ergebnisse schließen das Kapitel ab.

5.2 Kompensation

Die Kompensation einer Operation hat zum Ziel, die semantischen Effekte dieser Operation zu beseitigen. Unglücklicherweise ist dies jedoch nicht immer möglich. Inwieweit die Kompensation einer Operation möglich ist hängt hierbei sowohl von der Operation als solcher als auch vom Anwendungsprogramm ab. In diesem Abschnitt wird diese Problematik der Kompensation näher betrachtet. Eine sehr ausführliche Diskussion findet man in dem Artikel von KORTH, LEVY UND SILBERSCHATZ (1990).

Der bei Kompensationsoperationen wünschenswerte Fall ist, daß die Kompensationsoperation den Effekt der zu kompensierenden Operation komplett eliminiert. Wird die Kompensationsoperation CO_1 direkt nach der zu kompensierenden Operation O_1 ausgeführt, so wird in diesem Falle der ursprüngliche, erweiterte Zustandsraum Z wieder hergestellt, d.h. $CO_1(O_1(Z)) \equiv Z$. Wird zwischen der Ausführung von O_1 und CO_1 eine beliebige andere Operation (oder Menge von Operationen) O_2 auf den ursprünglichen, erweiterten Zustandsraum ausgeführt, dann spiegelt in diesem Fall der nach der Kompensationsoperation CO_1 resultierende Zustandsraum nur noch die Auswirkungen der Operation(en) O_2 wider, d.h. $CO_1(O_2(O_1(Z))) \equiv O_2(Z)$. Anwendungen, bei denen diese Art der Kompensation möglich ist, sind in der realen Welt nur sehr selten anzutreffen. Kann auf ein Bankkonto mit unbegrenztem Kredit nur mittels den Operationen *Abheben*(*Geld*) und *Einzahlen*(*Geld*) zugegriffen werden, so sind *Abheben*(..) und *Einzahlen*(..) die Kompensationsoperationen der jeweils anderen Operation. Wird von kurzfristig anfallenden Zinsen abgesehen, so kann in diesem Fall ein *Abheben*(x) durch ein *Einzahlen*(x) jederzeit vollständig kompensiert werden und umgekehrt. Dies gilt schon nicht mehr, wenn das Konto keinen unbegrenzten Kredit erlaubt. Wird nach einem *Einzahlen*(x) eine Operation *Abheben*(y) ausgeführt, welche das Konto bis zum maximalen Kredit leert, dann ist die Kompensationsoperation *Abheben*(x) zur Kompensation von Einzahlen(x) nicht erfolgreich, da der Kreditrahmen überzogen würde. Besitzt das Konto zwar einen unbegrenzten Kreditrahmen, jedoch eine zusätzliche (und durchaus übliche) Operation zum Lesen des Kontostandes, dann ist auch hier die vollständige Kompensation nicht immer möglich, da Operationen abhängig gemacht werden können von der Höhe des Kontostandes. Beispiel: Anwendung 1 führt *Einzahlen*(x) aus. Da der Kontostand eine gewisse Summe übersteigt, führt Anwendung 2 danach *Abheben*(y) aus. Setzt danach Anwendung 1 ihre Ausführung zurück, kann man zwei Fälle unterscheiden. Hätte Anwendung 2 auch ohne das *Einzahlen*(x) das *Abheben*(y) ausgeführt, so wird mit *Abheben*(x) das *Einzahlen*(x) vollständig kompensiert. Im anderen Falle müßten jedoch für eine vollständige Kompensation auch die Operationen der zweiten Anwendung zurückgesetzt werden.

Vollständige Kompensation ist also eher selten möglich, jedoch reicht es bei vielen Anwendungen aus, daß durch die Kompensation entweder ein semantisch äquivalenter Zustand oder ein im Rahmen der Anwendung akzeptabler Zustand hergestellt wird. Verwendet beispielsweise ein Agent (die Aussagen gelten auch entsprechend für beliebige Programme) digitales Geld (vgl. z.B. CHAUM (1985)), welches er in seinem Datenzustand mit sich führt, um Waren einzukaufen, dann bekommt er bei der Kompensation des Einkaufs denselben Geldbetrag vom Händler zurück. Da der Händler die vom Agent erhaltenen elektronischen Münzen jedoch im allgemeinen nicht mehr besitzt – bei heute gängigen Systemen werden die Münzen beispielsweise sofort bei der Bank eingelöst – bekommt der Agent andere Münzen zurück, die sich zumindest in der Seriennummer unterscheiden. Hierdurch wird zwar ein semantisch äquivalenter Zustand im Agenten hergestellt – der Agent enthält dieselbe Geldsumme wie vor dem Einkauf. Die Repräsentation des Zustandes jedoch unterscheidet sich vom Original. Es ist jedoch auch denkbar, daß der Händler für die Kompensation des Einkaufs eine Aufwandsentschädigung erhebt. In diesem Falle bekommt der Agent weniger Geld zurück, als er beim Einkauf ausgegeben hat, womit der resultierende Zustand nicht mehr semantisch äquivalent ist – der Agent hat weniger Geld als im Ausgangszustand. Ob dies für den Agent bzw. den Anwender akzeptabel ist, muß dieser schon vor dem Einkauf entscheiden – und gegebenenfalls bei einem Händler kaufen, der im Falle einer Kompensation keine Gebühren erhebt. Noch ungünstiger ist der Fall, wenn der Händler bei einer Kompensation anstatt von Geld nur eine Gutschrift zurückgibt. In diesem Falle enthält der Agent nach der Kompensation vom Typ her andere Information als vor dem Einkauf. Unabhängig davon, ob der Agent nun anderes Geld, weniger Geld oder gar nur eine Gutschrift erhält, muß der Agent nach der Kompensation mit dieser geänderten Situation zurechtkommen, d.h. die verschiedenen möglichen Resultate einer Kompensation müssen dem Entwickler des Agenten bekannt sein und von ihm bei der Implementation des Agenten berücksichtigt werden.

Wie schon weiter oben erwähnt gibt es Fälle, in denen Kompensation nur bedingt möglich ist. Beispielsweise ist die Kompensation einer Einzahlung auf ein Bankkonto nur dann möglich, wenn zwischen der Einzahlung und der Kompensation nicht soviel Geld vom Konto entnommen wird, daß die Kompensation den Kreditrahmen überzieht. Die Lösung dieses Problems liegt außerhalb des Rahmens dieser Arbeit. Lösungsansätze werden in GARCIA-MOLINA UND SALEM (1987) und REUTER, SCHNEIDER UND SCHWENKREIS (1997) diskutiert.

Schließlich gibt es noch Fälle, in denen gar keine Kompensation möglich ist. Klassische Beispiele sind hier Interaktionen mit der realen Welt wie das Bohren eines Loches durch einen programmgesteuerten Bohrautomat oder die Ausgabe von Bargeld am Bankautomat. Aber auch innerhalb des Rechners gibt es Operationen, bei denen eine Kompensation nicht möglich ist. Ist beispielsweise ein Händler nicht bereit oder durch Konkurs nicht in der Lage, einen Einkauf rückgängig zu machen, kann die Kaufoperation nicht im Sinne des Käufers kompensiert werden. Hier muß also im Falle einer "Kompensation" der Verlust der ausgegebenen Summe in Kauf genommen werden. Ein weiteres Beispiel ist das Löschen einer sehr großen Datenmenge. Um diese Operation zu kompensieren, müßten die gelöschten Daten bekannt sein – beispiels-

weise durch Logging der gelöschten Daten. Aus Effizienzgründen ist es jedoch nicht sinnvoll, solch große Datenmengen in ein Log zu schreiben. Ist es notwendig, solch nicht kompensierbare Operationen zurückzusetzen, ist im allgemeinen ein manueller Eingriff zur Durchführung des Rücksetzens notwendig.

Da der Schwerpunkt dieses Kapitels auf den Aspekten des verteilten Rücksetzens mobiler Agenten und möglicher Optimierungen liegt, wird für den Rest des Kapitels davon ausgegangen, daß die in den einzelnen (Agenten-)Schritten ausgeführten Operationen immer kompensiert werden können in dem Sinne, daß durch die Kompensation ein aus Sicht der Anwendung(en) akzeptabler Systemzustand entsteht.

5.3 Erweiterung des Agentenmodells

Mit den in den letzten Abschnitten gewonnenen Erkenntnissen ist es nun möglich, das in Abschnitt 2.2 vorgestellte Agentenmodell um das partielle Rücksetzen von Agenten zu erweitern.

Beim Rücksetzen eines einzelnen Schrittes ändert sich sowohl der Datenzustand des Agenten als auch der Zustand jener Ressourcen, auf die der Agent während des rückzusetzenden Schrittes zugegriffen hatte. Wie die vorhergehenden Abschnitte zeigten, werden beim Rücksetzen des Agenten im allgemeinen weder der originale Zustand des Agenten noch die originalen Zustände der Ressourcen wieder hergestellt. Deshalb sind die folgenden Definitionen notwendig:

Definition 5-2: Agentenzustand $\overline{A}_i$

Der Agentenzustand $\overline{A}_i$ beschreibt den Datenzustand des Agenten A nach dem Rücksetzen des i-ten Schrittes.

Definition 5-3: Ressourcenzustände R_i^3, R_i^4

Die Ressourcenzustände R_i^3, R_i^4 beschreiben die Zustände der Ressourcen des Knotens K_i vor (R_i^3) beziehungsweise nach (R_i^4) dem Rücksetzen des i-ten Schrittes des Agenten.

Da zwischen der Ausführung des i-ten Schrittes und seinem Rücksetzen andere Agenten bzw. Anwendungen auf die Ressourcen zugreifen können ist R_i^3 im allgemeinen ungleich R_i^2 (Zustand der Ressourcen nach der Ausführung des Schrittes).

Definition 5-4: Rücksetzinformationen I_i

Die Rücksetzinformationen I_i beschreiben die zusätzlich zu Agenten- und Ressourcenzustand notwendigen Daten zum Rücksetzen des Schrittes i.

Zur Gewinnung der Rücksetzinformationen I_i muß die in Abschnitt 2.2 eingeführte Schrittfunktion $\boldsymbol{s_i}$ erweitert werden:

$$s_i\colon (A_i, R_i^1) \mapsto (A_{i+1}, R_i^2, I_i) \tag{5-1}$$

Die Rücksetzfunktion $\boldsymbol{r_i}$ beschreibt die Änderung des Agenten- und Ressourcenzustandes durch das Rücksetzens eines einzelnen, bereits abgeschlossenen Schrittes *i*:

$$r_i\colon (A_{i+1}, R_i^3, I_i) \mapsto (\overline{A}_i, R_i^4) \tag{5-2}$$

Wurde zuvor auch Schritt *i*+1 schon zurückgesetzt, gilt

$$r_i\colon (\overline{A}_{i+1}, R_i^3, I_i) \mapsto (\overline{A}_i, R_i^4) \tag{5-3}$$

Die Ausführung der Rücksetzfunktion $\boldsymbol{r_i}$ ist nur dann möglich, wenn sich der Agent momentan im Zustand A_{i+1} (bzw. $\overline{A}_{i+1}$) befindet. Das Rücksetzen der *n* Schritte *i*, *i*-1,..., *i*-*n*+1 ($n \leq i$) wird durch

$$r_{i,n}\colon (A_{i+1}, R_i^3, R_{i-1}^3, R_{i-2}^3, \ldots, R_{i-n+1}^3, I_i, I_{i-1}, \ldots, I_{i-n+1}) \mapsto (\overline{A}_{i-n+1}, R_i^4, R_{i-1}^4, R_{i-2}^4, \ldots, R_{i-n+1}^4) \tag{5-4}$$

beziehungsweise

$$r_{i,n}\colon (\overline{A}_{i+1}, R_i^3, R_{i-1}^3, R_{i-2}^3, \ldots, R_{i-n+1}^3, I_i, I_{i-1}, \ldots, I_{i-n+1}) \mapsto (\overline{A}_{i-n+1}, R_i^4, R_{i-1}^4, R_{i-2}^4, \ldots, R_{i-n+1}^4) \tag{5-5}$$

beschrieben. Die Funktion $r_{i,n}$ ist hierbei eine Verkettung $r_{i,n} = r_{i-n+1} \circ r_{i-n+2} \circ \ldots \circ r_i$ der Rücksetzfunktionen der einzelnen Schritte, d.h. auf A_{i+1}, R_i^3 und I_i wird r_i angewendet, auf das resultierende $\overline{A}_i$ und die R_{i-1}^3 und I_{i-1} wird dann r_{i-1} angewendet und so weiter.

Wird das Modell in der bis hierher beschriebenen Version in die Praxis umgesetzt, bedeutet dies für den Entwickler eines Agenten, daß er für jeden Schritt des Agenten Code bereitstellen muß (nämlich die Rücksetzfunktion), der sämtliche Auswirkungen des Schrittes auf Ressourcen und auf den Agent selbst kompensiert. Außerdem müssen während der Ausführung eines Schrittes die Rücksetzinformationen für den Code zur Kompensation manuell gesammelt werden, d.h. das Sammeln der notwendigen Rücksetzinformationen muß in den Code des Schrittes integriert werden. Dies ist sowohl sehr unkomfortabel als auch fehleranfällig.

Eine Verfeinerung des Agentenmodelles ermöglicht in gewissem Umfang die Unterstützung des Agentenentwicklers durch die Agentenplattform. Die Verfeinerung ergibt sich aus der Beobachtung, daß man die im Datenzustand des Agenten enthaltenen Daten in zwei Kategorien klassifizieren kann:

Definition 5-5: Stark reversible Objekte (engl.: strongly reversible objects), $A_{S,i}$

Stark reversible Objekte sind Datenobjekte im (privaten) Datenzustand des Agenten, die nach dem Rücksetzen eines Schrittes immer dieselben Daten enthalten wie vor der Ausführung des rückzusetzenden Schrittes und daher mittels einer Kopie der Objekte wiederhergestellt werden können. Der Zustand der stark reversiblen Objekte eines Agenten A vor der Ausführung des Schrittes i wird mit $A_{S,i}$ bezeichnet.

Definition 5-6: Schwach reversible Objekte (engl.: weakly reversible objects), $A_{W,i}$

Schwach reversible Objekte sind Datenobjekte im (privaten) Datenzustand des Agenten, die nach dem Rücksetzen eines Schrittes Daten enthalten können, welche sich von den originalen Daten vor der Ausführung des rückzusetzenden Schrittes unterscheiden. Der Zustand der schwach reversiblen Objekte eines Agenten A vor der Ausführung des Schrittes i wird mit $A_{\mathrm{W},i}$ bezeichnet. Der Zustand der schwach reversiblen Objekte eines Agenten A nach dem Rücksetzen des Schrittes i wird mit $\overline{A}_{\mathrm{W},i}$ bezeichnet.

Sammelt beispielsweise ein Agent Daten und speichert diese in einem Vektor in seinem Datenzustand ab, so kann diese Operation einfach dadurch rückgängig gemacht werden, indem der originale Zustand des Vektors wiederhergestellt wird. Datenobjekte, für die dies bei jedem Schritt zutrifft, sind stark reversible Objekte. Deklariert der Agentenentwickler diese Objekte als stark reversible Objekte, so kann das Rücksetzen dieser Objekte automatisch von der Laufzeitumgebung des Agenten durchgeführt werden, ohne daß der Agentenentwickler hierfür Kompensationsoperationen bereit stellen muß. Die hierzu notwendige Rücksetzinformation wird von der Laufzeitumgebung in Form von Kopien – auch bezeichnet als *Before Image*, vgl. HÄRDER UND RAHM (1999) – der stark reversiblen Objekte im Rücksetz-Log (vgl. Abschnitt 5.4.2) gespeichert.

Ein Beispiel für schwach reversible Objekte wurde bereits in Abschnitt 5.2 beschrieben. Bezahlt ein Agent eingekaufte Ware beim Händler mittels elektronischem Geld welches auf dem in CHAUM (1985) beschriebenen Algorithmus basiert, so löst der Händler dieses Geld i.a. noch während des Bezahlvorganges bei der Bank ein. Soll diese Bezahlung später rückgängig gemacht werden, so erhält der Agent deshalb vom Händler nicht dieselben digitalen Münzen zurück, die zur Bezahlung verwendet wurden. Erhebt der Händler eine Gebühr für das Rücksetzen der Kauftransaktion, erhält der Agent weniger Geld zurück. Im Extremfall bekommt der Agent sogar nur eine Gutschrift anstatt des Geldes. In allen diesen Fällen enthalten jene Datenobjekte

im Agent, die die elektronische Geldbörse verwalten, nach dem Rücksetzen der Transaktion nicht die ursprünglichen Daten. Sie sind somit schwach reversible Objekte. Es liegt in der Verantwortung des Entwicklers eines Agenten, daß er die zum Rücksetzen dieser Objekte notwendigen Kompensationsoperationen inklusive Rücksetzinformationen zur Verfügung stellt, welche ebenfalls im Rücksetz-Log gespeichert werden. Details hierzu findet man in den folgenden Abschnitten.

Um den Agent auf jeden beliebigen Zustand $\overline{A}_i$ rücksetzen zu können, ist es notwendig, daß die Laufzeitumgebung bei jeder Migration die zur Wiederherstellung der stark reversiblen Objekte notwendige Information im Rücksetz-Log abspeichert. Abhängig davon, wie die Rücksetzinformation im Log gespeichert wird und wieviel stark reversible Objekte im Agent enthalten sind, können hierbei sehr viele Daten anfallen. Je nach Anwendung ist es jedoch eventuell gar nicht notwendig, auf jeden Zustand $\overline{A}_i$ zurücksetzen zu können. Bilden beispielsweise mehrere nacheinander ausgeführte Schritte eine logische Einheit, so ist möglicherweise nur das vollständige Rücksetzen aller dieser Schritte sinnvoll; Zustände zwischen diesen Schritten müssen nicht wiederhergestellt werden können. Aus diesem Grunde wird zusätzlich noch das Konzept des Agenten-Rücksetzpunktes eingeführt:

Definition 5-7: Agenten-Rücksetzpunkt (engl.: agent savepoint)

> Ein Agenten-Rücksetzpunkt ist ein Zustand in der Ausführung eines Agenten, auf den der Agent zurückgesetzt werden kann. Agenten-Rücksetzpunkte können sich nur zwischen den Schritten der Agentenausführung befinden (vgl. Abschnitt 5.1).

Soll es möglich sein, einen Agent auf den Zustand $\overline{A}_i$ (d.h. auf den Zustand vor der Ausführung des Schrittes *i*) zurückzusetzen, muß vom Code des Agenten am Schluß des Schrittes *i*-1 ein Rücksetzpunkt etabliert werden. Der Rücksetzpunkt befindet sich in diesem Falle dann zwischen Schritt *i*-1 und Schritt *i*. Nur wenn ein solcher Agenten-Rücksetzpunkt veranlaßt wird, muß die zum Rücksetzen der stark reversiblen Objekte notwendige Information ins Rücksetz-Log geschrieben werden. Umgekehrt heißt dies, daß beim Rücksetzen eines Agenten der Zustand von dessen stark reversiblen Objekten jeweils nur beim Erreichen eines Rücksetzpunktes wieder hergestellt wird.

Die erweiterte Rücksetzfunktion $\hat{r}_i$ berücksichtigt dies, indem sich die stark reversiblen Objekte nur dann ändern, wenn ein Rücksetzpunkt erreicht wird. Wird kein Rücksetzpunkt erreicht, ändern sich die stark reversiblen Objekte des Agenten nicht:

$$\hat{r}_i\colon (A_{W,\,i+1}, A_{S,i+k}, R_i^3, I_i) \mapsto \begin{cases} (\overline{A}_{W,\,i}, A_{S,\,i}, R_i^4) & \text{falls } A_i \text{ Rücksetzpunkt} \\ (\overline{A}_{W,\,i}, A_{S,\,i+k}, R_i^4) & \text{sonst} \end{cases} \quad \text{für } k \ge 1 \tag{5-6}$$

beziehungsweise

$$\hat{r}_i\colon (\bar{A}_{W,\,i+1}, A_{S,i+k}, R_i^3, I_i) \mapsto \begin{cases} (\bar{A}_{W,\,i}, A_{S,\,i}, R_i^4) & \text{falls } A_i \text{ Rücksetzpunkt} \\ (\bar{A}_{W,\,i}, A_{S,\,i+k}, R_i^4) & \text{sonst} \end{cases} \quad \text{für } k \geq 1 \tag{5-7}$$

wobei A_{i+k} der "letzte" Rücksetzpunkt (bzw. der Ausgangspunkt des Rücksetzens) ist. Die Rücksetzfunktion setzt sich hierbei aus der (den) vom Agentenentwickler bereitgestellten Kompensationsoperation(en) zum Rücksetzen der schwach reversiblen Objekte und – falls ein Rücksetzpunkt erreicht wird – dem Rücksetzen der stark reversiblen Objekte (durch die Laufzeitumgebung) zusammen.

