Providing Reliable Agents for Electronic Commerce

Markus Straßer, Kurt Rothermel, Christian Maihöfer

Institute of Parallel and Distributed High-Performance Systems (IPVR),
University of Stuttgart, Germany

{strasser,maihoefer,rothermel}@informatik.uni-stuttgart.de

Abstract. It is widely agreed that mobile agents in conjunction with WWW technology will provide the technical foundation for future electronic commerce. A prerequisite for the use of mobile agents in a commercial environment is, that agents have to be executed reliable, independent of communication and node failure.

In this paper, we first present a recently proposed fault-tolerant protocol to ensure the exactly-once execution of an agent by monitoring the agents execution. With this protocol, agents are performed in so-called stages. Each stage consists of a number of nodes. One of these nodes executes the agent while the other nodes monitor the execution.

The main focus of this paper is the construction of stages. In particular, we will investigate how the number of nodes per stage influence the probability of an agent to be blocked due to failures and which nodes should be selected when forming a stage to minimize the overhead caused by the protocol. Also a flexible itinerary concept is proposed that gives agent systems the freedom to do various kinds of optimizations when determining the next node and constructing a stage.

1 Introduction

With the rapidly increasing use of on-line services by professional and private users, huge demand arises for an open and secure electronic marketplace offering services and goods to customers in the network. New *virtual shopping malls* or *electronic store fronts*, providing their offers to a huge customer base, spring up like mushrooms almost every day. The turnover of on-line business transactions increases by huge amounts every year.

It is widely agreed that agent technology in conjunction with WWW will provide the technical foundation for future electronic commerce. Mobile agents are autonomous objects that are able to migrate from node to node in a computer network using services offered on these nodes. The ability to roam the net (by moving the agent's code, data and execution state) is provided by a middleware platform, a mobile agent execution environment (e.g. Aglets[7], Ara[8], Concordia[11], Mole[1]). In electronic commerce scenarios, agents autonomously go shopping on the user's behalf, make the reservations needed for a business trip, or monitor the stock market and trigger user-defined operations when certain conditions occur. Obviously, many of these tasks require an agent to be executed "exactly once". Let us consider a scenario: A user