Die Funktion $\hat{r}_i$ kann nur dann isoliert ausgeführt werden, wenn A_i ein Rücksetzpunkt ist und der Schritt i der einzige bereits abgeschlossene Schritt ist, der zurückgesetzt werden muß. In diesem Falle gilt für den Eingabeparameter $A_{S,i+k}$, daß k=1 ist (d.h. der Zustand der stark reversiblen Objekte nach der Ausführung von Schritt i). In allen anderen Fällen wird $\hat{r}_i$ nur als Teil des Rücksetzens mehrerer Schritte ausgeführt:

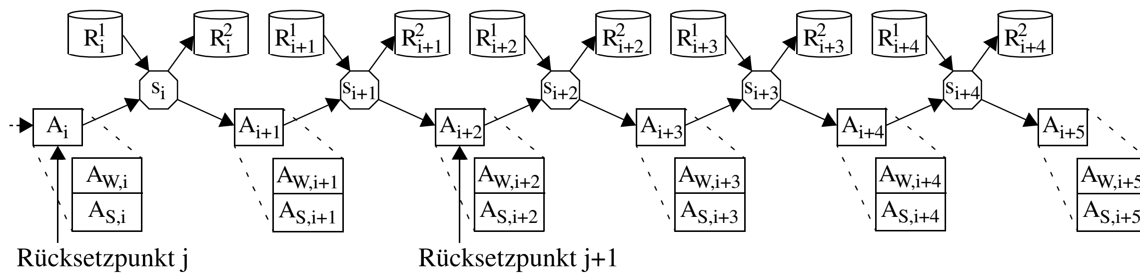
$$\hat{r}_{i,n}\colon (A_{W,\,i+1}, A_{S,\,i+1}, R_i^3, R_{i-1}^3, R_{i-2}^3, \ldots, R_{i-n+1}^3, I_i, I_{i-1}, \ldots, I_{i-n+1}) \mapsto (\bar{A}_{W,\,i-n+1}, A_{S,\,i-n+1}, R_i^4, R_{i-1}^4, R_{i-2}^4, \ldots, R_{i-n+1}^4) \tag{5-8}$$

beziehungsweise

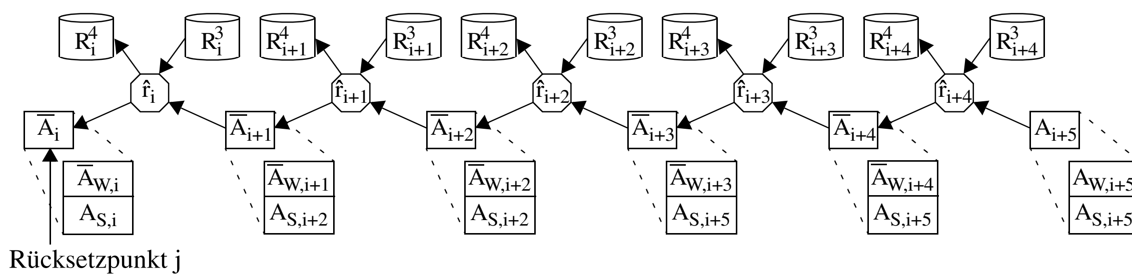
$$\hat{r}_{i,n}\colon (\bar{A}_{W,\,i+1}, A_{S,\,i+1}, R_i^3, R_{i-1}^3, R_{i-2}^3, \ldots, R_{i-n+1}^3, I_i, I_{i-1}, \ldots, I_{i-n+1}) \mapsto (\bar{A}_{W,\,i-n+1}, A_{S,\,i-n+1}, R_i^4, R_{i-1}^4, R_{i-2}^4, \ldots, R_{i-n+1}^4) \tag{5-9}$$

Hierbei beschreibt $\hat{r}_{i,\,n}$ das partielle Rücksetzen der vollständig ausgeführten Schritte i, i-1,..., i-n+1 auf den Rücksetzpunkt zwischen den Schritten i-n und i-n+1. Auch hier ist diese Rücksetzfunktion eine Verkettung der Rücksetzfunktionen der einzelnen Schritte. Abbildung 5-2 illustriert dies anhand der Ausführung der Schritte i, i+1,..., i+4 eines Agenten mittels der Schrittfunktion $s_{i,5}$ (Erweiterung der Schrittfunktion s_i aus Abschnitt 2.2 um die Ausführung mehrerer Schritte) und dem Rücksetzen dieser Schritte mittels der Rücksetzfunktion $\hat{r}_{i+4,\,5}$ (die bei der Ausführung des Agenten erzeugten und beim Rücksetzen verwendeten Rücksetzinformationen

Ausführung des Agenten mittels $s_{i,5}$:



Rücksetzen mittels $\hat{r}_{i+4,5}$:



A_i: Agentenzustand vor Ausführung von Schritt i	R_i^1: Zustand der Ressourcen von Knoten i vor Ausführung des Schrittes i
$\overline{A}_i$: Agentenzustand nach Rücksetzen von Schritt i	R_i^2: Zustand der Ressourcen von Knoten i nach Ausführung des Schrittes i
$A_{W,i}$: schwach reversible Objekte des Agenten vor Ausführung von Schritt i	R_i^3: Zustand der Ressourcen von Knoten i vor Rücksetzen des Schrittes i
$\overline{A}_{W,i}$: schwach reversible Objekte des Agenten nach Rücksetzen von Schritt i	R_i^4: Zustand der Ressourcen von Knoten i nach Rücksetzen des Schrittes i
$A_{S,i}$: stark reversible Objekte des Agenten vor Ausführung / nach Rücksetzen von Schritt i	$\hat{r}_i$: Rücksetzfunktion für Schritt i
s_i: Schrittfunktion für Schritt i	

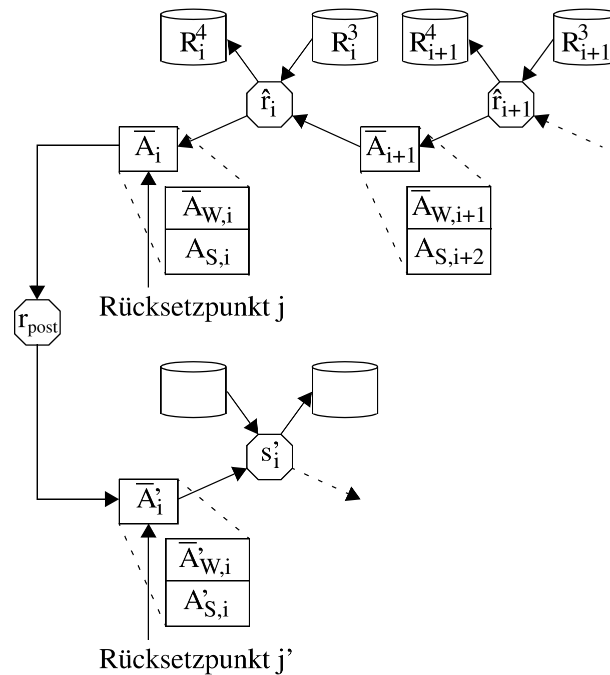
Abbildung 5-2. Ausführen und Rücksetzen eines Agenten

wurden zur Wahrung der Übersichtlichkeit weggelassen). Bei der Ausführung des Agenten wurde vor den Schritten *i* und *i*+2 jeweils ein Rücksetzpunkt gesetzt. In der Abbildung ist zu erkennen, daß, wie oben beschrieben, der Zustand der stark reversiblen Objekte nur beim Erreichen der Rücksetzpunkte durch die Rücksetzfunktionen der Schritte *i*+2 und *i* ($\hat{r}_{i+2}$ und $\hat{r}_i$)aktualisiert wird, die Rücksetzfunktionen der anderen Schritte ändern an den stark reversiblen Objekten nichts. Dies hat zur Folge, daß sich die stark reversiblen Objekte des Agenten während der Ausführung der Rücksetzfunktion von Schritt *i*+3 noch im selben Zustand befinden wie nach der Ausführung des Schrittes *i*+4. Die stark reversiblen Objekte enthalten also während der Ausführung von Rücksetzfunktionen, d.h. während der Ausführung der vom Agentenentwickler zur Verfügung gestellten Kompensationsoperation(en), potentiell “zukünftige” Daten. Aus diesem Grund können die stark reversiblen Objekte von den vom Agentenentwickler zur Verfügung gestellten Kompensationsoperationen nicht verwendet werden. Da durch das Rücksetzen die stark

reversiblen Objekte im Originalzustand wieder hergestellt werden, bleibt der Rücksetzpunkt (im Beispiel Rücksetzpunkt j), auf welchen die Ausführung des Agenten zurückgesetzt wurde, erhalten.

Vorausgesetzt der Entwickler eines Agenten stellt korrekte Operationen zur Kompensation der schwach reversiblen Objekte zur Verfügung, stellt die Funktion $\hat{r}_{i,n}$ einen Zustand $\overline{A}_{i-n+1}$ her, welcher ein im Sinne der Anwendung zum Zustand A_{i-n+1} semantisch äquivalenter Zustand ist. Im Idealfall wurden während der Ausführung der Schritte i-n+1, i-n,..., i nur stark reversible Objekte geändert. In diesem Fall wäre $\overline{A}_{i-n+1}=A_{i-n+1}$. Ein bisher nicht betrachtetes Problem tritt allerdings dann auf, wenn der Agent nach dem (partiellen) Rücksetzen seine Arbeit basierend auf dem wiederhergestellten, zum Ausgangszustand äquivalenten Zustand, fortsetzt. Da die Reiseroute des Agenten beim Rücksetzen wieder in ihren ursprünglichen Zustand zurückgesetzt wird, kann es passieren, daß der Agent nach dem Rücksetzen dieselben Schritte ausführt wie zuvor. Dies ist kein Problem wenn die Ausführung des Agenten beispielsweise wegen der temporären Nichtverfügbarkeit einer Ressource zurückgesetzt wurde. Hat jedoch der Agent das Rücksetzen beispielsweise veranlaßt, weil die von ihm momentan durchgeführte Strategie nicht zum Ziel führt, dann müssen dem Agent nach dem Rücksetzen Informationen zur Verfügung gestellt werden, die es ihm erlauben, seine Strategie zu ändern – beispielsweise durch Änderungen in seinem Datenzustand oder durch Änderungen in der Reiseroute. Aus diesem Grunde wird zusätzlich nach dem Erreichen des Rücksetzpunktes eine Wiederaufnahme-Funktion r_{post} ausgeführt, in der die notwendigen Änderungen durchgeführt werden können.

Da die gewünschte Funktionalität von r_{post} im allgemeinen von der Situation, in der das Rücksetzen initiiert wird, abhängt, kann r_{post} inklusive ihrer Parameter vom Agent bei der Initiierung des Rücksetzens angegeben werden (vgl. auch nächster Abschnitt). Abbildung 5-3 zeigt, wie nach dem Rücksetzvorgang aus Abbildung 5-2 die normale Abarbeitung des Agenten wieder aufgenommen wird, indem zuerst die Funktion r_{post} ausgeführt wird und erst dann der nächste Schritt des Agenten ausgeführt wird. Die Funktion r_{post} kann dabei als eine Art Schritt interpretiert werden, der nach Schritt i-1 in die Reiseroute eingefügt wird, wobei dieser Schritt wieder aus der Reiseroute entnommen werden muß, falls später auf einen noch früheren Rücksetzpunkt zurückgesetzt werden soll. Die Ausführung von r_{post} kann dabei auf einem beliebigen Knoten erfolgen. Da r_{post} konzeptionell nur dafür zuständig ist, den Zustand des Agenten, speziell auch dessen Reiseroute, so zu ändern, daß der Grund des Rücksetzens bei der weiteren Verarbeitung des Agenten berücksichtigt werden kann, sollte durch r_{post} kein Ressourcenzustand geändert werden. Da r_{post} jedoch sowohl stark reversible als auch schwach reversible Objekte ändern kann, müssen für den Fall, daß später auf einen noch früheren Rücksetzpunkt zurückgesetzt werden soll, sowieso Kompensationsoperationen für die Änderungen der schwach reversiblen Objekte in r_{post} angegeben werden. Somit stehen aus technischer Sicht Änderungen von Ressourcen durch r_{post} nichts im Wege, solange auch hierfür die entsprechenden Kompensationsoperationen zur Verfügung gestellt werden. Wie schon weiter oben erwähnt, bleibt der Rücksetzpunkt, auf den ein Agent zurückgesetzt wurde, erhalten – in Abbildung 5-3 beispielsweise



A_i:	Agentenzustand vor Ausführung von Schritt i	R^1_i :	Zustand der Ressourcen von Knoten i vor Ausführung des Schrittes i
$\overline{A}_i$:	Agentenzustand nach Rücksetzen von Schritt i	R^2_i :	Zustand der Ressourcen von Knoten i nach Ausführung des Schrittes i
$A_{W,i}$:	schwach reversible Objekte des Agenten vor Ausführung von Schritt i	R^3_i :	Zustand der Ressourcen von Knoten i vor Rücksetzen des Schrittes i
$\overline{A}_{W,i}$:	schwach reversible Objekte des Agenten nach Rücksetzen von Schritt i	R^4_i :	Zustand der Ressourcen von Knoten i nach Rücksetzen des Schrittes i
$A_{S,i}$:	stark reversible Objekte des Agenten vor Ausführung / nach Rücksetzen von Schritt i	$\hat{r}_i$:	Rücksetzfunktion für Schritt i
s_i:	Schrittfunktion für Schritt i		

Abbildung 5-3. Fortsetzen der Ausführung eines Agenten nach Rücksetzen

der Rücksetzpunkt *j*. Da die weitere Ausführung des Agenten jedoch auf dem durch r_{post} hergestellten Zustand basiert, ist es sehr wahrscheinlich, daß im Falle eines Rücksetzens nicht auf den originalen Rücksetzpunkt *j* zurückgesetzt werden soll sondern auf den durch r_{post} hergestellten Zustand. Um dies zu ermöglichen, wird nach der Ausführung von r_{post} automatisch ein weiterer Rücksetzpunkt eingefügt (Rücksetzpunkt *j*’ in Abbildung 5-3).

Im folgenden Abschnitt wird ein auf dem hier vorgestellten, erweiterten Agentenmodell basierender Mechanismus präsentiert, welcher auch für das Rücksetzen von Agenten eine genau-einmal Semantik garantiert. Weitere Verfeinerungen und Erweiterungen des Modells werden anschließend verwendet, um den vorgestellten Mechanismus zu optimieren.

5.4 Basismechanismus

Der in diesem Abschnitt vorgestellte Mechanismus realisiert das partielle Rücksetzen der Ausführung von Agenten, welche mittels des in Abschnitt 4.3 vorgestellten Basisprotokolls zur genau-einmal Ausführung mobiler Agenten ausgeführt werden. Ziel des Mechanismus ist hierbei, sowohl die im vorherigen Abschnitt beschriebene Semantik des partiellen Rücksetzens zu bieten als auch – analog zur genau-einmal Ausführung des Agenten – das Rücksetzen des Agenten ebenfalls genau einmal auszuführen.

5.4.1 Überblick

Die in Abschnitt 4.3 vorgestellte Idee zur Realisierung der genau-einmal Ausführung eines Agenten läßt sich auf das partielle Rücksetzen von Agenten übertragen. Bei der Ausführung eines Schrittes eines Agenten werden innerhalb einer Transaktion der Agent aus der Eingangswarteschlange des ausführenden Knotens entnommen, die Schrittfunktion auf dem Agenten (und den Ressourcen) ausgeführt und dann der Agent in die Eingangswarteschlange des Knotens, auf dem der nächste Schritt ausgeführt werden soll, geschrieben. Wie in Abschnitt 4.3 ausgeführt wird dadurch garantiert, daß der Agent nicht verloren gehen kann und daß alle Aktionen des Agenten (logisch gesehen) exakt einmal ausgeführt werden. Das Rücksetzen eines Schrittes S des Agenten unterscheidet sich nicht wesentlich von der Ausführung dieses Schrittes S. Da beim Rücksetzen auch der Zustand der Ressourcen des Knotens N, auf dem der Schritt S ausgeführt wurde, zurückgesetzt werden muß, muß auch die Rücksetzfunktion des Agenten auf diesem Knoten N ausgeführt werden. Es bietet sich daher an, beim Rücksetzen eines Schrittes des Agenten ebenso zu verfahren wie bei der Ausführung des Schrittes: Innerhalb einer Rücksetztransaktion wird der Agent aus der Eingangswarteschlange des Knotens, auf dem die Rücksetzfunktion ausgeführt werden muß, entnommen, die Rücksetzfunktion wird ausgeführt und dann wird der Agent in die Eingangswarteschlange des Knotens geschrieben, auf dem die nächste Rücksetzfunktion ausgeführt werden muß.

Abbildung 5-4 zeigt die Ausführung der Schritte i, i+1, i+2 und i+3 eines Agenten A und das Rücksetzen dieser Schritte mittels des Basismechanismus' zum Rücksetzen. Die Schritte i, i+1 und i+2 werden komplett ausgeführt. Während des Schrittes i+3 beschließt der Agent, seinen Zustand auf den Rücksetzpunkt j, welcher direkt vor Schritt i gesetzt wurde, zurückzusetzen. Der Basismechanismus bekommt hierzu als Parameter das Ziel des Rücksetzens (Rücksetzpunkt j) und die nach dem Rücksetzen auszuführende Funktion r_{post} übergeben (siehe weiter unten). Um die durch die Ausführung des Schrittes i+3 an Agent und Ressourcen verursachten Änderungen zurückzusetzen, reicht es aus, die Schritt-Transaktion T_{i+3} abzubrechen. Dies bewirkt, daß der Ausgangszustand R^1_{i+3} der Ressourcen des Knotens K_{i+3} wiederhergestellt wird und daß sich der Agent wieder im Zustand A_{i+3} in der Eingangswarteschlange Q_{i+3} des Knotens K_{i+3} befindet. Um den Schritt i+2 auf dem Knoten K_{i+2} zurücksetzen zu können, muß sich der Agent

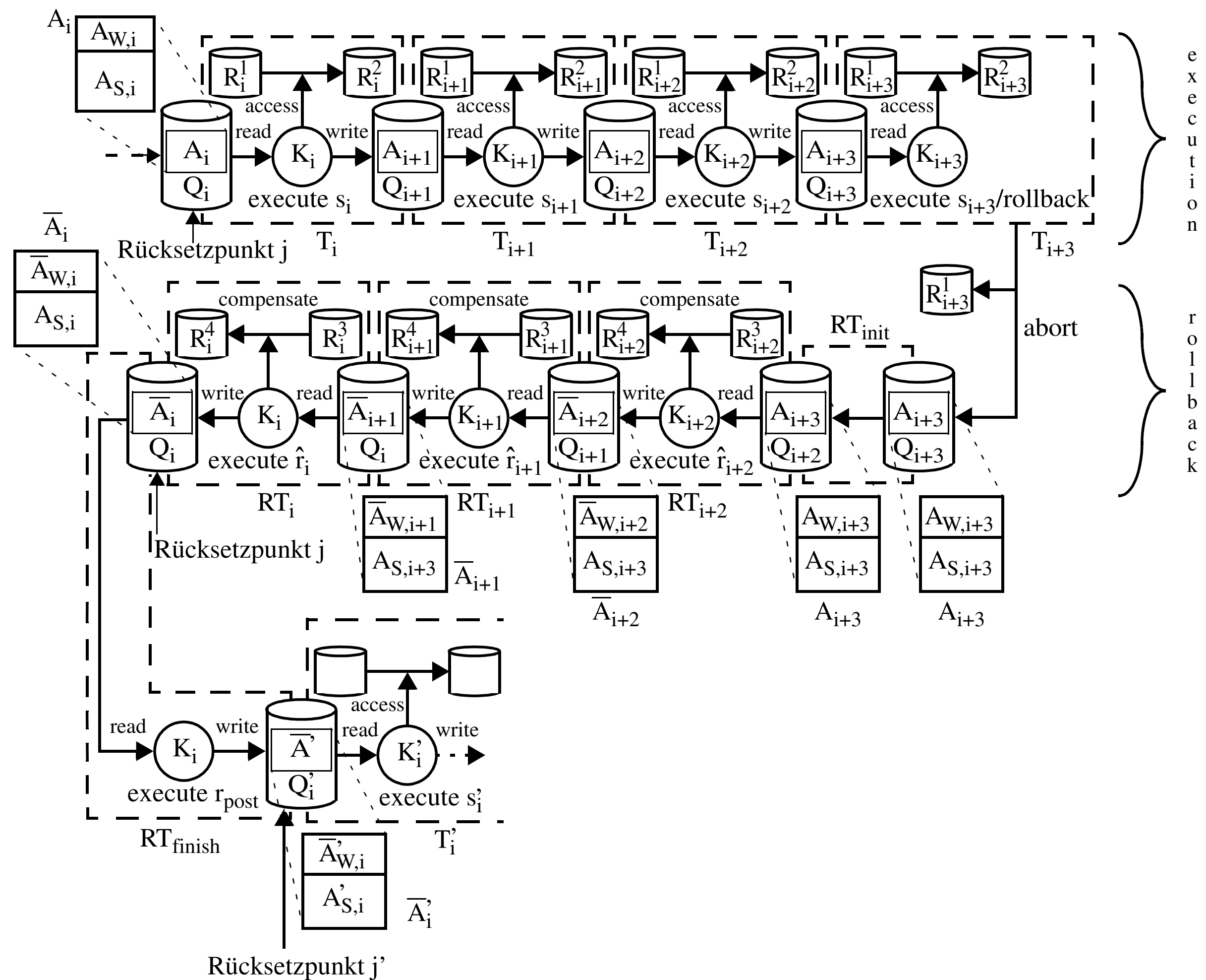


Abbildung 5-4. Rücksetzen mittels des Basismechanismus'

in dessen Eingangswarteschlange Q_{i+2} befinden. Daher wird in einer Transaktion RT_{init} der Agent von Q_{i+3} nach Q_{i+2} verschoben. Jetzt beginnt erst die eigentliche Kompensation der vollständig ausgeführten Schritte. Innerhalb einer Rücksetztransaktion RT_k wird dann jeweils der Agent aus der Eingangswarteschlange Q_k gelesen, der Schritt k mittels $\hat{r}_k$ zurückgesetzt und der Agent in $Q_{k\text{-}1}$ (Q_i für $k=i$) geschrieben (in der Reihenfolge $k=i+2$, $i+1$, i). Erwähnenswert ist hierbei nochmal, daß beim Rücksetzen der Schritte $i+2$ und $i+1$ nur die schwach reversiblen Objekte des Agenten und die Ressourcen kompensiert werden. Erst beim Rücksetzen des Schrittes i (und damit beim Erreichen eines Rücksetzpunktes) werden zusätzlich auch die stark reversiblen Objekte des Agenten zurückgesetzt. Sobald der gewünschte Rücksetzpunkt nach Rücksetzen des Schrittes i erreicht wird, wird der Agent nicht in $Q_{i\text{-}1}$ sondern in Q_i geschrieben. Danach wird in einer weiteren Transaktion die Funktion r_{post} ausgeführt, mittels der der Agent durch das Rücksetzen notwendig gewordene Änderungen in seinem Zustand durchführen kann (z.B. die Reiseroute abändern). Dies geschieht ebenfalls, indem innerhalb einer Transaktion der Agent aus Q_i gelesen wird, die Funktion r_{post} ausgeführt wird, und schließlich der Agent in die Eingangswarteschlange jenes Knoten K_i'geschrieben wird, auf dem der nächste Schritt auszu-

führen ist. Wurde in r_{post} die Reiseroute des Agenten geändert, so ist i.a. $K_i \neq K_i$'. Wie in Abschnitt 5.3 ausgeführt wird hierbei auch noch der Rücksetzpunkt *j*' geschrieben. Damit ist das Rücksetzen des Agenten beendet und die reguläre Ausführung des Agenten analog Abschnitt 4.3 kann wieder aufgenommen werden.

5.4.2 Logging

Die für das Rücksetzen der einzelnen Schritte eines Agenten notwendigen Daten werden im *Agenten-Rücksetz-Log* (engl.: *agent rollback log*) abgelegt. Hierzu gehört neben der zum Rücksetzen von Ressourcen- und Agentenzustand notwendigen Rücksetzinformation *I* auch die Information, welche Kompensationsoperationen für die Kompensation eines Schrittes ausgeführt werden müssen. Das Rücksetz-Log wird an den Agenten angehängt und migriert daher mit dem Agenten von Knoten zu Knoten. Da das Rücksetz-Log nur Daten enthält, welche zum Rücksetzen von erfolgreich abgeschlossenen (d.h. mit Commit beendeten) Schritten notwendig sind, reicht es aus, das Rücksetz-Log am Ende einer Transaktion (sowohl Schritt- als auch Rücksetz-Transaktion) persistent zu machen, d.h. mit dem Agenten in die Eingangswarteschlange des nächsten Knotens zu schreiben.

Das Anhängen des Rücksetz-Logs an den Agenten hat zwei Vorteile. Erstens ist es nach dem Beenden der Ausführung eines Agenten nicht notwendig, globale Aktionen zum Löschen des Rücksetz-Logs auf den durch den Agenten besuchten Knoten durchzuführen. Zweitens ist das Rücksetz-Log immer genau dann verfügbar, wenn auch der Agent verfügbar ist. Dadurch kann der Agent immer dann zurückgesetzt werden, wenn die rückzusetzenden Ressourcen verfügbar sind. Dieser zweite Vorteil ist zwar im Kontext des Basismechanismus nicht relevant, gewinnt jedoch im Zuge der weiter hinten vorgestellten Optimierungen wesentlich an Bedeutung. Der Nachteil des Anhängens des Rücksetz-Logs an den Agenten ist offensichtlich, daß die Größe des Rücksetz-Logs und damit die Menge der zu migrierenden Daten im Verlauf der Ausführung des Agenten stetig zunimmt. Wie stark der Zuwachs der Größe des Rücksetz-Logs ausfällt, hängt hierbei vor allem von der Anwendung ab.

Im hier vorgestellten Mechanismus wird eine Mischung aus *physischem* und *logischem Logging* (engl.: *physical* and *logical logging*, vgl. auch HÄRDER UND REUTER (1983) bzw. HÄRDER UND RAHM (1999)) verwendet. Für die stark reversiblen Objekte wird physisches Logging verwendet. Hier kann entweder ein komplettes *Abbild* (engl.: *image*) dieser Objekte ins Rücksetz-Log geschrieben werden (*Zustands-Logging*, engl.: *state logging*) oder nur die Zustands-Differenzen dieser Objekte zwischen zwei aufeinanderfolgenden Rücksetzpunkten (*Übergangs-Logging*, engl.: *transition logging*). Diese Informationen werden als Bestandteil eines *Rücksetzpunkteintrages* (*RP*, engl.: *savepoint entry*) in das Log geschrieben. Ein solcher Rücksetzpunkteintrag wird genau dann in das Rücksetz-Log geschrieben, wenn vom Agent ein Rücksetzpunkt initiiert wird. Neben dem Abbild der stark reversiblen Objekte enthält ein Rücksetzpunkteintrag zusätzlich noch einen (eindeutigen) Bezeichner. Zur Vereinfachung wird im folgenden davon ausge-

gangen, daß Zustands-Logging verwendet wird. Das Format, in dem das Abbild der stark reversiblen Objekte im Rücksetz-Log gespeichert wird, hängt von der zur Programmierung des Agenten verwendeten Programmiersprache ab.

Für die Kompensation der schwach reversiblen Objekte und der Ressourcenzustände wird logisches Logging verwendet. Hierbei werden die Kompensationsoperationen und deren Parameter in das Rücksetz-Log geschrieben. Eine Kompensationsoperation mit den zugehörigen Parametern wird als *Operationseintrag* (*OE*, engl.: *operation entry*) bezeichnet. Die Anzahl der zur Kompensation eines Schrittes notwendigen Kompensationsoperationen (und damit die Anzahl der im Rücksetz-Log enthaltenen Operationseinträge für einen Schritt) kann zwischen einer (komplexen) Operation, welche die Auswirkungen des gesamten Schrittes kompensiert, und beliebig vielen Operationen liegen. Die Agenten-Plattform muß Operationen zur Verfügung stellen, mit denen Operationseinträge an das Rücksetz-Log angehängt werden können. Existieren für einen Schritt mehrere Operationseinträge, so werden beim Rücksetzen die Operationseinträge (genauer: die in den Operationseinträgen spezifizierten Kompensationsoperationen) in umgekehrter Reihenfolge, d.h. beginnend mit dem zuletzt angehängten Eintrag, abgearbeitet. Hintergrund ist, daß bei der Kompensation im allgemeinen "von hinten nach vorne" vorgegangen wird, d.h. die zuletzt ausgeführte Operation muß zuerst kompensiert werden, dann die vorletzte ausgeführte Operation und so weiter. Wird beispielsweise zuerst ein Betrag von einem Konto *A* auf ein Konto *B* überwiesen und Konto *B* anschließend zur Bezahlung beim Einkaufen verwendet, dann muß zuerst der Einkauf kompensiert werden, bevor die Rücküberweisung erfolgen kann. Wird nun das Rücksetz-Log wie oben beschrieben abgearbeitet, dann kann dadurch direkt jeweils nach Ausführung der zu kompensierenden Operation die Kompensationsoperation ins Rücksetz-Log geschrieben werden.

```
abstract class CompensationObject{
    abstract void compensate();
}

class CompensateTransfer
  extends CompensationObject{
    Bank theBank;
    Account account1, account2;
    Amount amount;
    void compensate(){
        theBank.transfer(account2,
          account1,amount)
}   }
```

Abbildung 5-5. CompensationObject

Das Format, in dem die Kompensationsoperationen und deren Parameter im Rücksetz-Log abgespeichert werden, hängt von der zur Programmierung des Agenten verwendeten Programmiersprache ab. Wird eine objektorientierte Sprache wie Java verwendet, so kann beispielsweise der Vererbungsmechanismus und die damit einhergehende Polymorphie als Lösungsansatz dienen. Die Idee hierbei ist, die Kompensationsoperation in einem Objekt als Methode des Objekts und die Parameter der Kompensationsoperation als Attribute des Objektes zu realisieren. Hierbei dient eine abstrakte Oberklasse *CompensationObject* mit einer abstrakten (d.h. nicht implementierten) Methode *compensate()* als Grundbaustein. Für jede Kompensationsoperation muß dann eine Unterklasse von *CompensationObject* implementiert werden, in der die Kompensationsoperation von der *compensate()*-Methode realisiert wird und deren Attribute als Parameter der Kompensationsoperation dienen.

Abbildung 5-5 demonstriert die Idee anhand der Rücksetzoperation für eine Überweisung zwischen zwei Konten, welche in einem Objekt der Klasse *CompensateTransfer* gekapselt wird. Die für die Rücksetzoperation notwendigen Parameter Bank (*theBank*), Ursprungskonto der Überweisung (*account1*), Zielkonto der Überweisung (*account2*) und der überwiesene Betrag (*amount*) sind als Attribute der Klasse im Objekt enthalten. Die Methode *compensate()* implementiert die Kompensationsoperation – in diesem Falle die Rücküberweisung. Bei dieser Realisierung enthält ein Operationseintrag des Rücksetz-Logs einfach ein Objekt einer Unterklasse der Klasse *CompensationObject* – im Beispiel also ein Objekt der Klasse *CompensateTransfer*, welches die zur Kompensation notwendigen Daten und die Kompensationsoperation enthält. Zur Kompensation muß nur dieses Objekt aus dem Rücksetz-Log gelesen werden und die *compensate()*-Methode auf diesem Objekt aufgerufen werden.

Zusätzlich zu diesen Eintragstypen enthält das Rücksetz-Log noch Einträge für den Beginn und das Ende der Abarbeitung eines Schrittes (*begin-of-step* (*BS*), *end-of-step* (*ES*)). Diese Einträge enthalten unter anderem den Bezeichner des Knotens, auf dem der Schritt ausgeführt wurde. Mögliche weitere Inhalte dieser Einträge werden weiter unten diskutiert.

Für den Fall daß ein Schritt nicht rücksetzbar ist, da er, entgegen der in Abschnitt 5.2 getroffenen Annahme, nicht rücksetzbare Operationen durchführt, können alle bisherigen Einträge des Rücksetz-Log verworfen und nach der Ausführung des Schrittes ein Rücksetzpunkteintrag ins Log geschrieben werden. In diesem Falle kann die Ausführung des Agenten nicht mehr auf einen früheren Zustand zurückgesetzt werden. Dieser Fall wird, da nicht mit den getroffenen Annahmen konform, im weiteren Verlauf nicht mehr betrachtet.

Abbildung 5-6 zeigt einen Auszug aus einem Rücksetz-Log. Er enthält die Einträge $OE_{n,1}$, $OE_{n,2}$, ..., $OE_{n,p}$ zur Kompensation des *n*-ten Schrittes des Agenten inklusive der Einträge für den Beginn und das Ende der Abarbeitung dieses Schrittes sowie den *k*-ten Rücksetzpunkteintrag, welcher sich direkt vor Schritt *n* befindet. Um den Agenten auf diesen *k*-ten Rücksetzpunkt zurückzusetzen, muß das Rücksetz-Log beginnend von seinem aktuellsten (d.h. zuletzt in das Log geschriebenen) Eintrag bis zum Rücksetzpunkteintrag abgearbeitet werden. Wird beispielsweise das Rücksetzen des Agenten in Schritt *n*+1 initiiert, dann ist ES_n der aktuellste Eintrag, da alle Änderungen des Schrittes *n*+1 inklusive der Änderungen am Rücksetz-Log durch

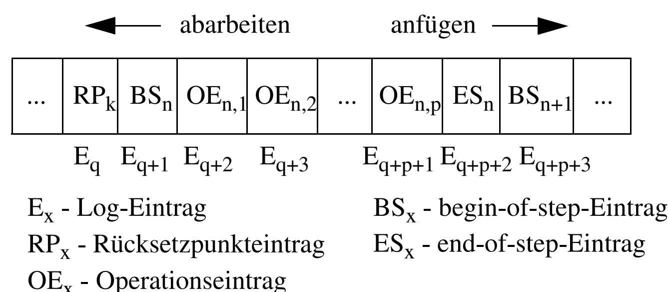


Abbildung 5-6. Beispiel-Log

den Abbruch der Schritt-Transaktion des Schrittes n+1 verworfen werden. In diesem Falle muß nur Schritt n kompensiert werden, was durch sukzessive Ausführung der Operationseinträge von Schritt n in der Reihenfolge $OE_{n,p}$, $OE_{n,p\text{-}1}$, ..., $OE_{n,1}$ (genauer: Ausführung der in diesen Einträgen spezifizierten Kompensationsoperationen) geschieht. Die Details des Rücksetzmechanismus zeigt der folgende Abschnitt.

5.4.3 Algorithmus

Nachdem in Abschnitt 5.4.1 bereits die wesentlichen Grundzüge des Algorithmus und in Abschnitt 5.4.2 das Logging vorgestellt wurden, klärt dieser Abschnitt die noch verbleibenden Details und präsentiert in Pseudo-Code-Form den Basismechanismus zum Rücksetzen der Ausführung mobiler Agenten.

Wie schon in Abschnitt 5.1 ausgeführt, wird der Rücksetzalgorithmus zuerst als Erweiterung des Basisprotokolles zur genau-einmal Ausführung mobiler Agenten (Algorithmus 4-1) konzipiert, bevor er dann im folgenden Abschnitt in das blockierungsfreie Protokoll integriert wird. Da zum Rücksetzen des Agenten notwendige Informationen allerdings schon während der Ausführung des Agenten erzeugt und im Rücksetz-Log gespeichert werden, müssen auch Änderungen am Basisprotokoll zur Agentenausführung vorgenommen werden. Algorithmus 5-1 zeigt (in Fettschrift) die notwendigen Modifikationen des Basisprotokolles. Neben der Erweiterung des Protokolles um Lesen bzw. Schreiben des Logs von der bzw. in die Eingangswarteschlange der Knoten sind dies vor allem das Schreiben der Einträge für den Beginn und das Ende der Ausführung eines Schrittes sowie – falls notwendig – das Schreiben eines Rücksetzpunkteintrages. Für das Schreiben der Operationseinträge in das Log ist der Agent selbst zuständig. Zusätzlich

```
ForEach (Agent a, Log log) in localNodeInputQueue q{
    Begin Transaction // step transaction
    q.ReadAndDestroy(Agent a, Log log)
    log.insertBeginOfStepEntry(idOfLocalNodeInputQueue)
    safepointId = Execute(Agent a)
    log.insertEndOfStepEntry(idOfLocalNodeInputQueue)
    if (savepointID≠null){// agent initiated savepoint
        log.writeSavepointEntry(savepointID)
    }
    nextPossibleSteps = QueryItinerary()
    if (nextPossibleSteps≠∅){
        // execution not yet finished
        nextStep = ChooseOneOf(nextPossibleSteps)
        Write(Agent a, Log log) to NodeInputQueue of node on which nextStep takes place
        if not successful then Abort Transaction
    }
    Commit Transaction
}
```

Algorithmus 5-1. Modifiziertes Basisprotokoll zur genau-einmal Ausführung mobiler Agenten

```
rollback(SpID spID, CompObject r_post){
    Abort Transaction // step transaction
    Begin Transaction // move to correct queue
    localNodeInputQueue.get (Agent a, Log log)
    if (savepoint spID reached){ (1)
        target=localInputQueue
    }else{ (2)
        if (last log entry is savepoint){ (3)
            log.pop() // remove entry
        }
        target = determine from last
            end-of-step entry in log
    }
    Write(a, log, spId, r_post) To target (4)
    Commit Transaction
}
```

a. Start des Rücksetzalgorithmus

```
ForEach (Agent a, Log log, SpID spID,
            ResumeObject r_post) in
            localNodeInputQueue q{ (5)
    Begin Transaction
    q.ReadAndDestroy(a, log, spID, r_post)
    if (savepoint spID not reached){ (6)
        rollbackOneStep(a, log, spID, r_post)
    }else{ (7)
        resumeExecution(a, log, r_post)
    }
    End Transaction
} // Main Loop

rollbackOneStep(Agent a, Log log, SpID spID,
                ResumeObject r_post){
    log.pop() // remove end-of-step (8)
    entry=log.pop()
    while (entry≠begin-of-step){
        entry.compensate() (9)
        entry=log.pop()
    }
    if (last entry in log is savepoint){ (10)
        restore strongly reversible objects
        if (savepoint spID not reached){ (11)
            log.pop() // sp not needed anymore
        }
    }
    if (savepoint spID reached){ (12)
        target=localNodeInputQueue
    }else{ (13)
        target = determine from last
            end-of-step entry in log
    }
    Write(a,log,spID,r_post) To target (14)
}

resumeExecution(Agent a, Log log,
                ResumeObject r_post){
    log.insertBeginOfStepEntry()
    newSpID = r_post.resume() (15)
    log.insertEndOfStepEntry()
    log.writeSavepointEntry(newSpID) (16)
    nextPossibleSteps = QueryItinerary()
    if nextPossibleSteps≠∅ then{ (17)
        // execution not yet finished
        nextStep = ChooseOneOf(
            nextPossibleSteps)
        target = GetInputQueue( node
            on which nextStep takes place)
        Write(Agent a, Log log) To target
    }
}
```

b. Auf jedem Knoten ausgeführter Rücksetzalgorithmus

Algorithmus 5-2. Basisalgorithmus zum Rücksetzen der Agentenausführung

zu den gezeigten Modifikationen muß noch beim Start des Agenten (also vor der Ausführung der ersten Schritt-Transaktion) ein Rücksetzpunkteintrag geschrieben werden, sodaß der Agent auch ganz auf den Anfang seiner Ausführung zurückgesetzt werden kann.

Algorithmus 5-2 realisiert den Basisalgorithmus zum Rücksetzen der Agentenausführung. Beschließt ein Agent, auf einen früheren Punkt zurückzusetzen, ruft er die in Algorithmus 5-2a dargestellte *rollback*(..)-Methode. Diese Methode bekommt als Parameter den Bezeichner *spID* des Rücksetzpunktes, auf den zurückgesetzt werden soll, und die Funktion *r_post*, welche nach dem Erreichen des Ziel-Rücksetzpunktes ausgeführt wird.

Für die Repräsentation von *r_post* wurde hier eine Repräsentation analog zu der in Abschnitt 5.4.3 vorgeschlagenen Repräsentation für Operationseinträge angenommen: Es existiert eine abstrakte Klasse *ResumeObject* mit der (ebenfalls abstrakten) Methode *resume()*. Unterklassen dieser Klasse enthalten in der *resume*()-Methode dann die auszuführende Funktion *r_post*, die Parameter sind als Attribute in der Klasse enthalten.

In der *rollback()*-Methode wird zuerst die aktuelle Schritt-Transaktion abgebrochen. Danach wird der Agent innerhalb einer neuen Transaktion in die Eingangswarteschlange des Knotens geschrieben, auf dem die nächste Aktion durchzuführen ist. Hierzu wird zuerst der ursprüngliche Zustand (d.h. der Zustand vor Ausführung des gerade abgebrochenen Schrittes) inklusive Log aus der Eingangswarteschlange gelesen. Hat der Agent den gewünschten Rücksetzpunkt schon erreicht **(1)**[1] – erkennbar dadurch daß der aktuellste Log-Eintrag ein Rücksetzpunkt mit dem Bezeichner *spID* ist – dann wird der Agent wieder in die lokale Eingangswarteschlange geschrieben **(4)**.

Hat der Agent den gewünschten Rücksetzpunkt noch nicht erreicht **(2)**, dann wird aus dem aktuellsten *end-of-step*-Eintrag des Logs ermittelt, auf welchem Knoten die nächste Rücksetztransaktion auszuführen ist. Hierbei wird zuerst ein eventuell am Ende des Logs stehender Rücksetzpunkteintrag entfernt **(3)**, welcher in diesem Falle nicht abgearbeitet werden muß, da sich die stark reversiblen Objekte schon in dem Zustand befinden, der in diesem Eintrag enthalten ist. Der Agent wird dann in die Eingangswarteschlange des ermittelten Knotens geschrieben **(4)**.

In beiden Fällen (sowohl beim Schreiben in die lokale als auch in die entfernte Warteschlange) wird zusätzlich zu Agent und Log auch der Bezeichner *spID* des Rücksetzpunktes, auf den der Agent zurückgesetzt werden soll, und die Funktion *r_post* mit Parametern geschrieben. Ist die Transaktion, in der dies alles ausgeführt wird, erfolgreich, so steht der Agent zur weiteren Bearbeitung des Rücksetzens in der Warteschlange des Knotens, auf dem die nächste Aktion erfolgen muß. Schlägt die Transaktion jedoch beispielsweise wegen Ausfall des Knotens fehl, so steht immer noch der Agent inklusive Log in der Eingangswarteschlange des lokalen Knotens. Dies ist äquivalent zu der Situation, daß die Schritt-Transaktion schon vor Aufruf von *rollback*(..) abbricht und ist daher ein akzeptabler Zustand. In diesem Falle wird der (abgebrochene) Schritt auf dem Knoten erneut gestartet, kommt (eventuell) erneut zu der Erkenntnis daß zurückgesetzt werden muß und ruft erneut *rollback*(..) auf. Da der Abbruch der Transaktion nur durch Fehler des lokalen Knotens, des Netzwerkes und des Zielknotens geschehen kann und diese Fehler laut dem zugrundeliegenden Fehlermodell nur temporär sind, wird dieser Teil des Rücksetzens letztendlich erfolgreich sein.

Der Hauptteil des Rücksetzens wird von Algorithmus 5-2b durchgeführt. Sobald ein Tupel (*Agent, Log, SpId, ResumeObject*) in der Eingangswarteschlange eines Knotens erscheint, wird dieses innerhalb einer Transaktion aus der Warteschlange entnommen und bearbeitet **(5)**. Hier-

1. Die Zahlen in Klammern beziehen sich auf die Fall-Numerierungen in Algorithmus 5-2

bei sind zwei Fälle zu unterscheiden. Wurde der Ziel-Rücksetzpunkt *spID* noch nicht erreicht – auch hier daran erkennbar ob der aktuellste Log-Eintrag ein Rücksetzpunkt mit dem Bezeichner *spID* ist – dann muß auf diesem Knoten ein Schritt rückgesetzt werden **(6)**, ansonsten muß die Ausführung des Agenten wieder aufgenommen werden **(7)**. Das Rücksetzen eines Schrittes wird von der Methode *rollbackOneStep*() erledigt, das Wiederaufnehmen der Ausführung geschieht in der Methode *resumeExecution*().

Beim Rücksetzen eines Schrittes in *rollbackOneStep*() werden nach dem Entfernen des *end-of-step*-Eintrages des rückzusetzenden Schrittes **(8)** sukzessive die Operationseinträge aus dem Log entnommen und die darin enthaltenen Kompensationsoperationen ausgeführt **(9)**, bis der *begin-of-step*-Eintrag des Schrittes dem Log entnommen wurde. Ist nun der aktuellste Eintrag des Logs ein Rücksetzpunkt **(10)**, dann wird anhand dieses Eintrages der Zustand der stark reversiblen Objekte wieder hergestellt und der Rücksetzpunkteintrag wird entfernt, falls er nicht der Ziel-Rücksetzpunkt des Rücksetzens, d.h. der Rücksetzpunkt mit Bezeichner *spID*, ist **(11)**. Ist der gewünschte Rücksetzpunkt erreicht **(12)**, wird der Agent in die lokale Eingangswarteschlange geschrieben. Wenn nicht **(13)**, wird aus dem aktuellsten Log-Eintrag, welcher in diesem Falle ein *end-of-step*-Eintrag ist, der Knoten ermittelt, auf dem die nächste Rücksetztransaktion stattfinden muß, und der Agent in die dortige Eingangswarteschlange geschrieben. In beiden Fällen wird wieder das Tupel (*Agent, Log, SpID, ResumeObject*) in die Warteschlange geschrieben **(14)**.

Ist die Transaktion erfolgreich, dann wurde durch *rollbackOneStep*() die rückzusetzenden Kompensationsoperationen genau einmal durchgeführt und die Änderungen an Ressourcen (durch Kompensationsoperationen) und Agent (schwach reversible Objekte: durch Kompensationsoperationen; stark reversible Objekte: durch Info aus Rücksetzpunkt, falls vorhanden) sind permanent. Der Agent befindet sich dann in der Eingangsschlange des Knotens, auf dem die nächste Aktion, d.h. entweder eine weitere Kompensation oder die Wiederaufnahme der Ausführung, stattfinden soll. Bricht die Transaktion wegen eines System-Fehlers ab, dann werden die Änderungen der Transaktion automatisch zurückgesetzt und die Rücksetztransaktion wird nach Beheben des Fehlers erneut gestartet. Da die Ausführung der Kompensationsoperationen selbst nach Voraussetzung auf jeden Fall erfolgreich sein muß und Systemfehler nach Voraussetzung nur temporär sind, wird eine Rücksetztransaktion letztendlich erfolgreich durchgeführt werden.

Um die Ausführung eines Agenten wieder aufzunehmen, wird in *resumeExecution()* zuerst die Funktion *r_post* ausgeführt **(15)**. Da hierfür, wie in Abschnitt 5.3 ausgeführt, auch Kompensationsoperationen zur Verfügung gestellt werden müssen, werden entsprechend ein *begin-of-step*-Eintrag und ein *end-of-step*-Eintrag geschrieben. Danach wird, wie ebenfalls in Abschnitt 5.3 ausgeführt wurde, ein Rücksetzpunkteintrag geschrieben **(16)**. Den hierzu notwendigen Rücksetzpunkt-Bezeichner gibt *r_post* als Ergebnis zurück. Nun ist das Rücksetzen vollständig beendet und die normale Schrittausführung kann beginnen. Hierzu wird analog zu Algorithmus 5-1 aus der Reiseroute einer der nächsten möglichen Schritte bestimmt, dafür der Zielknoten er-

mittelt und dann das Tupel (*Agent, Log*) in die Eingangswarteschlange des Zielknotens geschrieben **(17)**. Die weitere Ausführung des Agenten geschieht dann wieder mittels Algorithmus 5-1.

Auch hier stellt die Transaktion die Atomizität der gesamten Operation (Lesen aus Eingangswarteschlange, *r_post* ausführen, Schreiben in Eingangswarteschlange des Zielknotens) sicher. Um sicherzustellen, daß die Transaktion erfolgreich ausgeführt wird, muß – analog zu den Kompensationsoperationen – durch den Entwickler des Agenten sichergestellt werden, daß die Funktion *r_post* erfolgreich ausgeführt werden kann.

Unter der Voraussetzung, daß die vom Agentenentwickler implementierten Kompensationsoperationen und die Wiederaufnahme-Funktionen *r_post* korrekt implementiert sind, d.h. die Operationen schlagen nicht fehl bzw. sind letztendlich erfolgreich, führt der vorgestellte Algorithmus das Rücksetzen der Ausführung des Agenten nach dem Modell aus Abschnitt 5.3 durch: Die schwach reversiblen Objekte und die durch den Agenten geänderten Ressourcen werden durch vom Entwickler vorgegebene Kompensationsoperationen in der korrekten Reihenfolge (umgekehrte Ausführungsreihenfolge) kompensiert und die stark reversiblen Objekte werden durch die in Rücksetzpunkten enthaltenen Informationen zurückgesetzt. Um den Zugriff des Agenten auf die zu kompensierenden Ressourcen zu gewährleisten, werden die zu einem Schritt gehörenden Kompensationsoperationen jeweils auf dem Knoten ausgeführt, auf dem der zu kompensierende Schritt ausgeführt wurde. Nach erfolgreichem Rücksetzen wird vor Wiederaufnahme der Ausführung des Agenten noch die Funktion *r_post* ausgeführt, die es dem Agenten erlaubt, den Grund für das Rücksetzen bei der weiteren Ausführung zu berücksichtigen. Die Ausführung der Kompensationsoperationen eines Schrittes innerhalb einer Transaktion in Kombination mit dem persistenten Speichern des Agenten innerhalb derselben Transaktion garantiert analog zum in Abschnitt 4.3 vorgestellten Basisprotokoll zur genau-einmal Ausführung mobiler Agenten, daß das Rücksetzen eines Agenten genau einmal geschieht.

5.4.4 Integration in den blockierungsfreien Mechanismus

Die Integration des Basismechanismus' zum Rücksetzen in das in Abschnitt 4.4 vorgestellte Protokoll zur blockierungsfreien Ausführung ist nicht sehr aufwendig. Analog zu Algorithmus 5-1 muß der Teil des Protokolles, welcher für das Lesen des Agenten aus der Eingangswarteschlange, die Ausführung des Agenten und das Schreiben des Agenten in die nächste Stufe zuständig ist (vgl. Algorithmus 4-2), um die Verwaltung des Rücksetz-Logs (Lesen von/Schreiben in Eingangswarteschlange, Schreiben der begin-of-step-, end-of-step- und savepoint-Einträge in das Log) erweitert werden. Auch beim Rücksetzalgorithmus selbst sind nur wenige Änderungen notwendig – und zwar nur beim Übergang von Ausführung auf Rücksetzen und umgekehrt.

Ruft der Agent beim Protokoll zur blockierungsfreien Ausführung *rollback*(..) auf, wird analog zu Algorithmus 5-2a die Schritt-Transaktion abgebrochen und der Agent innerhalb einer neuen

Transaktion von der lokalen Eingangswarteschlange gelesen und inklusive des Bezeichners des Ziel-Rücksetzpunktes in die Eingangswarteschlange des Knotens geschrieben, auf dem die nächste Aktion (Kompensation/*r_post*) durchgeführt werden soll. Eine möglichst einfache Integration in das blockierungsfreie Protokoll wird dadurch erreicht, indem dafür gesorgt wird, daß diese Transaktion für alle anderen Knoten der aktuellen Stufe aussieht wie eine Schritt-Transaktion. Dies erreicht man, indem man einerseits das Monitoringprotokoll nicht abbricht (d.h. es werden weiterhin *I_AM_ALIVE(..)*-Nachrichten an die anderen Knoten der Stufe verschickt) und andererseits den Koordinator des aktuellen Knotens an dieser Transaktion teilnehmen läßt. Hierdurch wird am Ende der Transaktion analog zu den Schritt-Transaktionen das Votier-Protokoll durchgeführt. Ist das Votieren erfolgreich, so wird einerseits für die Knoten der Stufe die Stufe abgeschlossen und andererseits die Transaktion erfolgreich abgeschlossen, sodaß sich der Agent nun mitsamt Log und Ziel-Rücksetzpunkt in einer Eingangswarteschlange befindet und von Algorithmus 5-2b weiterverarbeitet werden kann. Hiermit ist der Übergang von der Ausführung des Agenten zum Rücksetzalgorithmus abgeschlossen. Ist das Votieren nicht erfolgreich, so wird die Transaktion vom Koordinator abgebrochen. In diesem Falle wird ein anderer Knoten die Stufe beenden und die Entscheidung zum Rücksetzen des Agenten wird hierdurch obsolet.

Der Übergang von Rücksetzen des Agenten zur weiteren Ausführung geschieht analog der Methode *resumeExecution(..)* aus Algorithmus 5-2b. Anstatt der Auswahl eines einzelnen Knotens wird jedoch mittels dem in Abschnitt 4.5.3 vorgestellten Algorithmus zur Stufenkonstruktion die nächste Stufe zusammengestellt und der Agent in die Eingangswarteschlangen der Knoten in dieser Stufe geschrieben.

Diese geringfügigen Änderungen integrieren das Protokoll zum Rücksetzen von Agenten in das blockierungsfreie Protokoll zur Agentenausführung. Die Integration stellt auch hier sicher, daß der Agent nicht verloren geht und daß das Rücksetzen und anschließend die weitere Ausführung des Agenten sichergestellt wird. Im Gegensatz zur Ausführung des Agenten im blockierungsfreien Protokoll ist natürlich das Rücksetzen des Agenten bei dieser Art der Integration nicht blockierungsfrei: fällt ein Knoten aus, auf dem Kompensationsoperationen ausgeführt werden sollen, so ist das Rücksetzen solange blockiert, bis der Knoten wieder funktionsbereit ist.

5.5 Optimierungen

Zwei Möglichkeiten zur Optimierung des vorgestellten Rücksetzmechanismus sind die Vermeidung unnötiger Agententransporte und die Reduzierung der Größe des Rücksetz-Log. Die beiden folgenden Abschnitte präsentieren Mechanismen für diese Optimierungen.

5.5.1 Vermeidung unnötiger Agententransporte

Der in Abschnitt 5.4 vorgestellte Algorithmus transportiert (bzw. migriert) den Agenten zum Kompensieren eines Schrittes jeweils auf den Knoten, auf dem der zu kompensierende Schritt ausgeführt wurde. Häufig ist dies jedoch nicht unbedingt notwendig. Greift der Agent beispielsweise bei der Kompensation nicht auf die Ressourcen des Knoten zu oder sind zum Rücksetzen eines Schrittes gar keine Kompensationsoperationen notwendig, dann ist der Transport des Agenten auf diesen Knoten überflüssig. Ein Beispiel ist ein Agent, der auf einem Knoten während eines Schrittes nur Informationen sammelt, aber dabei den Zustand der Ressourcen (logisch) nicht ändert. Beim Rücksetzen dieses Schrittes müssen nur die auf dem Knoten gesammelten Informationen aus dem Zustand des Agenten entfernt werden – der Zugriff auf die Ressourcen des Knotens (und daher ein Transport des Agenten auf den Knoten) ist nicht notwendig. Weiterhin gibt es Fälle, bei denen zwar auf die Ressourcen zur Kompensation zugegriffen werden muß, sich der Transport des gesamten Agenten auf den entsprechenden Knoten aber nicht lohnt. Dieser Abschnitt präsentiert die für diese Optimierungen notwendigen Erweiterungen des Basisalgorithmus' zum Rücksetzen der Agentenausführung. Eine etwas einfachere, weniger mächtige Version dieser Erweiterungen wurde bereits in STRASSER UND ROTHERMEL (2000) vorgestellt.

5.5.1.1 Typen von Operationseinträgen

Damit entschieden werden kann, ob ein Agent zur Durchführung der Kompensation eines Schrittes migrieren muß oder nicht, muß das Rücksetz-Log entsprechende Informationen enthalten. Da die Notwendigkeit der Migration eng mit den durchzuführenden Kompensationsoperationen zusammenhängt, ist es naheliegend, die Operationseinträge des Rücksetz-Logs um die notwendigen Informationen und/oder Funktionalität zu erweitern. Um ein möglichst flexibles und effizientes Rücksetzen zu ermöglichen, werden vier verschiedene Typen von Operationseinträgen im Log unterschieden.

Definition 5-8: Agentenkompensationseintrag.

> Ein Agentenkompensationseintrag enthält eine Kompensationsoperation, welche nur schwach reversible Objekte des Agentenzustandes kompensiert und keinerlei Zugriff auf Ressourcen benötigt.

Die für die Ausführung dieser Kompensationsoperation notwendigen Informationen müssen im Operationseintrag und in den schwach reversiblen Objekten des Agenten enthalten sein. Wie schon in Abschnitt 5.3 erläutert, haben Kompensationsoperationen keinen Zugriff auf die stark reversiblen Objekte. Die Kompensationsoperation wird auf dem Knoten ausgeführt, auf dem sich der Agent befindet. Da kein Zugriff auf Ressourcen erlaubt ist, kann dies ein beliebiger Knoten sein. Da sich der Begriffs der schwach reversiblen Objekten direkt auf die Abhängigkeit

von der Kompensation von Ressourcen gründet (vgl. Abschnitt 5.3), wird dieser Typ von Operationseintrag eher selten benötigt.

Definition 5-9: Ressourcenkompensationseintrag.

> Ein Ressourcenkompensationseintrag enthält eine Kompensationsoperation welche nur den Zustand der durch den Agenten geänderten Ressourcen kompensiert und keinen Zugriff auf den Datenzustand des Agenten benötigt.

Die für die Ausführung dieser Kompensationsoperation notwendigen Informationen, z.B. Daten aus den stark reversiblen Objekten, müssen komplett im Operationseintrag und im Zustand der Ressourcen enthalten sein. Ein Zugriff auf den Datenzustand des Agenten ist nicht erlaubt. Hat ein Agent beispielsweise auf einem Knoten mehrere Überweisungen zwischen verschiedenen Bankkonten ausgeführt, dann muß die Kompensationsoperation die entsprechenden Rücküberweisungen durchführen. Hierzu müssen die Daten der Überweisungen wie Konten und Überweisungsbeträge im Operationseintrag enthalten sein. Ressourcenkompensationseinträge (genauer: die in ihnen enthaltenen Kompensationsoperationen) müssen auf dem Knoten der zu kompensierenden Ressourcen, d.h. auf dem Knoten auf dem der zu kompensierende Schritt ausgeführt wurde, ausgeführt werden. Da die auszuführende Kompensationsoperation keinen Zugriff auf den Agent benötigt, ist es möglich, den Ressourcenkompensationseintrag ohne Agent auf den Knoten zu schicken, auf dem die Operation ausgeführt werden muß.

Definition 5-10: Gemischter Kompensationseintrag Typ I.

> Ein gemischter Kompensationseintrag Typ I enthält eine Kompensationsoperation, welche gleichzeitig Zugriff auf die schwach reversiblen Objekte des Agenten und auf die zu kompensierenden Ressourcen benötigt.

Ein Beispielszenario für diesen Eintragstyp ist ein Schritt, in dem der Agent auf der Bank digitales Geld von US$ in Euro umtauscht. Da digitales Geld nicht in stark reversiblen Objekten gespeichert werden kann (dies folgt aus der Diskussion in Abschnitt 5.2), benötigt die zum Umtausch gehörende Kompensationsoperation Zugriff auf das schwach reversible Objekt, welches den Euro-Betrag enthält, auf das schwach reversible Objekt, in dem der US$-Betrag gespeichert werden soll, und auf die Ressource, die das Geld tauscht. Für die Ausführung eines gemischten Kompensationseintrages Typ I muß sich der Agent auf dem Knoten der zu kompensierenden Ressource(n) befinden, damit die Kompensationsoperation sowohl auf den Agent als auch auf die Ressourcen zugreifen kann. Somit entspricht dieser Eintragstyp den im Basisalgorithmus verwendeten Operationseinträgen.

Ist im eben skizzierten Währungstausch-Szenario der Rücktausch des Geldes die einzige Kompensationsoperation welche Zugriff auf die Ressourcen des Knotens benötigt, so ist der Transport des Agenten auf den Knoten der zu kompensierenden Ressource ein nicht unerheblicher

Aufwand im Verhältnis zur Aktion des Geldrücktausches. Da jedoch nach dem in Abschnitt 2.2 beschriebenen Agentenmodell der Zugriff auf Ressourcen lokal geschieht, kann nicht einfach ein entfernter Prozeduraufruf zum Rücktausch des Geldes verwendet werden. Aus diesem Grund wird noch der nachfolgende Eintragstyp eingeführt.

Definition 5-11: Gemischter Kompensationseintrag Typ II

> Ein gemischter Kompensationseintrag Typ II beinhaltet insgesamt drei (Kompensations-)Operationen: *Vor*-, *Haupt*- und *Nachoperation*. Vor- und Nachoperation haben nur Zugriff die schwach reversiblen Objekte des Agenten, die Hauptoperation hat nur Zugriff auf die zu kompensierenden Ressourcen. Die Parameter der Voroperation sind im Kompensationseintrag vollständig enthalten, die Parameter der Haupt- bzw. Nachoperation sind nur zum Teil im Kompensationseintrag enthalten. Der fehlende Teil der Parameter wird für die Hauptoperation von der Voroperation und für die Nachoperation von der Hauptoperation generiert.

Vor- und Nachoperation müssen auf dem Knoten ausgeführt werden, auf dem sich der Agent befindet, die Hauptoperation auf dem Knoten der zu kompensierenden Ressource(n). Die diesem Eintragstyp zugrundeliegende Idee ist, daß mit der Voroperation die für die Hauptoperation notwendigen Daten aus den schwach reversiblen Objekten des Agenten gelesen werden und diese der Hauptoperation als Parameter übergeben werden. Entstehen bei der Hauptoperation neue Informationen, welche wieder in die schwach reversiblen Objekte des Agenten integriert werden müssen, so bekommt die Nachoperation diese Informationen von der Hauptoperation als Parameter geliefert, welche diese Daten wieder in den Agenten integriert. Für das Szenario des Geldrücktausches würde dies bedeuten, daß die Voroperation die digitalen Euro-Münzen aus dem Agent liest und diese der Hauptoperation als Parameter übergibt. Die Hauptoperation tauscht die Euro-Münzen in US$-Münzen und übergibt die US$-Münzen an die Nachoperation, welche die US$-Münzen dann in die schwach reversiblen Objekte des Agenten schreibt. Entweder die Voroperation oder die Nachoperation kann eine leere Operation sein – wären beide Operationen leer, entspräche dies einem Ressourcenkompensationseintrag. Ist die Voroperation eine leere Operation, dann benötigt die Hauptoperation keine Daten aus den schwach reversiblen Objekten. Ist die Nachoperation eine leere Operation, müssen keine bei der Hauptoperation entstehenden Daten in den Agent integriert werden.

Wie die verschiedenen Operationseintragstypen realisiert werden, hängt von der zur Programmierung des Agenten verwendeten Programmiersprache ab. Eine Möglichkeit der Realisierung ist die Erweiterung des in Abschnitt 5.4.2 (Abbildung 5-5) vorgestellten Ansatzes. Für jeden Operationseintragstyp existiert eine Unterklasse von *CompensationObject*. Abbildung 5-7 zeigt dies anhand der Klassen für Ressourcenkompensationseinträge und gemischte Kompensationseinträge Typ II.

Für einen Ressourcenkompensationseintrag muß der Agentenentwickler eine Unterklasse von *ResourceCompensationObject* implementieren. Diese Unterklasse muß die zur Kompensation notwendigen Informationen als Attribute enthalten und die *compensate()*-Methode implementieren. Wird bei der Ausführung eines Schrittes ein Objekt dieser Unterklasse ins Rücksetzlog geschrieben, dann reicht es beim Rücksetzen dieses Schrittes aus, dieses Objekt auf den Knoten, auf dem der Schritt ausgeführt wurde, zu schicken und die *compensate()*-Methode des Objektes auszuführen. Bei Agentenkompensationseinträgen und gemischten Kompensationseinträgen Typ I ist das Vorgehen analog, die Kompensationseinträge werden jedoch direkt beim Agent ausgeführt. Bei einem gemischten Kompensationseintrag Typ II weicht das Vorgehen etwas von den anderen Eintragsarten ab. Die *compensate()*-Methode ist in diesem Falle die Hauptoperation des Eintrags, *pre()*- und *post()*-Methode stellen die Voroperation und die Nachoperation dar. Sämtliche Parameter dieser Methoden sind als Attribute im Objekt enthalten. Die von *pre()* für *compensate()* generierten Parameter werden genauso in Attributen des Objektes abgelegt wie die von *compensate()* für *post()* generierten Parameter. *hasPre()* bzw. *hasPost()* geben an, ob eine Voroperation oder eine Nachoperation existiert. Wird bei der Ausführung eines Schrittes ein solches Objekt ins Rücksetzlog geschrieben, dann wird beim Rücksetzen dieses Schrittes zuerst *pre()* beim Agenten ausgeführt (falls *hasPre()* wahr liefert). Das Objekt wird dann auf den Knoten, auf dem der Schritt ausgeführt wurde, geschickt und die *compensate()*-Methode des Objektes wird ausgeführt. Falls *hasPost()* wahr liefert, wird das Objekt dann wieder zurückgeschickt und *post()* ausgeführt. Die Klasse *CompensateMoneyExchange* zeigt die für das oben angeführte Szenario des Rücktausches von digitalem Geld notwendige Unterklasse von *MixedCompensationObjectII*. Die Methode *pre()* liest die zu tauschenden Dollar aus dem Agent und speichert sie im Attribut *forDollars*, *compensate()* tauscht diese bei der Ressource *theChangeMachine* in Euro um und speichert das Ergebnis in

```
abstract class CompensationObject{
  abstract void compensate();
}

abstract class ResourceCompensationObject
extends CompensationObject{};

abstract class MixedCompensationObjectII
extends CompensationObject{
  abstract void pre();
  abstract void post();
  abstract boolean hasPre();
  abstract boolean hasPost();
};
class CompensateMoneyExchange
extends MixedCompensationObjectII{
  Purse forDollars, for Euros;
  MoneyExchange theChangeMachine;
  void pre(){
    forEuros = getMoneyFromAgent();
  }
  void compensate(){
    forDollars = theChangeMachine.toUSD(forEuros);
  }
  void post(){
    writeMoneyToAgent(forDollars);
  }
  boolean hasPre(){return true};
  boolean hasPost(){return true};
}
```

Abbildung 5-7. Verschiedene Typen von Operationseinträgen

forEuros, *post()* schreibt das Ergebnis schließlich in den Agenten zurück.

5.5.1.2 Möglichkeiten der Optimierung

Die Vermeidung unnötiger Agententransfers als Ziel der Optimierung führt nicht unbedingt zu einer verbesserten Leistung: Aus der Definition der verschiedenen Typen von Operationseinträgen wird ersichtlich, daß die Migration des Agenten zur Kompensation eines Schrittes nur dann zwingend notwendig ist, wenn das Rücksetz-Log für diesen Schritt einen gemischten Kompensationseintrag Typ I enthält. Daraus den Schluß zu ziehen, daß bei Nicht-Existenz eines solchen Kompensationseintrages die Migration nicht notwendig ist, kann bedeuten, daß beim Vorhandensein mehrerer gemischter Kompensationseinträge Typ II mit Vor- und Nachoperationen durch die dadurch notwendige mehrfache Kommunikation ein Vielfaches an Netzwerkbandbreite und Zeit benötigt wird. Für eine Optimierung ist also ein anderes Kostenmaß notwendig, nach dem optimiert werden kann. Gängige Kostenmaße sind Netzwerklast, die für das Rücksetzen notwendigen Zeit bzw. eine Kombination von Netzwerklast und Ausführungszeit. Hierfür werden Leistungsmodelle ähnlich dem in STRASSER UND SCHWEHM (1997) vorgestellten Modell benötigt. Im folgenden wird darauf eingegangen, welche Möglichkeiten zur Optimierung sich durch die im letzten Abschnitt eingeführten Operationseinträge ergeben.

Sofern für das Rücksetzen eines Schrittes *S*, welcher auf Knoten *K* ausgeführt wurde, kein gemischter Kompensationseintrag Typ I im Rücksetz-Log enthalten ist, besteht grundsätzlich die Möglichkeit, daß der Agent zum Rücksetzen von *S* nicht nach *K* migriert wird, sondern nur die Ressourcenkompensationseinträge und die gemischten Kompensationseinträge Typ II auf den Knoten *K* verschickt werden und der Datenzustand des Agenten auf jenem Knoten rückgesetzt wird, auf dem sich der Agent momentan befindet. Eine einfache Implementierung, welche die die *rollbackOneStep*(..)-Funktion aus Algorithmus 5-2 ersetzt, zeigt Algorithmus 5-3.

Falls der Agent sich auf dem Knoten befindet, auf dem der zu kompensierende Schritt ausgeführt wurde, dann können alle Kompensationsoperationen lokal ausgeführt werden **(1)**[1]. Ist dies nicht der Fall, muß für jeden Log-Eintrag anhand des Eintragstyps entschieden werden, wie mit dem Eintrag zu verfahren ist **(8)**. Agentenkompensationseinträge werden lokal ausgeführt **(9)**.

Bei Ressourcenkompensationseinträgen wird der Eintrag mitsamt dem Bezeichner der aktuellen Rücksetztransaktion auf den Knoten geschickt, auf dem der zu kompensierende Schritt ausgeführt wurde **(10)**. Dieser Knoten, im folgenden *Ressourcenknoten* genannt, wurde dem *end-of-step*-Eintrag des zu kompensierenden Schrittes entnommen **(3)**. Es wird gewartet, bis die Vollendung der Ausführung des Ressourcenkompensationseintrages bestätigt wird. Empfängt ein Knoten einen Ressourcenkompensationseintrag, dann wird die darin enthaltene Kompensationsoperation im Kontext der Transaktion, deren Bezeichner mit übertragen wurde, ausgeführt und danach eine Bestätigung zurückgeschickt **(14)**.

1. Die Zahlen in Klammern beziehen sich auf die Fall-Numerierungen in Algorithmus 5-3

```
rollbackOneStep(Agent a, Log log, SpID spID,
                ResumeObject r_post){
    entry = log.pop() // end-of-step
    if (entry.stepExecutionNode()==nodeId){ (1)
        entry=log.pop()
        while (entry≠begin-of-step){
            entry.compensate()
            entry=log.pop()
        }
    }else{ (2)
        remoteCompensation(log,
          entry.stepExecutionNode()) (3)
    }
    if (last entry in log is savepoint){ (4)
        restore strongly reversible objects
        if (savepoint spID not reached){
            log.pop() // sp not needed anymore
        }
    }
    if (savepoint spID reached){ (5)
        target=localNodeInputQueue
    }else{
        entry = last end-of-step entry in log
        if (
          entry.stepHasMixedCompensationObjectI()){
            target = entry.stepExecutionNode() (6)
        }else{
            target = localNodeInputQueue (7)
    }   }
    Write(a,log,spID,r_post) To target
}
```

```
remoteCompensation(Log log, Node n){ (8)
    entry=log.pop()
    while (entry≠begin-of-step){
        if (entry is AgentCompensationObject){ (9)
            entry.compensate()
        }elsif(entry is
          ResourceCompensationObject){ (10)
            Send (entry, transactionId, nodeId)
                To n
            Receive(acknowledgement) From n
        }else{ // MixedCompensationObjectII (11)
            if (entry.hasPre()){entry.pre()} (12)
            Send(entry, transactionId) To n
            if (entry.hasPost()){ (13)
                Receive(entry) From n
                entry.post()
            }else{
                Receive(acknowledgement)
                    From n
            }
        }
        entry=log.pop()
    } // while
}
```

a. Ausführung von Einträgen auf dem aktuellen Aufenthaltsknoten des Agenten

```
Receive(ResourceCompensationObject r,
        TransactionId tId, fromNode n){ (14)
    register with transaction manager for tId
    r.compensate()
    Send (acknowledgement) To n
}
```

```
Receive(MixedCompensationObjectII r,
        TransactionId tId, fromNode n){
    register with transaction manager for tId
    r.compensate()
    if (r.hasPost()){
        Send(r) To n (15)
    }else{
        Send(acknowledgement) To n (16)
    }
}
```

b. Ausführung von Einträgen auf Ressourcenknoten

Algorithmus 5-3. Ausführung unterschiedlicher Operationseintragstypen

Die Ausführung gemischter Kompensationseinträge Typ II geschieht analog **(11)**, jedoch wird vor dem Versenden des Eintrages noch die *pre()*-Methode ausgeführt **(12)**. Besitzt der Kompensationseintrag eine *post()*-Methode, schickt der *Ressourcenknoten* nach der Ausführung der *compensate()*-Methode den Kompensationseintrag zurück **(15)**, damit die *post()*-Methode auf dem den Kompensationsschritt ausführenden Knoten lokal ausgeführt werden kann **(13)**. Besitzt der Kompensationseintrag keine *post()*-Methode, schickt der *ResourceNode* eine Ausführungsbestätigung **(16)**.

Nach der Ausführung der Kompensationsoperationen werden wie in Algorithmus 5-2 eventuell noch die stark reversiblen Objekte zurückgesetzt **(4)**. Danach muß entschieden werden, ob ein Transport des Agenten notwendig ist oder ob er in die lokale Eingangswarteschlange geschrieben werden kann. Ist das Ziel des Rücksetzens erreicht, so wird der Agent in die lokale Eingangswarteschlange geschrieben **(5)**. Müssen weitere Schritte zurückgesetzt werden, dann wird geprüft, ob der nächste zurückzusetzende Schritt einen gemischten Ressourcenkompensationseintrag Typ I enthält. Damit dies einfach geprüft werden kann, wird diese Information schon bei der Ausführung des Schrittes im *end-of-step*-Eintrag vermerkt. Enthält der nächste Schritt einen gemischten Kompensationseintrag Typ I, so ist ein Transport notwendig **(6)**, ansonsten nicht **(7)**.

Das Verschicken einzelner Einträge führt zusätzliche Fehlerquellen ein. Nach dem in Abschnitt 4.1.3 beschriebenen Fehlermodell kann der Knoten, zu dem die Einträge verschickt werden, ausfallen und es können Netzwerkpartitionierungen (mit unbemerktem Verlust von Nachrichten) auftreten, sodaß sich der Agent und Ressourcen in unterschiedlichen Partition befinden. Treten diese Fehler auf, erhält der Knoten, auf dem sich der Agent aufhält, keine Bestätigung über die Ausführung der verschickten Einträge (bzw. das Resultat der Ausführung von gemischten Kompensationseinträgen Typ II). Eine einfache Lösung, die sicherstellt, daß die Kompensationsoperationen genau einmal ausgeführt werden, besteht darin, daß während des Wartens auf eine Ausführungsbestätigung bzw. auf ein Resultat periodisch eine Anfrage verschickt wird, ob die Einträge gerade noch ausgeführt werden. Im Falle einer negativen Antwort auf diese Anfrage bzw. des Ausbleibens einer Antwort wird einfach die Transaktion, innerhalb der die Kompensationsoperationen ausgeführt werden, abgebrochen und das Rücksetzen des gerade bearbeiteten Schrittes erneut gestartet.

Die in Algorithmus 5-3 vorgestellte Art der Ausführung nutzt nur einen kleinen Teil des durch die verschiedenen Eintragstypen vorhandenen Potentials – nämlich die örtliche Trennung der Kompensation des Agenten und der Ressourcen – und ist deshalb nur in Ausnahmefällen effizient. Nicht ausgenutzt wird die Tatsache, daß die in den verschiedenen Eintragstypen enthaltenen Operationen teilweise auf disjunkten Datenräumen arbeiten und daher unabhängig voneinander ausgeführt werden können. Deshalb ist beispielsweise die Ausführungsreihenfolge von einem Agentenkompensationseintrag und einem Ressourcenkompensationseintrag unabhängig von der Reihenfolge, in der diese beiden Einträge im Log erscheinen. Für den Fall, daß das Rücksetz-Log für einen Schritt nur Agenten- und Ressourcenkompensationseinträge enthält, bedeutet dies, daß beim Rücksetzen des Schrittes nur darauf geachtet werden muß, daß die

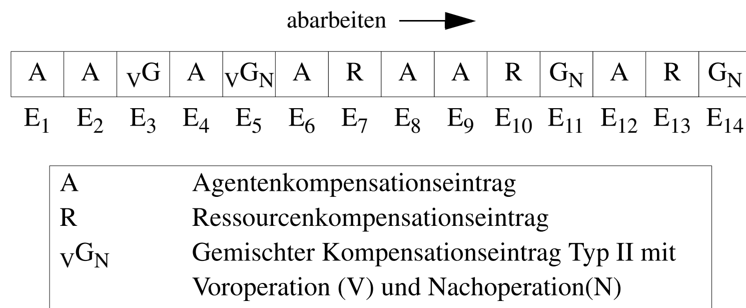


Abbildung 5-8. Beispiel-Log mit verschiedenen Operationseintragstypen

Agentenkompensationseinträge in der durch das Rücksetz-Log definierten Reihenfolge ausgeführt werden und ebenso die Ressourcenkompensationseinträge. Dadurch wird es möglich, alle Ressourcenkompensationseinträge gesammelt auf den Knoten zu schicken, auf dem der zu kompensierende Schritt ausgeführt wurde. Aus Sicht der Kompensationsoperationen ist es sogar möglich, zuerst alle Ressourcenkompensationseinträge zu verschicken und dann die Agentenkompensationseinträge parallel zu den Ressourcenkompensationseinträgen abzuarbeiten.

Etwas komplexer stellt sich die Situation dar, wenn das Rücksetz-Log für einen Schritt zusätzlich gemischte Kompensationseinträge Typ II enthält. Abbildung 5-8 zeigt ein Beispiel für ein solches Rücksetz-Log. Die Abbildung zeigt die zu einem Schritt gehörenden Operationseinträge des Rücksetz-Logs in der Reihenfolge, in der die Einträge beim Rücksetzen abzuarbeiten sind, d.h. E_1 muß zuerst abgearbeitet werden. Durch die enthaltenen Kompensationseinträge Typ II werden zwischen der Kompensation des Agentenzustandes und der Kompensation der Ressourcen Abhängigkeiten eingeführt. Durch diese Abhängigkeiten ergeben sich für die Abarbeitung der Rücksetz-Log-Einträge eines Schrittes die folgenden Randbedingungen:

- Ein Agentenkompensationseintrag kann dann abgearbeitet werden, wenn alle Operationen, die den Zustand des Agenten ändern und die laut Rücksetz-Log vor diesem Kompensationseintrag ausgeführt werden müssen, bereits ausgeführt sind. Damit ein Agentenkompensationseintrag ausgeführt werden kann, müssen daher die folgenden drei Bedingungen erfüllt sein:
    - Die laut Rücksetz-Log vorher auszuführenden Agentenkompensationseinträge müssen bereits abgearbeitet worden sein. In Abbildung 5-8 muß beispielsweise der Eintrag E_1 abgearbeitet worden sein bevor E_2 abgearbeitet wird.
    - Die laut Rücksetz-Log vorher auszuführenden gemischten Kompensationseinträge Typ II, welche eine Nachoperation besitzen müssen bereits abgearbeitet worden sein. Um den Agentenkompensationseintrag E_6 aus Abbildung 5-8 ausführen zu können muß also zuvor die Nachoperation von Eintrag E_5 und damit der gesamte Eintrag E_5 ausgeführt worden sein.

  - Von den laut Rücksetz-Log vorher auszuführenden gemischten Kompensationseinträgen Typ II, welche eine Voroperation besitzen, muß zumindest die Voroperation bereits ausgeführt worden sein. Für das Beispiel in Abbildung 5-8 bedeutet dies, daß der Agentenkompensationseintrag E_4 erst ausgeführt werden kann, wenn die Voroperation von E_3 bereits ausgeführt wurde. Die Hauptoperation von E_3 hingegen muß zur Ausführungszeit von E_4 noch nicht ausgeführt worden sein, da diese per Definition den Datenzustand des Agenten nicht ändert.

- Ein Ressourcenkompensationseintrag kann erst dann auf den Knoten der Ressourcen geschickt werden, wenn die laut Rücksetz-Log vor ihm abzuarbeitenden Ressourcenkompensationseinträge bzw. gemischten Kompensationseinträge Typ II schon verschickt wurden oder mit ihm zusammen verschickt werden, da auf dem Knoten der Ressourcen diese beiden Eintragstypen in der durch das Rücksetz-Log definierten Reihenfolge abgearbeitet werden müssen. Um beispielsweise den Ressourcenkompensationseintrag E_{10} aus Abbildung 5-8 verschicken zu können, müssen die Einträge E_3, E_5 und E_7 schon verschickt worden sein bzw. mit ihm verschickt werden, da diese Einträge schon abgearbeitet sein müssen, bevor E_{10} abgearbeitet werden kann.
- Ein gemischter Kompensationseintrag Typ II kann erst dann auf den Knoten der Ressourcen geschickt werden, wenn die laut Rücksetz-Log vor ihm abzuarbeitenden Ressourcenkompensationseinträge bzw. gemischten Kompensationseinträge Typ II schon verschickt wurden oder mit ihm zusammen verschickt werden. Für den Eintrag E_{11} aus Abbildung 5-8 bedeutet dies, daß E_3, E_5, E_7 und E_{10} vor oder mit ihm zusammen verschickt werden müssen. Hat ein gemischter Kompensationseintrag Typ II eine Voroperation, so müssen vor der Ausführung der Voroperation (und daher vor dem Verschicken des Eintrages) alle laut Rücksetz-Log vor ihm abzuarbeitenden Agentenkompensationseinträge (z.B. E_1 und E_2 vor E_3) und gemischten Kompensationseinträge Typ II, welche eine Nachoperation besitzen, abgearbeitet sein. Außerdem müssen die Voroperationen aller vor ihm abzuarbeitenden gemischten Kompensationseinträge Typ II bereits abgearbeitet sein (Voroperation von E_3 vor Voroperation von E_5). Dies stellt sicher, daß die Voroperation bei der Ausführung den korrekten Zustand des Agenten vorfindet.
- Auf dem Knoten der Ressourcen müssen Ressourcenkompensationseinträge und gemischte Kompensationseinträge Typ II in der durch das Rücksetz-Log definierten Reihenfolge abgearbeitet werden.

Diese Randbedingungen spannen den Raum der potentiellen Ausführungsmöglichkeiten auf, innerhalb dem optimiert werden kann. Beim Einsatz dieser Optimierung ändert sich die Seman-

A	A	$_VG$	A	$_VG_N$	A	R	A	A	R	G_N	A	R	G_N
E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}

Abbildung 5-8. (Wdh.) Beispiel-Log mit verschiedenen Operationseintragstypen

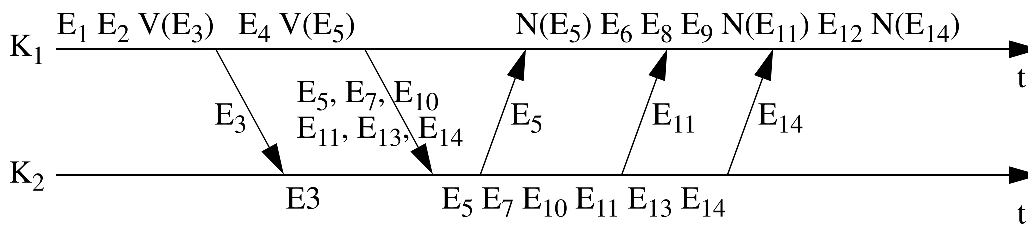
tik des Rücksetzens geringfügig. Während beim ursprünglichen Mechanismus die Funktion r_{post} immer auf dem Knoten ausgeführt wird, auf dem der zuletzt kompensierte Schritt ursprünglich ausgeführt wurde, wird sie nun auf dem Knoten ausgeführt, auf dem die letzten Kompensationsoperationen ausgeführt wurden. Dies entspricht jedoch immer noch dem Modell, welches aussagt, daß r_{post} auf einem beliebigen Knoten ausgeführt werden kann.

Unter Berücksichtigung dieser Randbedingungen gibt es viele Möglichkeiten, die Operationseinträge des Rücksetz-Logs aus Abbildung 5-8 abzuarbeiten. Eine Möglichkeit der Abarbeitung besteht darin, zuerst die Agentenkompensationseinträge E_1 und E_2, die Voroperation von E_3, den Agentenkompensationseintrag E_4 und die Voroperation von E_5 auszuführen. Danach können alle Ressourcenkompensationseinträge und gemischten Kompensationseinträge Typ II auf einmal zum Knoten mit den Ressourcen verschickt und dort in der Reihenfolge E_3, E_5, E_7, E_{10}, E_{11}, E_{13}, E_{14} ausgeführt werden. Hierbei werden von den gemischten Kompensationseinträgen nur die Hauptoperationen ausgeführt. Als Ergebnis werden die gemischten Kompensationseinträge E_5, E_{11} und E_{14} zurückgeschickt. Nachdem das Ergebnis angekommen ist, werden die Nachoperation von E_5, die Agentenkompensationseinträge E_6, E_8 und E_9, die Nachoperation von E_{11}, der Eintrag E_{12} und die Nachoperation von E_{14} ausgeführt. In diesem Fall werden also nur einmal Operationseinträge zum Knoten mit den Ressourcen verschickt und auch nur eine Antwort bestehend aus mehreren Operationseinträgen zurückgeschickt. Im Vergleich zur Ausführung mit dem einfachen Algorithmus 5-3, bei der jeder Ressourcenkompensationseintrag und jeder gemischte Kompensationseintrag einzeln verschickt worden wäre, wird hier also mehrfach die Kommunikationsverzögerung (engl.: *delay*) zwischen den Knoten eingespart.

Weitere Ausführungsmöglichkeiten ergeben sich, wenn man die Möglichkeit ausnutzt, Kompensationsoperationen auf Ressourcen und Agent parallel auszuführen. Dies ist vor allem dann sinnvoll, wenn die Agentenkompensationseinträge sehr rechen- und daher zeitintensiv sind. Abbildung 5-9 zeigt eines der möglichen Szenarien der parallelen Abarbeitung des Rücksetz-Logs aus Abbildung 5-8. Nach der Ausführung der Einträge E_1 und E_2 und der Ausführung der Voroperation von Eintrag E_3 wird der Eintrag E_3 bereits zum Knoten mit den Ressourcen geschickt. Dort kann E_3 dann schon ausgeführt werden, während lokal der Eintrag E_4 und die Voroperation von E_5 ausgeführt werden. Danach können die Einträge E_5, E_7, E_{10}, E_{11}, E_{13}, E_{14} zum Knoten mit den Ressourcen verschickt werden. Diese werden dort abgearbeitet während lokal die restlichen Agentenkompensationseinträge bzw. die Nachoperationen der gemischten Kompensationseinträge ausgeführt werden. Da die Nachoperationen immer erst dann lokal abgearbeitet werden können, sobald das Ergebnis der jeweiligen Hauptoperation vorliegt, werden die Ergebnisse immer sofort nach Abschluß der Hauptoperation verschickt. Für den Eintrag E_5 heißt dies

A	A	$_VG$	A	$_VG_N$	A	R	A	A	R	G_N	A	R	G_N
E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}

Abbildung 5-8. (Wdh.) Beispiel-Log mit verschiedenen Operationseintragstypen



K_1: Knoten, der Kompensationsschritt ausführt
K_2: Knoten mit den zu kompensierenden Ressourcen
E_x: Ausführung der (Haupt-)Operation von Eintrag E_x bzw. Transport von E_x auf anderen Knoten
V(x): Ausführung der Voroperation von Eintrag x
N(x): Ausführung der Nachoperation von Eintrag x

Abbildung 5-9. Parallele Ausführung von Agenten- und Ressourcenkompensation

beispielsweise, das nach der Ausführung der Hauptoperation auf dem Knoten mit den Ressourcen der Operationseintrag sofort wieder zum den Kompensationsschritt ausführenden Knoten verschickt wird, damit dort die lokale Verarbeitung mit der Ausführung der Nachoperation von E_5 fortgesetzt werden kann.

Um die optimale Ausführungsstrategie zu ermitteln, müssen anhand der Randbedingungen prinzipiell sämtliche möglichen Ausführungsmöglichkeiten bestimmt werden, mittels des Kostenmaßes die Kosten für die verschiedenen Möglichkeiten berechnet werden, die gemäß Kostenmaß optimale Möglichkeit ausgewählt und mit der Lösung “Migration zum Knoten, auf dem die Ressourcen kompensiert werden müssen” verglichen werden. Als Ergebnis erhält man dann die Entscheidung, ob der Agent migriert werden soll und, wenn diese mit “nein” ausfällt, auch die Strategie, wie die Operationseinträge auszuführen sind. Diese Optimierung kann separat für jeden zu kompensierenden Schritt geschehen, d.h. für jeden Schritt wird separat entschieden, ob der Agent für die Kompensation dieses Schrittes migrieren muß. Hierdurch wird ein lokales Optimum erreicht wird. Alternativ kann über alle rückzusetzenden Schritte gemeinsam optimiert werden, wodurch ein globales Optimum erreicht werden kann.

5.5.1.3 Algorithmus

Die Ermittlung der optimalen Ausführungsstrategie für den allgemeinen Fall ist sehr komplex. Daher wird in diesem Abschnitt ein optimierter Algorithmus entwickelt dem die Annahme zugrunde liegt, daß der Aufwand für die Durchführung der Kompensationsoperationen auf den Agentenzustand im Vergleich zum Aufwand für globale Kommunikation im allgemeinen vernachlässigt werden kann. Diese Annahme wird durch die sich aus der Definition der schwach reversiblen Objekte ergebende Vermutung gestützt, daß ein großer Teil der möglichen Anwendungen keine oder nur wenige schwach reversiblen Objekte verwendet und daher wenig Rechenaufwand für die lokal beim Agenten durchzuführenden Kompensationsoperationen entsteht.

Algorithmus 5-4 zeigt eine auf dieser Annahme basierende optimierte Implementierung der *remoteCompensation(..)*-Methode aus Algorithmus 5-3. Diese Methode ist für die Ausführung der Operationseinträge zuständig, wenn sich Agent und zu kompensierende Ressourcen nicht auf dem selben Knoten befinden. Ziel der Optimierung ist, möglichst selten mit dem Ressourcenknoten zu kommunizieren. Dies wird erreicht, indem immer möglichst viele Ressourceneinträge und gemischte Kompensationseinträge Typ II gemeinsam zum Ressourcenknoten verschickt werden.

Die Funktionsweise wird anhand des Logs aus Abbildung 5-10 erläutert. Der dort abgebildete Ausschnitt des Logs stellt die Operationseinträge eines einzelnen Schrittes in der Reihenfolge dar, in der sie zur Kompensation des Schrittes ausgeführt werden müssen.

Der Algorithmus besteht aus drei Phasen, die zyklisch durchlaufen werden. In einer *ersten Phase* des Algorithmus wird versucht, möglichst viel Einträge lokal abzuarbeiten. Dazu wird für jeden einzelnen Eintrag entschieden, wie mit diesem zu Verfahren ist. Agentenkompensationseinträge können sofort lokal abgearbeitet werden **(1)**[1]. Eintrag E_1 aus Abbildung 5-10 wird also sofort lokal abgearbeitet. Eintrag E_2, der ein Ressourcenkompensationseintrag ist, wird zum Verschicken auf den Ressourcenknoten vorgemerkt **(2)** – unter der Voraussetzung, daß er dort nicht schon ausgeführt worden ist (siehe weiter unten).

Die Behandlung von gemischten Kompensationseinträgen Typ II hängt davon ab, ob deren Hauptoperation schon auf dem Ressourcenknoten ausgeführt wurde. Wurde die Hauptoperation schon ausgeführt **(7)**, dann wird, falls vorhanden, noch die Nachoperation des Eintrages ausgeführt **(8)**.

Wurde die Hauptoperation noch nicht ausgeführt **(3)**, dann wird der Eintrag zum Verschicken auf den Ressourcenknoten vorgemerkt **(5)**. Zuvor wird jedoch noch, falls vorhanden, die Voroperation des Eintrages ausgeführt. Da dies auf Eintrag E_3 zutrifft, wird also dessen Voroperation ausgeführt und der Eintrag zum Verschicken vorgemerkt. Besitzt ein solcher Eintrag noch eine Nachoperation, dann beendet dies die Phase der lokalen Ausführung **(6)**, da zur Ausführung der

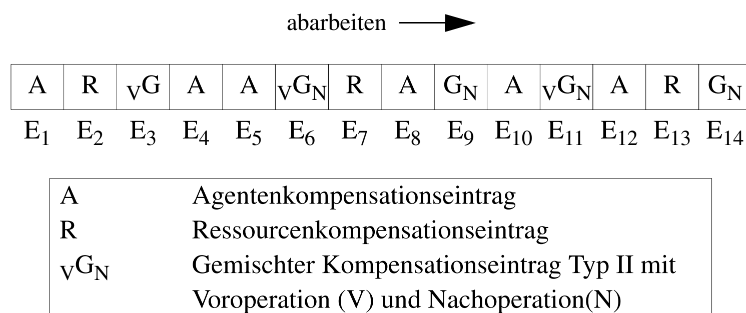


Abbildung 5-10. Beispiel-Log mit verschiedenen Operationseintragstypen

1. Die Zahlen in Klammern beziehen sich auf die Fall-Numerierungen in Algorithmus 5-4

```
remoteCompensation(Log log, Node n){
    entries = array of operation entries for this step in
              execution order
    nrEntries = number of elements in entries
    i = 0
    do{
        toSend = ∅
        stop = false
        do{
            if (entries[i] is
              AgentCompensationObject){ (1)
                entries[i].compensate()
                i++
            }else if (entries[i] is
              ResourceCompensationObject){
                if (entries[i] not already executed){
                    toSend = toSend + entries[i] (2)
                }
                i++
            }else{ // MixedCompensationObjectII
                if (main operation of entries[i] not yet
                  executed){ (3)
                    if (entries[i].hasPre()){ (4)
                      entries[i].pre()
                    }
                    toSend = toSend + entries[i] (5)
                    if (not entries[i].hasPost()){
                        i++
                    }else{
                        stop=true (6)
                    }
                }else{// main operation executed (7)
                    if (entries[i].hasPost()){ (8)
                        entries[i].post()
                    }
                    i++
            }   }
        }while( (i≠nrEntries) and not stop) (9)
        j=i+1
        stop = false
        while ((j<nrEntries) and not stop){ (10)
            if (entries[j] is
              ResourceCompensationObject){
                toSend = toSend + entries[j] (11)
            }else if (entries[j] is
              MixedCompensationObjectII){
                if (entries[i].hasPre()){
                    stop=true (12)
                }else{ (13)
                    toSend = toSend + entries[j]
            }   }
            j++
        }
        if (toSend ≠ ∅){
            Send (entries in toSend, transactionId,
              nodeId) To n (14)
            Receive(answers) From n (15)
            if (answers≠0){
                copy mixed compensation objects II
                    from answers to entries(16)
        }   }
    }while (i<nrEntries) (17)
    remove all entries of the step from log
}
```

a. Ausführung von Einträgen auf dem aktuellen Aufenthaltsknoten des Agenten

```
Receive (OperationEntries e,
         TransactionId tId, fromNode n){
    register with transaction manager for tId
    answers = ∅
    for (i=0; i<number entries in e; i++){
        e[i].compensate() (18)
        if ((e[i] is MixedCompensationObjectII)
          and e[i].hasPost()){
            answers = answers + e[i] (19)
    }   }
    Send (entries in answers) To n (20)
}
```

b. Ausführung von Einträgen auf Ressourcenknoten

Algorithmus 5-4. Optimierte Ausführung unterschiedlicher Operationseintragstypen

Nachoperation eines Eintrages zuerst dessen Hauptoperation abgeschlossen sein muß. Die erste Phase wird auch abgeschlossen, wenn alle lokal durchzuführenden Kompensationsoperationen (inklusive aller Nachoperationen) ausgeführt und die noch nicht vollständig ausgeführten Einträge zum Versenden vorgemerkt sind **(9)**.

Für das Log aus Abbildung 5-10 bedeutet dies, daß die Agentenkompensationseinträge E_4 und E_5 und dann noch die Voroperation von Eintrag E_6 ausgeführt werden und E_6 zum Versenden vorgemerkt wird.

In der *zweiten Phase* wird nun untersucht, ob zusätzlich zu den schon vorgemerkten Knoten weitere Einträge mit verschickt werden können **(10)**. Dazu wird das Log weiter linear nach Ressourcenkompensationseinträgen und gemischten Kompensationseinträgen Typ II untersucht. Ressourcenkompensationseinträge werden zum Versenden vorgemerkt **(11)**. Bei gemischten Kompensationseinträgen muß man unterscheiden. Besitzt der Kompensationseintrag eine Voroperation, so kann dieser nicht mit versendet werden, da hierzu zuerst die Voroperation ausgeführt werden muß. In diesem Fall wird das Durchsuchen des Logs abgebrochen, da die Reihenfolge der Kompensationsoperationen auf die Ressourcen gewahrt werden muß **(12)**.

Besitzt der gemischte Kompensationseintrag jedoch nur eine Nachoperation, dann kann er ebenfalls für das Verschicken vorgemerkt werden **(13)**. Die Nachoperation wird dann in der ersten Phase des nachfolgenden Zyklus ausgeführt.

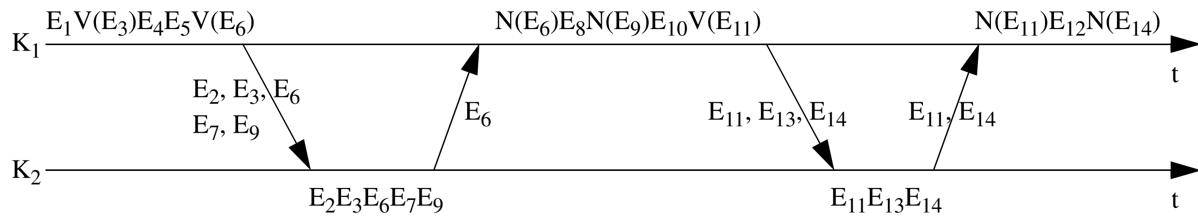
Für das Log aus Abbildung 5-10 bedeutet dies, daß der Ressourceneintrag E_7 und der gemischte Kompensationseintrag E_9 zusätzlich zum Versenden vorgemerkt werden. Der Eintrag E_{11} beendet die zweite Phase.

In der *dritten Phase* werden die in den ersten beiden Phasen zum Versenden vorgemerkten Einträge zum Ressourcenknoten verschickt **(14)**. Dort werden die Kompensationsoperationen ausgeführt **(18)**. Gemischte Kompensationseinträge Typ II mit Nachoperation werden dabei zum Rücksenden vorgemerkt **(19)**. Nachdem alle Einträge abgearbeitet sind, werden die (ausgeführten) gemischten Kompensationseinträge Typ II zu dem Knoten zurückgeschickt, der den Kompensationsschritt ausführt **(20)**. Dort angekommen, werden sie an ihre ursprüngliche Stelle im Log kopiert **(16)**.

Sind jetzt noch nicht alle Einträge des Logs abgearbeitet, wird wieder mit der ersten Phase begonnen. Für das Beispiel aus Abbildung 5-10 bedeutet dies, daß zuerst die Nachoperation des Eintrages E_6 ausgeführt wird. Eintrag E_7 wird ignoriert, da die Kompensationsoperation schon ausgeführt wurde. Eintrag E_8 wird ausgeführt, ebenso die Nachoperation des Eintrags E_9, dessen Hauptoperation schon ausgeführt wurde. Nach Ausführung von E_{10} wird noch die Vorope-

A	R	${}_VG$	A	A	${}_VG_N$	R	A	G_N	A	${}_VG_N$	A	R	G_N
E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}

Abbildung 5-10. (Wdh.) Beispiel-Log mit verschiedenen Operationseintragstypen



K_1: Knoten, der Kompensationsschritt ausführt
K_2: Knoten mit den zu kompensierenden Ressourcen
E_x: Ausführung der (Haupt-)Operation von Eintrag E_x bzw. Transport von E_x auf anderen Knoten
V(x): Ausführung der Voroperation von Eintrag x
N(x): Ausführung der Nachoperation von Eintrag x

Abbildung 5-11. Optimierte Ausführung von Agenten- und Ressourcenkompensation

ration des Eintrages E_{11} ausgeführt und dieser zum Versenden markiert. Durch die Nachoperation von E_{11} ist dann auch die erste Phase des zweiten Zyklus beendet. In der zweiten Phase werden noch E_{13} und E_{14} zum Versenden vorgemerkt und alle vorgemerkten Einträge werden dann in der dritten Phase verschickt und auf dem Ressourcenknoten ausgeführt. Da immer noch nicht alle Einträge abgearbeitet sind, wird ein dritter Zyklus gestartet. Hier werden nur noch die restlichen auszuführenden Operationen (Nachoperation von E_{11}, Agentenkompensationseintrag E_{12} und Nachoperation von E_{14}) ausgeführt. Danach sind alle Operationseinträge vollständig abgearbeitet. Abbildung 5-11 zeigt die vom optimierten Algorithmus erzeugte Ausführungssequenz im Überblick.

Die hier vorgestellte Erweiterung von Algorithmus 5-3 stellt nur eine optimierte Ausführung für den Fall der Ausführung des Kompensationsschrittes auf einem anderen Knoten als dem Ressourcenknoten bereit. Ist jedoch trotz dieser Optimierung umfangreiche Kommunikation mit dem Ressourcenknoten notwendig, wäre es möglicherweise effizienter, den Agenten zum Ressourcenknoten zu migrieren und die gesamte Kompensation dort durchzuführen. Dies kann berücksichtigt werden, indem bei der Entscheidung, ob der Agent für den nächsten Kompensationsschritt migriert werden soll, nicht nur die Existenz bzw. Nichtexistenz eines gemischten Kompensationseintrages Typ I als Kriterium dient (vgl. Algorithmus 5-3). Vielmehr muß mit einer Abwandlung des vorgestellten Mechanismus eine Ausführungsreihenfolge der Kompensationsoperationen inklusive der dabei notwendigen Kommunikation bestimmt werden und dafür mittels eines Kostenmaßes die Kosten berechnet werden. Diese können dann gegen die mit demselben Kostenmaß errechneten Kosten verglichen werden, welche bei Migration und lokaler Ausführung der Kompensation entstehen würden.

5.5.2 Reduzierung der Größe des Rücksetz-Logs

Durch das Anhängen des Rücksetz-Logs an den Agenten wird selbst während der normalen Ausführung des Agenten Mehraufwand erzeugt. Neben dem Zeitaufwand zum Schreiben der Logeinträge ergibt sich durch das Log ein erhöhter Speicherbedarf auf dem Rechner und eine erhöhte Netzwerkbelastung bei der Migration des Agenten. Abhängig von der Anwendung kann dieser Mehraufwand erheblichen Umfang annehmen. Ziel dieses Abschnitts ist es, durch Reduzierung der Größe des Rücksetz-Logs den erzeugten Mehraufwand zu vermindern.

Eine Reduzierung der Größe des Logs kann im wesentlichen nur erreicht werden, indem die Anzahl der Rücksetz-Zielpunkte eingeschränkt wird. Soll ein Agent auf den Zustand der Ausführung nach einem Schritt S zurückgesetzt werden können, so muß nach der Ausführung von Schritt S ein Rücksetzpunkteintrag ins Log geschrieben werden, welcher den möglicherweise sehr umfangreichen Zustand der stark reversiblen Objekte enthält. Wird darauf verzichtet, auf diesen Punkt zurückzusetzen, so entfällt der Rücksetzpunkteintrag. Diese Möglichkeit, den Umfang des Logs zu beschränken ist schon im in Abschnitt 5.3 vorgestellten Modell enthalten. Eine einfache Erweiterung dieser Möglichkeit ist, zu einem späteren Zeitpunkt nachträglich einen Rücksetzpunkteintrag aus dem Log zu entfernen, wenn festgestellt wird, daß dieser Rücksetzpunkt nicht mehr benötigt wird. Bei der Verwendung von Zustands-Logging ist dies kein Problem, da jeder Rücksetzpunkt die vollständigen, zum Restaurieren der stark reversiblen Objekte notwendigen Informationen besitzt. Wird hingegen Übergangs-Logging verwendet, so muß der zu entfernende Rücksetzpunkt in einen der beiden Nachbar-Rücksetzpunkte eingearbeitet werden – wodurch im ungünstigsten Fall kein oder nur sehr wenig Speicher eingespart wird.

Ein Sonderfall des nachträglichen Löschens von Rücksetzpunkteinträgen wird anhand eines Beispiellogs in Abbildung 5-12 illustriert, welches die Einträge in der Reihenfolge ihres Einfügens darstellt. Stellt die Anwendung fest, daß es auf keinen Fall mehr notwendig sein wird, auf den zum Start der Anwendung geschriebenen Rücksetzpunkt RP_1 zurückzusetzen, kann dieser gelöscht werden. Da nur auf Rücksetzpunkte zurückgesetzt werden kann, werden durch das Löschen von RP_1 alle Log-Einträge bis zum nächsten Rücksetzpunkt, im Beispiel RP_2, überflüssig und können gelöscht werden. Aus Anwendungssicht bedeutet dies, daß die Feststellung getroffen wurde, daß es bei der weiteren Ausführung des Agenten nicht mehr notwendig sein wird, auf einen Punkt vor Rücksetzpunkt RP_2 zurückzusetzen und daß die deshalb nicht mehr benötigten Teile des Rücksetz-Logs verworfen werden können. Hierdurch kann die Größe des Logs wesentlich reduziert werden.

Unglücklicherweise verkompliziert diese Optimierung die ohnehin schon komplizierte Aufgabe, die Rücksetzpunkte manuell in der Anwendung zu verwalten. Der folgende Abschnitt stellt eine komfortablere Lösung zur Verwaltung der Rücksetzpunkte vor, in die sich die vorgestellte Optimierung einfach integrieren läßt.

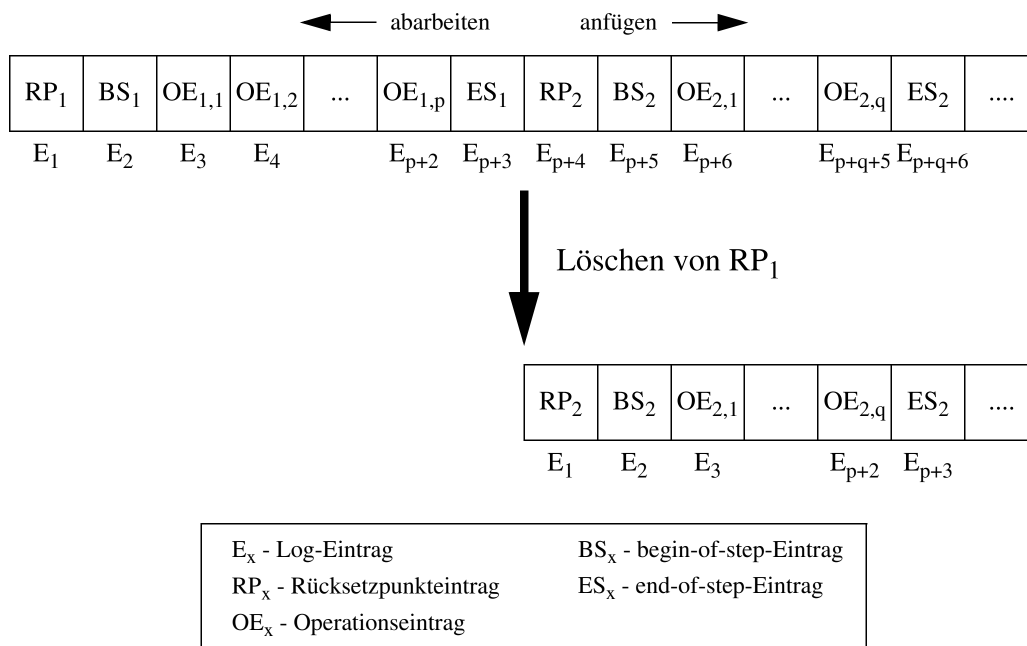


Abbildung 5-12. Löschen des ersten Rücksetzpunktes eines Logs

5.6 Verwaltung von Rücksetzpunkten

Die Verwaltung der Agenten-Rücksetzpunkte durch den Anwendungsentwickler kann bei komplexeren Anwendungen recht kompliziert werden: es muß entschieden werden, wann ein Rücksetzpunkt zu setzen ist und im Falle des Rücksetzens muß der richtige Rücksetzpunkt gefunden werden, auf den zurückgesetzt werden soll. Besonders aufwendig kann dies werden, wenn durch die Verwendung des in Abschnitt 3.2 vorgestellten Reiseroutenkonzeptes die Ausführung der Schritte des Agenten in vielen verschiedenen Reihenfolgen möglich wird.

Gerade dieses Reiseroutenkonzept bietet jedoch auch die Möglichkeit, die Verwaltung der Rücksetzpunkte zu vereinfachen. Eine der wesentlichen Eigenschaften des Reiseroutenkonzeptes ist es, durch Schachtelung mehrere Teilaufgaben des Agenten zu einer umfangreicheren Teilaufgabe in Form eines Reiserouten-Eintrages zusammenzufassen und damit eine Hierarchie von auszuführenden Teilaufgaben aufzubauen. Soll die Ausführung des Agenten partiell zurückgesetzt werden, ist es sehr wahrscheinlich, daß die gesamte gerade ausgeführte Teilaufgabe oder sogar eine der weiter oben in der Hierarchie angesiedelten Teilaufgaben zurückgesetzt werden soll. In der in Abschnitt 3.2 eingeführten Terminologie bedeutet dies entweder, daß die Ausführung des Reiserouten-Eintrags, in dem der momentan ausgeführte Basis-Eintrag direkt enthalten ist, zurückgesetzt werden soll oder daß die Ausführung eines der weiter oben in der Hierarchie angesiedelten Reiserouten-Einträge, welcher den momentan ausgeführten Basis-Eintrag indirekt beinhaltet, zurückgesetzt werden soll.

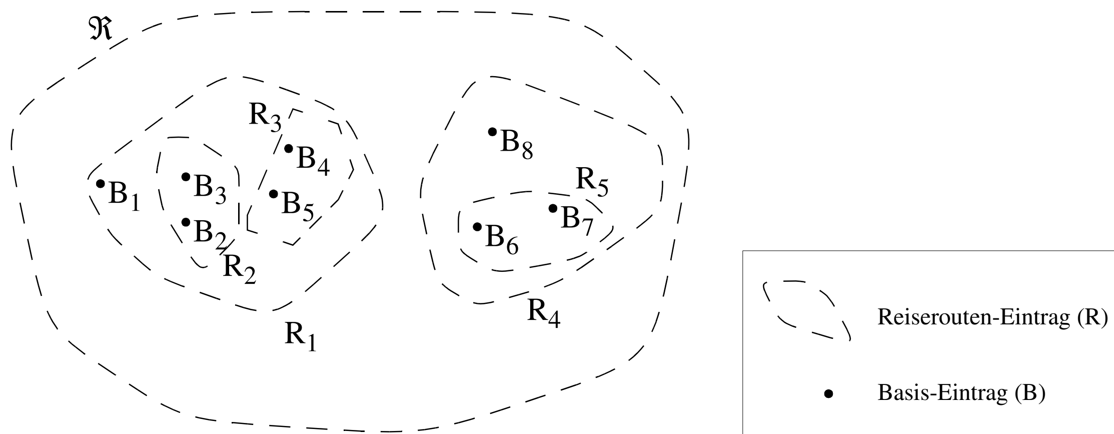


Abbildung 5-13. Teilaufgaben-Hierarchie in einer Reiseroute ℜ

Abbildung 5-13 zeigt anhand einer Beispiel-Reiseroute ℜ eine durch Reiserouten-Einträge gebildete Hierarchie von Teilaufgaben. Die Vorbedingungen und Prioritäten der Reiseroute werden in der Abbildung nicht dargestellt. Der Agent hat die Aufgabe, ein Fest zu organisieren. Hierbei stehen mehrere verschiedene Daten zur Auswahl, an denen das Fest stattfinden kann. Diese Aufgabe ist in zwei größere Teilaufgaben gegliedert. Im Reiserouten-Eintrag R_1 werden die essentiellen Aufgaben wie Festhalle buchen sowie Verpflegung und Getränke bestellen erledigt. Erst nachdem dies erledigt ist und der Termin für das Fest endgültig feststeht, wird der Rest der Organisation, z.B. das Buchen eines Alleinunterhalters bei einer Künstleragentur, im Eintrag R_4 durchgeführt. Nachdem in Basis-Eintrag B_1 die Festhalle gebucht wird, werden in R_2 die Verpflegung und in R_3 die Getränke geordert. Da schon im voraus bekannt ist, daß sowohl keiner der in Frage kommenden Party-Services die gesamte gewünschte Verpflegung als auch keine der Getränkehandlungen alle gewünschten Getränke liefern kann, werden in den Einträgen B_2 und B_3 die Verpflegung bei zwei verschiedenen Partyservices und in den Einträgen B_4 und B_5 die Getränke bei zwei verschiedenen Getränkehandlungen bestellt. Bei der Ausführung des Agenten kann durch verschiedene Umstände ein Rücksetzen der Ausführung notwendig werden. Kann beispielsweise einer der Partyservices oder eine der Getränkehandlungen an dem Datum, für das die Festhalle gebucht wurde, nicht liefern, so muß der Termin für das Fest auf eines der anderen möglichen Daten gelegt werden, da es leider keine alternativen Partyservices bzw. Getränkehandlungen gibt, die das gewünschte liefern können. In diesem Falle muß die Ausführung von R_1 zurückgesetzt werden. Stellt sich nach der Abarbeitung von B_2 während der Abarbeitung von B_3 heraus, daß der in B_3 besuchte Party-Service nicht ausreichend Verpflegung liefern kann, so reicht es aus, die Ausführung von R_2 zurückzusetzen und dann bei der erneuten Ausführung von B_2 bei diesem Party-Service entsprechend mehr zu bestellen. Entsprechendes gilt für die Getränkehandlung.

Für das gezeigte Szenario reicht es also aus, daß jeweils nur komplette Teilaufgaben-Hierarchien zurückgesetzt werden können. Dies kann vom System unterstützt werden, indem jeweils bei

Beginn der Abarbeitung einer nicht-atomaren Teilaufgabe, d.h. eines Reiserouten-Eintrages, automatisch ein Rücksetzpunkteintrag ins Log geschrieben wird. Für die Abarbeitung der Reiseroute in der Reihenfolge B_1, B_2, B_3, B_4, B_5,... heißt dies, daß vor der Abarbeitung von B_1, B_2 und B_4 jeweils ein Rücksetzpunkteintrag ins Log geschrieben wird. Soll die Ausführung des Agenten zurückgesetzt werden, muß dann als Ziel des Rücksetzens nur noch angegeben werden, wieviele Hierarchien zurückgesetzt werden sollen. Wird bei obiger Ausführungsreihenfolge während der Ausführung von B_3 entschieden, daß eine Hierarchie zurückgesetzt werden soll, so wird die Ausführung von B_3 und B_2 zurückgesetzt, für zwei Hierarchien muß zusätzlich noch B_1 zurückgesetzt werden. Reichen einer Anwendung die auf diese Weise vom System automatisch gesetzten Rücksetzpunkte nicht aus, so kann die Anwendung manuell zusätzliche Rücksetzpunkte setzen.

Das in Abschnitt 3.2 vorgestellte Reiseroutenkonzept unterscheidet bei Reiserouten-Einträgen zwischen geschlossenen und offenen Reiserouten-Einträgen. Werden die Einträge R_2 und R_3 aus Abbildung 5-13 als offene Reiserouten-Einträge spezifiziert, so wäre beispielsweise auch eine Ausführungsreihenfolge B_2, B_4, B_3, B_5 möglich. Da das Rücksetzen der Agentenausführung nur in umgekehrter Ausführungsreihenfolge möglich ist, ist es in diesem Fall nicht möglich, daß während der Ausführung von B_5 das Rücksetzen einer Hierarchie, d.h. von B_5 und B_4, beschlossen wird, da dafür auch B_3 zurückgesetzt werden müßte. Für dieses Problem gibt es zwei verschiedene Lösungen. Eine Möglichkeit ist, die offenen Reiserouten-Einträge komplett aus dem Reiseroutenkonzept zu entfernen. Dadurch entfällt ein wesentlicher Teil der Flexibilität des Reiseroutenkonzeptes. Die andere Möglichkeit ist, daß das Rücksetzen auf die geschlossenen Reiserouten-Einträge beschränkt wird, d.h. Rücksetzpunkteinträge werden nur bei Beginn der Ausführung eines geschlossenen Reiserouten-Eintrages ins Log geschrieben. Wie in Abbildung 5-14 zu sehen ist, ergibt sich hierdurch neben der Hierarchie der Teilaufgaben, welche durch geschlossene und offene Reiserouten-Einträge gebildet wird, eine weitere, nur durch geschlossene Reiserouten-Einträge gebildete Hierarchie der rücksetzbaren Teilaufgaben. Diese

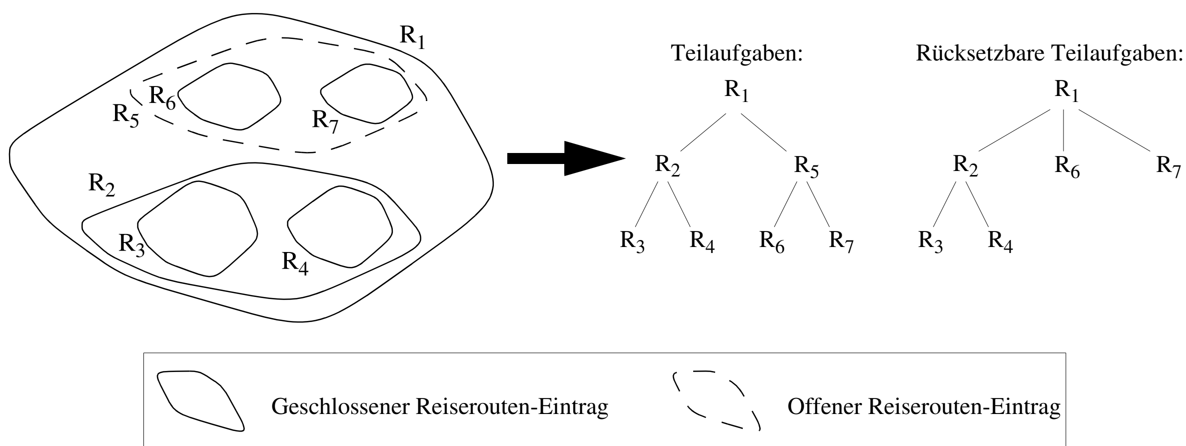


Abbildung 5-14. Teilaufgaben versus rücksetzbare Teilaufgaben

Asymmetrie zwischen Teilaufgaben und rücksetzbaren Teilaufgaben macht die grundlegende Idee, daß einfach nur die gerade durchgeführte bzw. eine weiter oben in der Hierarchie angesiedelte Teilaufgabe, zurückgesetzt wird, zunichte und vermindert dadurch den praktischen Nutzen der Integration von Rücksetzpunkten in die Reiseroute erheblich. Wird die extreme Flexibilität der Reiseroute nicht unbedingt benötigt, dann ist deshalb der Ansatz, keine offenen Reiserouteneinträge zu erlauben, auf jeden Fall die sinnvollere Lösung.

Neben der einfacheren Verwaltung von Rücksetzpunkten bringt dieses Konzept einen weiteren Vorteil mit sich. Rücksetzpunkteinträge, welche bei Beginn der Ausführung eines geschlossenen Reiserouten-Eintrages ins Log geschrieben wurden, werden nur so lange benötigt, bis dieser Reiserouten-Eintrag vollständig abgeschlossen wird. Wird beispielsweise die Reiseroute aus Abbildung 5-13 in der Reihenfolge B_1, B_2, B_3, B_4, B_5 ausgeführt, dann kann während der Ausführung von B_5 nur beschlossen werden, daß entweder R_3 oder R_1 zurückgesetzt wird. Der im Kontext der Ausführung von R_2 vor Ausführung von B_2 gesetzte Rücksetzpunkt ist daher nicht mehr notwendig. Solchermaßen überflüssig gewordene Rücksetzpunkte können vom System automatisch aus dem Rücksetz-Log entfernt werden. Dies ist jedoch nur dann möglich, wenn während der Ausführung eines Reiserouten-Eintrages von der Anwendung kein manueller Rücksetzpunkt gesetzt wird. Die Integration von Rücksetzpunkten in die Reiseroute erlaubt somit also die automatische Reduzierung der Größe des Rücksetz-Logs.

Die zweite Möglichkeit zur Reduzierung der Größe des Rücksetz-Logs, nämlich das Verwerfen nicht mehr benötigter Kompensationsoperationen, läßt sich durch eine Erweiterung des hier vorgestellten Konzepts zur Verwaltung von Rücksetzpunkten erreichen. Wie schon in Abschnitt 5.5.2 beschrieben, können Operationseinträge inklusive der dazugehörigen *begin-of-step*- und *end-of-step*-Einträge nur dann aus dem Log entfernt werden, wenn vor ihnen kein Rücksetzpunkteintrag mehr im Log steht. Damit dies geschehen kann, muß die Anwendung den ersten Rücksetzpunkteintrag im Log, der bei der Erzeugung des Agenten ins Log geschrieben wird, löschen. Die Anwendung entscheidet in diesem Falle, daß sie garantiert nie wieder soweit rücksetzen möchte. Diese Entscheidung wird mit hoher Sicherheit nur dann getroffen, wenn eine der

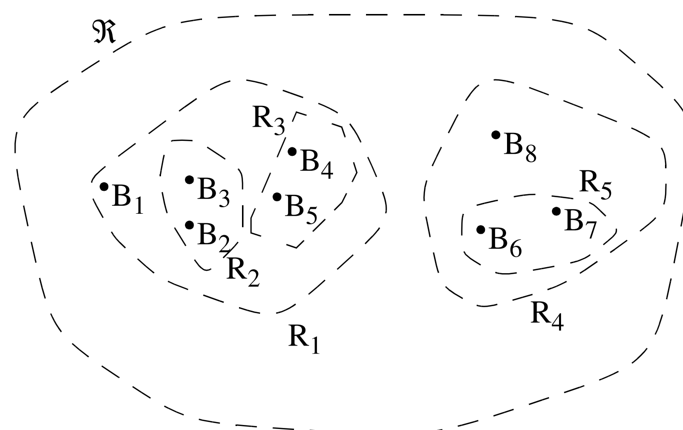


Abbildung 5-13. (Wdh.) Teilaufgaben-Hierarchie in einer Reiseroute ℜ

Haupt-Teilaufgaben des Agenten erledigt ist. Dies kann im vorgestellten Konzept dadurch unterstützt werden, indem die Hauptreiseroute nur geschlossene Reiserouten-Einträge enthalten darf. Diese werden dann als abgeschlossene Teilaufgaben interpretiert, die nach Abschluß ihrer Durchführung garantiert nicht mehr zurückgesetzt werden müssen.

Die technische Realisierung ist dann einfach: Bei der Erzeugung des Agenten wird kein Rücksetzpunkteintrag ins Log geschrieben. Solange die Anwendung keine zusätzlichen manuellen Rücksetzpunkteinträge schreibt, ist der erste Eintrag des Rücksetz-Logs immer der Rücksetzpunkteintrag jenes Reiserouten-Eintrages aus der Reiseroute, der gerade ausgeführt wird. Sobald die Ausführung eines solchen Reiserouten-Eintrages beendet ist, kann dieser Rücksetzpunkteintrag gelöscht werden. Wurde kein zusätzlicher Rücksetzpunkteintrag manuell geschrieben, bedeutet dies, daß alle Log-Einträge gelöscht werden können, da die während der Ausführung des Reiserouten-Eintrages automatisch erzeugten Rücksetzpunkteinträge auch alle automatisch wieder gelöscht wurden. Dies läßt sich auch an der in Abbildung 5-13 abgebildeten Reiseroute demonstrieren, welche in der Reihenfolge B_1, B_2, B_3, B_4, B_5, ... ausgeführt wird. Die Entwicklung des Logs während der Ausführung zeigt Abbildung 5-15. Sobald die Ausführung des Reiserouten-Eintrages R_1 mit der Ausführung von B_1 begonnen wird, wird der erste Rücksetzpunkteintrag RP_1 ins Log geschrieben. Bei Beginn der Ausführung von R_2 wird ein weiterer Rücksetzpunkteintrag RP_2 geschrieben, der nach Ausführung von B_3 wieder entfernt werden kann, da R_2 vollständig ausgeführt ist. Dasselbe gilt für die Ausführung von R_3. Nach Ausführung von R_3 ist R_1 vollständig abgearbeitet. Somit kann der zu Beginn geschriebene Rücksetzpunkteintrag RP_1 gelöscht werden. Da dieser der einzige Rücksetzpunkteintrag im Log ist, können somit alle Logeinträge von B_1, B_2, B_3, B_4 und B_5 gelöscht werden. Würde die Anwendung nach der Ausführung von B_2 einen manuellen Rücksetzpunkt RP_m setzen, dann würde sich ein komplett anderes Bild ergeben: da sowohl RP_1 als auch RP_2 nicht gelöscht werden könnten, könnten in diesem Fall gar keine Einträge aus dem Log entfernt werden.In diesem Falle müßte das Log nach manuellem Löschen von RP_m auf dann nicht mehr benötigte Rücksetzpunkte untersucht werden.

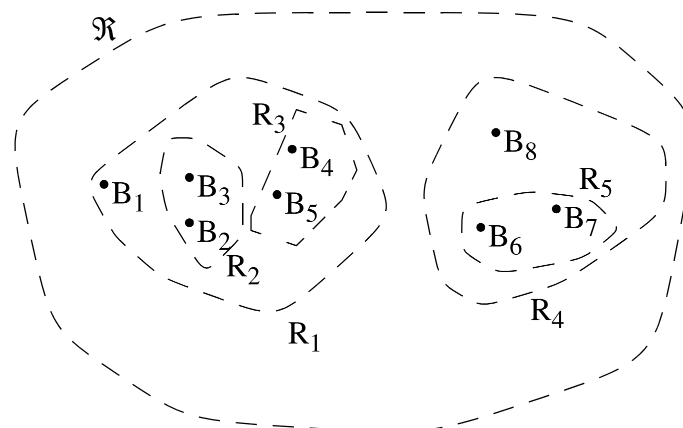


Abbildung 5-13. (Wdh.) Teilaufgaben-Hierarchie in einer Reiseroute ℜ

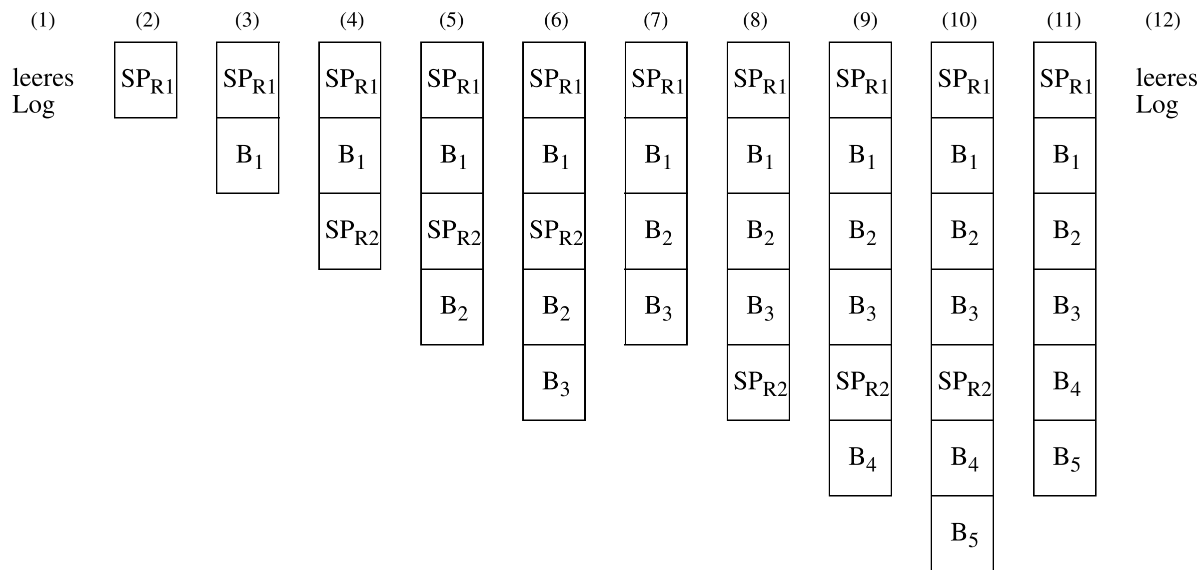


Abbildung 5-15. Automatisches Löschen von Rücksetzpunkten

5.7 Verwandte Arbeiten

Im Bereich der mobilen Agenten existiert bisher nur in dem von ASSIS SILVA UND KRAUSE (1997) vorgestellten, auf mobilen Agenten basierenden Modell für verteilte Transaktionen (vgl. auch Abschnitt 4.8.2) die nicht detailliert beschriebene Idee, der Applikation die Möglichkeit des partiellen Zurücksetzens mittels Kompensationsoperationen zu bieten. Im Bereich der Transaktionsverarbeitung hingegen existieren mehrere verwandte Arbeiten, von denen im folgenden nur die wichtigsten kurz vorgestellt werden.

Wie schon in Abschnitt 4.8.1 beschrieben, schlagen GARCIA-MOLINA UND SALEM (1987) die Verwendung von Kompensationsschritten zur Kompensation schon erfolgreich abgeschlossener Schritte bei Abbruch einer Saga und zur Backward-Recovery bei Fehlern vor. Bei der Backward-Recovery werden hierbei nur die Kompensationsschritte der seit dem letzten geschriebenen Rücksetzpunkt ausgeführten Schritte durchgeführt. Die Kompensationsschritte kompensieren nur den Zustand der an der Ausführung der Saga beteiligten Ressourcen (z.B. der Datenbank). Der Zustand der Anwendung selbst, d.h. Programmzähler, Stapel und lokale Variablen, werden mittels der beim Rücksetzpunkt gespeicherten Daten restauriert. Wie dieses Kapitel gezeigt hat, ist dies nur für sehr wenige Anwendungsfälle ausreichend. Weiterhin sieht das Saga-Konzept ein durch die Anwendung initiiertes partielles Zurücksetzen der Programmausführung auf einen spezifizierten Rücksetzpunkt nicht vor.

Eine formale Betrachtung der Kompensation findet man in KORTH, LEVY UND SILBERSCHATZ (1990). Richtungsweisend ist hier vor allem die Erkenntnis, daß zum Rücksetzen einer Programmausführung das alleinige Rücksetzen der Ressourcen mittels Kompensation und die Wiederherstellung des Programmzustandes mittels bei einem Rücksetzpunkt gespei-

cherter Daten nicht ausreicht, sondern daß auch der Programmzustand mittels Kompensationsoperationen restauriert werden muß.

Sehr ausführlich beschäftigt sich das ConTract Modell von REUTER, SCHNEIDER UND SCHWENKREIS (1997) mit dem partiellen Zurücksetzen der Programmausführung mittels Kompensation. Äußerst interessant ist hier vor allem der Ansatz, die Ausführbarkeit von Kompensationsoperationen während der Ausführung eines Scripts mittels den Schritten zugeordneten Eingangs- und Ausgangs-Invarianten zu garantieren. Hierbei wird verhindert, daß andere Anwendungen Ressourcenzustände so abändern, daß die Durchführbarkeit der Kompensationsoperationen eines gerade ablaufenden Scripts nicht mehr sichergestellt ist.

5.8 Diskussion

Aufbauend auf einer Analyse der in Kapitel 4 entwickelten zuverlässigen Ausführungsmechanismen für mobile Agenten wurde in diesem Kapitel ein zuverlässiger Mechanismus zum partiellen Rücksetzen der Ausführung mobiler Agenten entwickelt. In der Basisversion dieses Mechanismus migriert der Agent hierbei in zur Ausführung des Agenten umgekehrter Reihenfolge auf die Knoten, auf denen er Schritte ausgeführt hat und setzt den dort ausgeführten Schritt mittels Kompensationsoperationen zurück. Diese Kompensationsoperationen müssen vom Entwickler des Agenten zur Verfügung gestellt werden. Um den Entwickler hierbei zu unterstützen, wurden die im Datenzustand des Agenten enthaltenen Daten in zwei verschiedene Datentypen klassifiziert: stark reversible Objekte und schwach reversible Objekte. Der Entwickler selbst ist nur für die Kompensation der schwach reversiblen Objekte zuständig, für das Rücksetzen der stark reversiblen Objekte ist keine Unterstützung durch den Entwickler notwendig. Da in vielen Anwendungen der Agent hauptsächlich stark reversible Objekte enthält, schränkt dies den Aufwand des Entwicklers wesentlich ein, da er in diesem Fall vor allem nur für die Kompensation der durch den Agenten geänderten lokalen Ressourcen der besuchten Knoten zuständig ist.

Die für das Rücksetzen eines Agenten notwendigen Informationen (inklusive Kompensationsoperationen) werden in einem Rücksetz-Log abgelegt, das mit dem Agent migriert. Dies hat den Vorteil, daß einerseits am Ende der Ausführung des Agenten das Log sehr einfach gelöscht werden kann und andererseits das Log immer dann verfügbar ist, wenn auch der Agent verfügbar ist.

Das Ziel, das Rücksetzen des Agenten zuverlässig durchzuführen, wird erreicht, indem – analog zum Ausführungsmechanismus – das Rücksetzen eines Schrittes inklusive Migration auf den nächsten Knoten in einer Transaktion durchgeführt wird. Hierdurch wird sichergestellt, daß der Agent nicht verloren geht und jeder Schritt genau einmal zurückgesetzt wird. Unter der Voraussetzung, daß Systemfehler nur von kurzer Dauer sind und daß die vom Agentenentwickler zur Verfügung gestellten Kompensationsoperationen letztendlich immer erfolgreich ausgeführt werden, ist das Rücksetzen des Agenten erfolgreich.

Im Anschluß wurden zwei Möglichkeiten zur Optimierung des Rücksetzmechanismus diskutiert: die Vermeidung unnötiger Agententransporte und die Reduzierung der Größe des Rücksetz-Logs. Die Möglichkeit zur Vermeidung unnötiger Agententransporte ergibt sich, indem die Kompensationsoperationen danach klassifiziert werden, inwiefern sie auf den Agentenzustand bzw. den Ressourcenzustand zugreifen müssen. Kann die Kompensation von Agentenzustand und Ressourcen in getrennte Operationen gelegt werden, so ist es eventuell ausreichend, nur die entsprechenden Kompensationsoperationen der Ressourcen auf die Knoten zu verschicken, auf denen die Ressourcen kompensiert werden müssen. Wurden bei der Ausführung eines Schrittes keine Ressourcen geändert, kann die Kompensation des Agentenzustandes prinzipiell auf jedem Knoten ausgeführt werden. Die Reduzierung der Größe des Logs kann nur erreicht werden, wenn der Agent bewußt (d.h. aktiv) darauf verzichtet, auf bestimmte Zustände in der Vergangenheit zurücksetzen zu können. Die Integration der Verwaltung von Rücksetzpunkten in das Reiseroutenkonzept erlaubt dem Agentenentwickler, dies auf elegante Weise zu spezifizieren.

Kapitel 6

Resümee

6.1 Zusammenfassung

Mobile Agenten sind eine neue, vielversprechende Technologie zur Entwicklung verteilter Anwendungen in weitverteilten, heterogenen und offenen Netzwerken. Für den sinnvollen Einsatz dieser Technologie ist neben der Lösung von Sicherheitsfragen und der Problematik der Agentenkontrolle vor allem die zuverlässige Ausführung der mobilen Agenten von äußerster Wichtigkeit. Abschnitt 4.8 zeigt, daß bisher verfügbare Lösungen in diesem Bereich jeweils nur Teile der Gesamtproblematik angehen.

In der vorliegenden Arbeit wurden Mechanismen erarbeitet, welche die zuverlässige Ausführung von Agenten für das in Abschnitt 2.2 vorgestellte Agentenmodell lösen. Schwerpunkte waren die Entwicklung von Algorithmen zur genau-einmal Ausführung mobiler Agenten in Kapitel 4 und die Entwicklungen von Algorithmen zum partiellen Rücksetzen der Ausführung eines Agenten in Kapitel 5. Die Entwicklung eines Reiseroutenkonzeptes für mobile Agenten in Kapitel 3 erlaubt die Definition flexibler Reisepläne, welche von den Algorithmen zur genau-einmal Ausführung dazu benutzt werden können, die Auswahl der als nächstes zu besuchenden Knoten zu optimieren.

Ziel der genau-einmal Ausführung eines Agenten ist es, daß alle von einem Agent auszuführenden Aktionen exakt einmal ausgeführt werden, auch wenn während der Ausführung Systemfehler auftreten. Das in Abschnitt 4.3 entwickelte Basisprotokoll löst das Problem dadurch, daß der Agent zwischen den auszuführenden Teilaufgaben auf stabilem Speicher zwischengespeichert wird und die Ausführung einer Teilaufgabe im Kontext einer ACID-Transaktion erfolgt, die im Fehlerfalle einfach wiederholt werden kann.

Um zu verhindern, daß ein Agent durch einen lange andauernden Knotenausfall in seiner Ausführung blockiert wird, wurde das Basisprotokoll in Abschnitt 4.4 um weitere fehlertolerante Elemente erweitert. In diesem erweiterten Protokoll wird die Ausführung auf einem “Arbeiter”-Knoten durch “Beobachter”-Knoten überwacht. Im Falle eines Systemfehlers übernimmt einer

der Beobachter die Ausführung des mobilen Agenten. Da im allgemeinen nicht zwischen Knotenausfall und Netzwerkfehler unterschieden werden kann, wurde zur Sicherstellung der genau-einmal Eigenschaft ein Votier-Protokoll integriert, welches sicherstellt, daß die eine Teilaufgabe ausführende Transaktion nur auf einem Knoten erfolgreich beendet werden kann.

Das Basisprotokoll und seine Erweiterungen wurden analytisch hinsichtlich gewonnener Fehlertoleranz und zu erwartender Leistung untersucht. Es zeigte sich, daß sich die Wahrscheinlichkeit der Blockierung eines Agenten durch einen Systemfehler bei Verwendung der blockierungsfreien Variante zwar drastisch verringert, die durchschnittliche Gesamtausführungszeit für einen Agenten jedoch zunimmt. Messungen an einer prototypischen Implementierung der Protokolle zeigten, daß sich bei der Verwendung der blockierungsfreien Version des Protokolles die Ausführungszeit eines Agenten mit zunehmender Anzahl an Beobachtern linear erhöht.Der Einsatz dieser Variante lohnt sich also nur dann, wenn für einen mobilen Agent eine längerfristige Blockierung auf jeden Fall vermieden werden muß.

Um die Möglichkeit eines partiellen Rücksetzens für mobile Agenten zur Verfügung stellen zu können, wurden in Kapitel 5 Mechanismen entwickelt, die – analog zum Basisprotokoll – sicherstellen, daß auch für das Rücksetzen die genau-einmal Eigenschaft gewahrt bleibt. Da für das Rücksetzen im allgemeinen schon abgeschlossene Transaktionen zurückgesetzt werden müssen, mußte hierfür ein Ansatz verwendet werden, in dem schon abgeschlossene Operationen durch Kompensationsoperationen zurückgesetzt werden. Um den Anwendungsentwickler zu entlasten wurde vorgeschlagen, die im Agent rückzusetzenden Daten in zwei Typen zu untergliedern: Stark reversible Objekte sind Daten, die von der Ausführungsumgebung der Agenten ohne Unterstützung durch den Anwendungsentwickler zurückgesetzt werden können. Für das Rücksetzen der schwach reversiblen Objekte muß der Entwickler Kompensationsoperationen zur Verfügung stellen.

Der Rücksetzmechanismus ähnelt in der Ausführung dem Basisprotokoll, nur daß hier anstatt von Teilaufgaben die Kompensationsoperationen ausgeführt werden und daß sich der Agent dazu “rückwärts” bewegt, d.h. sich für die Kompensation einer Teilaufgabe auf den Knoten begibt, auf dem diese Teilaufgabe ausgeführt wurde. Eine Unterteilung der Kompensationsoperationen in verschiedene Typen erlaubt es, hierbei nicht notwendige Migrationen zu eliminieren. Ein Vorschlag zur Integration von Reiseroute und Verwaltung von Rücksetzpunkten erlaubt eine substantielle Reduktion des durch die Verwaltung des Rücksetz-Log eingeführten Mehraufwandes.

Insgesamt bieten die in dieser Arbeit vorgestellten Algorithmen zur genau-einmal Ausführung eine im Bereich mobile Agenten bisher so nicht verfügbare Zuverlässigkeit bei der Ausführung von Agenten, die nicht nur den Verlust eines Agenten durch Fehler ausschließt, sondern auch die längerfristige Blockierung eines Agenten durch Systemfehler (Knotenausfälle, Netzwerkpartitionierungen) verhindert. Die vorgestellten Rücksetzmechanismen schlagen für Agenten erstmalig konkrete Mechanismen zum partiellen Rücksetzen der Agenten-Ausführung vor.

6.2 Allgemeinheit der Ergebnisse

Die Übertragbarkeit der Ergebnisse auf andere Gebiete ist nicht ohne weiteres gegeben, da die entwickelten Mechanismen stark vom gewählten Ausführungsmodell für mobilen Agenten abhängen, welches schwache Migration verwendet.

Die Verwendung der Protokolle zur genau-einmal Ausführung ist ohne größere Probleme auch bei Agentenplattformen möglich, welche starke Migration (d.h. Migration inklusive Ausführungszustand, sodaß auf dem Zielsystem direkt nach dem Migrationsbefehl weitergearbeitet wird) verwenden. Es muß sich hier lediglich der Programmierer der Anwendung darüber klar sein, daß der Migrationsbefehl die aktuelle Transaktion beendet und eine neue Transaktion beginnt. Sollen die Protokolle außerhalb der Welt der mobilen Agenten Anwendung finden, so ist dies sicher nicht ohne Einschränkung und entsprechende Abänderungen möglich. Recht gut integrieren lassen sich die Ideen der Protokolle in alle Arten von Anwendungen, in denen einzelne, abgeschlossene Teilaufgaben nach einem irgendwie gearteten Ablaufplan abgearbeitet werden. Ein Beispiel für solche Anwendungen sind Workflow-Management-Systeme, für die LEYMANN UND ROLLER (1998) ein dem hier vorgestellten Basisprotokoll sehr ähnliches Protokoll vorschlagen. Die Anwendung der Protokolle für allgemeine Anwendungen ist eher schwierig, da die Protokolle voraussetzen, daß "zwischen" den Transaktionen ein Rücksetzpunkt auf stabilen Speicher geschrieben wird, mit dem die Verarbeitung wieder aufgesetzt werden kann. Existiert ein solcher Mechanismus, ist der Einsatz des Basisprotokolles relativ einfach möglich. Für den Einsatz der blockierungsfreien Erweiterung muß sichergestellt sein, daß ein auf stabilen Speicher geschriebener Rücksetzpunkt auf einem beliebigen Rechner (genauer: auf jedem der gewählten Beobachter) zum Wiederaufsetzen der Verarbeitung verwendet werden kann.

Wesentlich schwieriger gestaltet sich die allgemeine Verwendung des entwickelten Rücksetzmechanismus. Während in Anwendungen, in denen das Ausführungsmodell dem in dieser Arbeit zugrunde liegenden Agentenausführungsmodell ähnelt, z.B. Workflow-Management-Systeme, die Adaption des Rücksetzprotokolles keine größeren Schwierigkeiten bereiten sollte ist eine direkte Adaption auf Systeme, deren Rücksetzpunkte den Ausführungszustand, d.h. den Stack der Anwendung enthalten, beinahe nicht möglich. Der Grund hierfür ist die Trennung des Anwendungszustandes in stark und schwach reversible Objekte. Liegt ein schwach reversibles Objekt auf dem Stack, so müßten die Kompensationsoperationen auf dem Stack arbeiten, was im allgemeinen nicht möglich ist.

6.3 Ausblick

Die in dieser Arbeit vorgestellten Mechanismen haben die Technologie der mobilen Agenten ein ganzes Stück näher an die Anwendbarkeit dieser Technologie bei der Realisierung verteilter Anwendungen herangebracht. Das den Mechanismen unterliegende Agenten-Ausführungsmodell unterstützt hierbei die wichtigsten Anwendungsklassen. Um die fehlertolerante Agenten-

ausführung für beliebige Anwendungen zu ermöglichen kann das Ausführungsmodell erweitert und die Mechanismen daran adaptiert werden.

Eine für die fehlertolerante Ausführung relativ einfach durchzuführende Erweiterung ist die Möglichkeit, daß Agenten selbst wieder neue Agenten starten. Hier muß nur sichergestellt werden, daß die (Kind-)Agenten erst nach dem erfolgreichen Commit der Schritt-Transaktion ausgeführt werden, in der sie gestartet wurden. Kritischer ist diese Erweiterung für das partielle Rücksetzen, da beim Rücksetzen über den Startpunkt eines Kind-Agenten hinweg dieser Kind-Agent ebenfalls zurückgesetzt und dazu sein aktueller Aufenthaltsort bestimmt werden muß.

Eine weitere Erweiterungsmöglichkeit ist, den Kontext einer Transaktion bei der Ausführung des Agenten nicht nur über jeweils einen Schritt zu erstrecken, sondern über mehrere Schritte hinweg. Für das Basisprotokoll ist diese Erweiterung ziemlich einfach zu realisieren, für die blockierungsfreie Erweiterung hingegen gibt es mehrere mögliche Ansätze – unter anderem die Verwendung von geschachtelten Transaktionen – die im einzelnen untersucht werden müßten.

Die am schwierigsten umzusetzende Erweiterung ist die Unterstützung von direkter Kommunikation zwischen Agenten – entweder nur innerhalb einer geschlossenen Gruppe von Agenten oder allgemein zwischen beliebigen mobilen Agenten. Diese Erweiterung ist deshalb problematisch, weil durch die Kommunikation zwischen den Agenten Abhängigkeiten entstehen, die im Falle des Abbruches einer Schritt-Transaktion oder im Falle des partiellen Rücksetzens der Ausführung eines Agenten das (partielle) Rücksetzen aller Kommunikationspartner nach sich zieht. Effiziente Mechanismen zur Lösung dieser Problematik sind weiterhin Gegenstand der aktuellen Forschung.

Literaturverzeichnis

ABU-AMARA (1988)

Abu-Amara, H. (1988), “Fault-Tolerant Distributed Algorithm for Election in Complete Networks”, *IEEE Transactions on Computers 37*, 4, Seiten 449-453

AGUILERA, CHEN UND TOUEG (1998)

Aguilera, M. K. und Chen, W. und Toueg, S. (1998), “Failure detection and consensus in the crash-recovery model”, Technischer Bericht TR98-1676, Cornell University, Computer Science Department.

ARNOLD UND GOSLING (1997)

Arnold, K. und Gosling, J. (1997), *The Java(tm) Programming Language, Second Edition*, Addison-Wesley

ASSIS SILVA UND KRAUSE (1997)

De Assis Silva, F.M. und Krause, S. (1997), "A Distributed Transaction Model Based on Mobile Agents", in ROTHERMEL UND POPESCU-ZELETIN (1997), Seiten 198-209

ASSIS SILVA UND POPESCU-ZELETIN (1998)

De Assis Silva, F.M. und Popescu-Zeletin, R. (1998), “An Approach for Providing Mobile Agent Fault Tolerance”, in ROTHERMEL UND HOHL (1998), Seiten 14-25

BADER (1998)

Bader, M. (1998), “Konzeption und Implementation eines zuverlässigen und skalierbaren Agentenservers”, Diplomarbeit 1624, Fakultät Informatik, Universität Stuttgart

BARBARA, GARCIA-MOLINA UND SPAUSTER (1989)

Barbara, D. und Garcia-Molina, H. und Spauster, A. (1989), “Increasing Availability under Mutual Exclusion Constraints with Dynamic Vote Reassignments”, *ACM Transactions on Computer Systems 7*, 4, Seiten 394-426

BAUMANN *ET AL.* (1997)

Baumann, J. und Hohl, F. und Radouniklis, N. und Rothermel, R. und Straßer, M. (1997), “Communication Concepts for Mobile Agent Systems”, in ROTHERMEL UND POPESCU-ZELETIN (1997), Seiten 123-135

BAUMANN *ET AL.* (1998A)

Baumann, J. und Hohl, F. und Rothermel, K. und Straßer, M. (1998): “Mole - Concepts of a Mobile Agent System”, *World Wide Web Journal 1*, 3, Baltzer Science Publishers, Niederlande, Seiten 123-137

BAUMANN *ET AL.* (1998B)

Baumann, J. und Hohl, F. und Rothermel, K. und Schwehm, M. und Straßer, M. (1998), “Mole 3.0: A middleware for java-based mobile software agents”, in *Proceedings Middleware ’98*, N. Davies, K. Raymond, J. Seitz, Eds., Springer-Verlag London, Seiten 355 - 370

BAUMANN (1999)

Baumann, J. (1999), “Control Algorithms for Mobile Agents”, Dissertation, Fakultät Informatik, Universität Stuttgart,
URL: http://elib.uni-stuttgart.de/opus/volltexte/2000/616

BAUMANN (2000)

Baumann J. (2000), *Mobile Agents: Control Algorithms*, Lecture Notes in Computer Science 1658, Springer Verlag

BEEDUBAIL *ET AL.* (1995)

Beedubail, G. und Karmarkar, A. und Gurijala, A. und Marti, W. und Pooch, U. (1995), "Fault Tolerant Objects in Distributed Systems Using Hot Replication", Technischer Bericht TR_95-023, Department of Computer Science, Texas A&M University

BERNSTEIN UND NEWCOMER (1997)

Bernstein, P.A. und Newcomer, E. (1997), *Principles of Transaction Processing*, Morgan Kaufmann Publishers Inc., San Francisco, California

BRADSHAW (1997)

Bradshaw, J., Ed. (1997), *Software Agents*, MIT Press

BRESSOUD (1998)

Bressoud, T.C. (1998), “TFT: A Software System for Application-Transparent Fault Tolerance", in *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS-28)*, IEEE Computer Society, Seiten 128-137

BUSCHLE (1999)

Buschle, J. (1999), “Reiserouten-Konzepte für Mobile Agenten”, Studienarbeit Nr. 1754, Fakultät Informatik, Universität Stuttgart

CABRI, LEONARDI UND ZAMBONELLI (1998)

Cabri, G. und Leonardi, L. und Zambonelli, F. (1998), “Reactive Tuple Spaces for Mobile Agent Coordination”, in ROTHERMEL UND HOHL (1998), Seiten 237-248

CARZANIGA, PICCO UND VIGNA (1997)

Carzaniga, A. und Picco, G.P. und Vigna, G. (1997), “Designing Distributed Applications with Mobile Code Paradigms”, in *Proceedings of the 19th International Conference on Software Engineering ICSE 97*, ACM, Seiten 22-32

CHAUM (1985)

Chaum, D. (1985), “Security Witout Identification: Transaction Systems to Make Big Brother Obsolete”, *Communications of the ACM*, 28(10), October 1985, Seiten 1030-1040

CHESS *ET AL.* (1997)

Chess, D. und Harrison, C. und Kershenbaum, A. (1997), "Mobile agents: are they a good idea?", in *Mobile Object Systems, Towards the Programmable Internet, Second International Workshop, MO'96, Selected Presentations and Invited Papers*, Vitek, J. und Tschudin, C., Eds., Springer, Berlin, Germany, Seiten 25-47

CRISTIAN, AGHILI UND STRONG (1986)

Cristian, F. und Aghili, H. und Strong, R. (1986), "Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins", in Proceedings of the *16th International Symposium on Fault Tolerant Computing Systems (FTCS-16)*, Seiten 218-223

CUGOLA ET AL. (1996)

Cugola, G. und Ghezzi, C. und Picco, G.P. und Vigna, G. (1996), "A Characterization of Mobility and State Distribution in Mobile Code Languages", in *Special Issues in Object-Oriented Programming, Workshop Reader ECOOP'96*, dpunkt.verlag, Seiten 309-318

DALMEIJER *ET AL.* (1998)

Dalmeijer, M. und Rietjens, E. und Soede, M. und Hammer, D.K. und Aerts, A.T. (1998), "A Reliable Mobile Agents Architecture", in *Proceedings of the 1st International Symposium on Object-oriented Real-Time Distributed Computing*, IEEE Computer Society, Seiten 64-72

FIPA (1999)

FIPA (1999), "Foundation for Intelligent Physical Agents", Webseite,
URL: http://www.fipa.org/

FRIEDEL (1998)

Friedel, K. (1998), "Fehlertolerantes Protokoll zur Exactly-Once-Ausführung von Agenten", Diplomarbeit 1652, Fakultät Informatik, Universität Stuttgart

FÜNFROCKEN UND MATTERN (1999)

Fünfrocken, S. und Mattern, F. (1999), "Mobile agents as an architectural concept for Internet-based distributed applications", in *Proceedings KiVS '99, Kommunikation in Verteilten Systemen*, R. Steinmetz, Ed., Informatik aktuell, Springer-Verlag, Seiten 32 - 43.

GAEDE (1977)

Gaede, K.-W. (1977), *Zuverlässigkeit, mathematische Modelle*, Carl Hanser Verlag München Wien

GARCIA-MOLINA (1982)

Garcia-Molina, H. (1982), "Elections in a distributed Computing System", *IEEE Transactions on Computers 31*, 1, Seiten 48-59

GARCIA-MOLINA UND BARBARA (1985)

Garcia-Molina, H. und Barbara, D. (1985) "How to Assign Votes in a Distributed System", *Journal of the ACM 32*, 4, Seiten 841-860

GARCIA-MOLINA *ET AL.* (1991)

Garcia-Molina, H. und Gawlick, D. und Klein, J. und Kleissner, K. und Salem, K. (1991), "Modeling Long-Running Activities as Nested Sagas", *IEEE Data Engineering Bulletin 14*, 1, Seiten 14-18

GARCIA-MOLINA UND SALEM (1987)

Garcia-Molina, H. und Salem, K. (1987), "SAGAS", in *Proceedings ACM SIGMOD International Conference on Management of Data*, Seiten 249-259

GIFFORD (1979)

Gifford, D.K. (1979), "Weighted Voting for Replicated Data", in *Proceedings of the 7th Symposium on Operating System Principles 1979 (SOSP'79)*, ACM Press, New York, 1979, Seiten 150-162

GRASSHOPPER (2000)

IKV++ GmbH (2000), "GRASSHOPPER - THE AGENT PLATFORM",
Web-Seite, URL: http://www.grasshopper.de

GRAY UND REUTER (1993)

Gray, J. und Reuter, A. (1993), *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Francisco, California

GRAY *ET AL.* (1998)

Gray, R.S. und Kotz, D. und Cybenko, G. und Rus, D. (1998), "D'Agents: Security in a multiple-language, mobile-agent system", in VIGNA (1998), Seiten 154-187

HÄRDER UND RAHM (1999)

Härder, T. und Rahm, E. (1999), *Datenbanksysteme, Konzepte und Techniken der Implementierung*, Springer-Verlag Berlin

HÄRDER UND REUTER (1983)

Härder, T. und Reuter, A. (1983), "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys 15*, 4, Seiten 287-317

HÖFLE-ISPHORDING (1978)

Höfle-Isphording, U. (1978), *Zuverlässigkeitsrechnung*, Springer Verlag

HOHL (2000)

Hohl, F., Ed. (2000), "The Mobile Agent List",
Web-Seite, URL: http://mole.informatik.uni-stuttgart.de/mal/mal.html

HOHL (2001)

Hohl, F. (2001), "Sicherheit in Mobile-Agenten-Systemen", Dissertation, Fakultät Informatik, Universität Stuttgart,
URL: http://elib.uni-stuttgart.de/opus/volltexte/2001/893

HUGHES UND GRAWOIG (1971)

Hughes, A. und Grawoig, D. (1971), *Statistics: A Foundation for Analysis*, Addison-Wesley Publishing Company

HYLTON ET AL. (1996)

Hylton, J. und Manheimer, K. und Drake F.L. Jr. und Warsaw, B. und Masse, R.

und van Rossum, G. (1996), “Knowbot programming: System support for mobile agents”, in *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS ’96)*, Seiten 8-13

JALOTE (1994)

Jalote, P. (1994), *Fault Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey

JOHANSEN, VAN RENESSE UND SCHNEIDER (1995)

Johansen, D. und van Renesse, R. und Schneider, F.B. (1995), “Operating system support for mobile agents”, in *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, IEEE Computer Society, NY, Seiten 42-45

JOHANSEN *ET AL.* (1999)

Johansen, D. und Marzullo, K. und Schneider, F.B. und Jacobsen K. und Zagorodnov, D. (1999), "NAP: Practical Fault-Tolerance for Itinerant Computations", in *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS’99)*, Seiten 180-189

KORTH, LEVY UND SILBERSCHATZ (1990)

Korth, H.F. und Levy, E. und Silberschatz, A. (1990), "A Formal Approach to Recovery by Compensating Transactions", in Proceedings of the 16th Conference on Very Large Databases", Morgan Kaufman, Los Altos CA, Seiten 95-106

LAHRES (1964)

Lahres, H. (1964), *Einführung in die diskreten Markoff-Prozesse und ihre Anwendungen*, Vieweg-Verlag Braunschweig

LANGE UND OSHIMA (1998)

Lange, D. B. und Oshima, M. (1998), *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, Reading, Massachusetts

LANGE UND OSHIMA (1999)

Lange, D. B. und Oshima, M. (1999), “Seven good reasons for mobile agents”, *Communications of the ACM 42*, 3, Seiten 88 - 89

LAMPSON (1981)

Lampson, B. (1981), “Atomic transaction”, *Distributed Systems - Architecture and Implementation*, Goos, G. und Hartmanis, J., Eds., Lecture Notes in Computer Science 105, Springer-Verlag, Seiten 246-265

LEYMANN UND ROLLER (1998)

Leymann, F. und Roller, D. (1998), “Building A Robust Workflow Management System With Persistent Queues and Stored Procedures”, in *Proceedings of the 14th International Conference on Data Engineering*, IEEE Computer Society, Seiten 254-258

MAIHÖFER (1997)

Maihöfer, C. (1997), “Ein Protokoll zur Wahrung der Exactly-Once-Eigenschaft Mobiler Agenten”, Diplomarbeit 1565, Fakultät Informatik, Universität Stuttgart

MASUZAWA *ET AL.* (1989)

Masuzawa, T. und Nishikawa, N. und Hagihara, K. und Tokura, N. (1989), “Optimal Fault-Tolerant Distributed Algorithms for Election in Complete Networks with a Global Sense of Direction”, in *Proceedings of the 3rd International Workshop on Distributed Algorithms*, Seiten 171-182

MESSNER (1999)

Messner, A. (1999), “Neuimplementierung einer transaktionalen Message Queue”, Studienarbeit Nr. 1750, Fakultät Informatik, Universität Stuttgart

MILOJICIC *ET AL.* (1998)

Milojicic, D. und Breugst, M. und Busse, I. und Campbell, J. und Covaci, S. und Friedman, B. und Kosaka, K. und Lange, D. und Ono, K. und Oshima, M. und Tham, C. und Virdhagriswaran, S. und White, J. (1998), “MASIF: the OMG mobile agent system interoperability facility”, in ROTHERMEL UND HOHL (1998), Seiten 50 - 67

MINSKY *ET AL.* (1996)

Minsky, Y. und van Renesse, R. und Schneider, F.B. und Stoller, S.D. (1996), "Cryptographic Support for Fault-Tolerant Distributed Computing", in *Distributed Computing , Proceedings of the Seventh ACM SIGOPS European Workshop*, Seiten 109-114

OUSTERHOUT (1994)

Ousterhout, J.K. (1994), *Tcl and the Tk Toolkit*, Addison-Wesley Professional Computing

PAPOULIDIS (1999)

Papoulidis, K. (1999), “Fehlertoleranz in Mole”, Diplomarbeit 1770, Fakultät Informatik, Universität Stuttgart

PEINE UND STOLPMANN (1997)

Peine, H. und Stolpmann, T. (1997), “The Architecture of the Ara Platform for Mobile Agents”, in ROTHERMEL UND POPESCU-ZELETIN (1997), Seiten 50-61

PINDONIS (1996)

Pindonis, I. (1996), “Kooperative Informationsbeschaffung mittels Mobiler Agenten am Beispiel Reiseroutenplanung”, Diplomarbeit 1368, Fakultät Informatik, Universität Stuttgart

PITOURA (1998)

Pitoura, E. (1998), "Transaction-Based Coordination of Software Agents", in *Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA)*, Lecture Notes in Computer Science 1460, Springer Verlag, Seiten 460-469

PULIAFITO, RICCOBENE UND SCARPA (1999)

Puliafito, A. und Riccobene, S. und Scarpa, M. (1999), “An Analytical Comparison of the Client-Server, Remote Evaluation and Mobile Agents Paradigms”, in *Proceedings of the first International Symposium on Agent Systems and Applications and third International Symposium on Mobile Agents (ASA/MA 99)*, IEEE computer society, Seiten 278-292

REUTER, SCHNEIDER UND SCHWENKREIS (1997)

Reuter, A. und Schneider, K. und Schwenkreis, F. (1997), “ConTracts Revisited”, *Advanced Transaction Models and Architectures*, Jajodia, S. und Kerschberg, L., Eds., Kluwer Academic Publisher, Seiten 127-151

ROTHERMEL UND HOHL (1998)

Rothermel, K. und Hohl, F., Eds. (1998), *Mobile Agents, Second International Workshop, MA'98*, Lecture Notes in Computer Science 1477, Springer-Verlag, Berlin,

ROTHERMEL UND POPESCU-ZELETIN (1997)

Rothermel, K. und Popescu-Zeletin, R., Eds. (1997), *Mobile Agents, First International Workshop, MA'97*, Lecture Notes in Computer Science 1219, Springer-Verlag, Berlin

ROTHERMEL UND SCHWEHM (1998)

Rothermel, K. und Schwehm, M. (1998), “Mobile agents”, in *Encyclopedia for Computer Science and Technology*, A. Kent, J. G. Williams, Eds., M. Dekker Inc., New York

ROTHERMEL UND STRASSER (1997)

Rothermel, K. und Straßer, M. (1997), “A Protocol for Preserving the Exactly-Once Property of Mobile Agents”, Technischer Bericht 1997/18, Fakultät Informatik, Universität Stuttgart

ROTHERMEL UND STRASSER (1998)

Rothermel, K. und Straßer, M. (1998), “A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents”, in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems 1998 (SRDS'98)*, IEEE Computer Society, Seiten100-108

SANDER UND TSCHUDIN (1998)

Sander, T. und Tschudin, C.F. (1998), “Protecting Mobile Agents Against Malicious Hosts”, in VIGNA (1998), Seiten 44-60

SCHILL (1992A)

Schill, A. (1992), "Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick. Teil 1: Grundlagen", *Informatik-Spektrum 15*, 4, Seiten 79-87

SCHILL (1992B)

Schill, A. (1992), "Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick. Teil 2: Erweiterte RPC-Ansätze", *Informatik-Spektrum 15*, 6, Seiten 145-155

SCHNEEWEISS (1973)

Schneeweiss, W.G. (1973), *Zuverlässigkeitstheorie*, Springer Verlag

SCHNEEWEISS (1992)

Schneeweiss, W.G. (1992), *Zuverlässigkeitstechnik - von den Komponenten zum System*, Datakontext-Verlag

SCHNEIDER (1984)

Schneider, F. B. (1984), “Byzantine generals in action: implementing fail-stop processors”, *ACM Transactions on Computer Systems 2*, 2, Seiten 145 - 154

SCHNEIDER (1997A)

Schneider, F.B. (1997), "Towards Fault-tolerant and Secure Agentry", in *Distributed Algorithms, 11th International Workshop, WDAG ’97*, Mavronicolas, M. und Tsigas, P., Eds. , Lecture Notes in Computer Science 1320, Springer Verlag, Seiten 1-14

SCHNEIDER (1997B)

Schneider, K. (1997), “APRICOTS - A PRototype Implementation of a COnTract System - Ein transaktionales System zur zuverlässigen Ausführung von Workflows”, APRICOTS-Projektbericht im Rahmen von SunTREC,

SCHNEIDER (1998)

Schneider, K. (1998), “APRICOTS - ein verteiltes, transaktionales Workflow-System auf der Basis von CORBA”, Extended Abstract, in: *Datenbank Rundbrief der Gesellschaft für Informatik, Mai 98*

SINGH (1996)

Singh, G. (1996), “Leader Election in the Presence of Link Failures”, *IEEE Transactions on Parallel and Distributed Computing 7*, 3, Seiten 231-236

SPECTOR (1982)

Spector, A.Z. (1982), "Performing Remote Operations Efficiently on a Local Compter Network", *Communications of the ACM 25*, 4, Seiten 246-260

STRASSER, BAUMANN UND HOHL (1996)

Straßer, M. und Baumann, J. und Hohl, F. (1996), “Mole - A Java Based Mobile Agent System”, in: *Special Issues in Object-Oriented Programming, Workshop Reader ECOOP’96*, dpunkt.verlag, Seiten 327-334

STRASSER UND SCHWEHM (1997)

Straßer, M. und Schwehm, M. (1997), “A Performance Model for Mobile Agent Systems”, in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97), Vol II*, Arabnia, H., Ed.,CSREA, Seiten 1132-1140

STRASSER UND ROTHERMEL (1998)

Straßer, M. und Rothermel, K. (1998), “Reliability Concepts for Mobile Agents”, *International Journal of Cooperative Information Systems (IJCIS) 7*, 4, Seiten 355-382

STRASSER UND ROTHERMEL (2000)

Straßer, M. und Rothermel, K. (2000), “System Mechanisms for Partial Rollback of Mobile Agent Execution”, in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, IEEE Computer Society, Seiten 20-28

STRASSER, ROTHERMEL UND MAIHÖFER (1998)

Straßer, M. und Rothermel, K. und Maihöfer, C. (2000), “Providing Reliable Agents for Electronic Commerce”, in Trends in Distributed Systems for Electronic Commerce (TREC'98), Lamersdorf, W. und Merz, M., Eds., Lecture Notes in Computer Science1402, Springer-Verlag, Seiten 241-253

STRASSER, BAUMANN UND SCHWEHM (1999)

Straßer, M. und Baumann, J. und Schwehm, M. (1999), “An Agent-based Framework for the Transparent Distribution of Computations”, in *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’99), Vol I*, Arabnia, H., Ed., CSREA, Seiten 376-382

TANG (90)

Tang, J. (1990), “Voting Class - an Approach to Achieving High Availability for Replicated Data.” in *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems (DPDS’90)*, Agrawal, R. und D.A. Bell, Eds., IEEE Computer Society, Seiten 146-154

TEL (1994)

Tel G. (1994), *Introduction to Distributed Algorithms*, Cambridge University Press

THEILMANN UND ROTHERMEL (1999)

Theilmann, W. und Rothermel, K. (1999), “Disseminating Mobile Agents for Distributed Information Filtering”, in *Proceedings of the first International Symposium on Agent Systems and Applications and third International Symposium on Mobile Agents (ASA/MA 99)*, IEEE computer society, Seiten 152-161

THEILMANN (2000)

Theilmann, W. (2000), “Themenspezifische Informationssuche im Internet mit Hilfe mobiler Programme”, Dissertation, Fakultät Informatik, Universität Stuttgart, URL: http://elib.uni-stuttgart.de/opus/volltexte/2000/703

THOMAS (1979)

Thomas, R.H. (1979), “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases”, *ACM Transactions on Database Systems 4*, 2, Seiten 180-209

TSCHUDIN (1997)

Tschudin, C. (1997), "The Messenger Environment M0 - A Condensed Description", in *Mobile Object Systems*, Vitek, J. und Tschudin, C., Eds., Lecture Notes in Computer Science1222, Seiten 149-156

VIGNA (1998)

Vigna, G., Ed. (1998), *Mobile Agents and Security*, Lecture Notes in Computer Science 1419, Springer-Verlag

VOGLER, KUNKELMANN UND MOSCHGATH (1997A)

Vogler, H. und Kunkelmann, T. und Moschgath, M.-L. (1997), ”Distributed Transaction Processing as a Reliability Concept for Mobile Agents", in *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS’97)*, IEEE Computer Society, Seiten 59-64

VOGLER, KUNKELMANN UND MOSCHGATH (1997B)

Vogler, H. und Kunkelmann, T. und Moschgath, M.-L. (1997), “An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transactions”, in *Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS’97)*, IEEE Computer Society, Seiten 268-274

VOIGT (1996)

Voigt, T. (1996), “Entwicklung und Implementation eines Modells zur Quantitativen Beurteilung der Implementation und der Anwendung von Remote-Execution-Mechanismen”, Diplomarbeit 1436, Fakultät Informatik, Universität Stuttgart

WALSH, PACIOREK UND WONG (1999)

Walsh, T. und Paciorek, N. und Wong, D. (1999), “Security and reliability in Concordia”, in *Proceedings of the Thirty-First Hawaii International Conference on System Sciences Band 7*, IEEE Computer Society, Seiten 44-53

WÄCHTER UND REUTER (1992)

Wächter, H. und Reuter, A. (1992), “The ConTract Model”, *Database Transaction Models for Advanced Applications*, Elmagarmid, A.K., Ed., Morgan Kaufmann Publishers, San Mateo, California, Seiten 219-263

WHITE (1997)

White, J.E. (1997), “Telescript”, In *Mobile Agents: Explanations and Examples with CD-ROM*, W. Cockayne, M. Zyda, Ed., Manning Publishing, Greenwich, CT, USA, Seiten 37-57

WONG *ET AL.* (1997)

Wong, D. und Paciorek, N. und Walsh, T. und DiCelie, J. und Young, M. und Peet, B. (1997), “Concordia: An Infrastructure for Collaborating Mobile Agents”, in ROTHERMEL UND POPESCU-ZELETIN (1997), Seiten 86-97

WONG *ET AL.* (1999)

Wong, D. und Paciorek, N. und Moore, D. (1999), “Java-based mobile agents”, *Communications of the ACM 42*, 3, Seiten 92-102

XOPEN (1991)

X/Open DTP (1991), “X/Open Common Application Environment”, “Distributed Transaction Processing:Reference Model”, “Distributed Transaction Processing: The XA Specification”, Reading, Berkshire, X/open Ltd

YAP, JALOTE UND TRIPATHI (1988)

Yap, K.S. und Jalote,P. und Tripathi, S. (1988), "Fault Tolerant Remote Procedure Call", in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS’88)*, IEEE Computer Society, Seiten 48-54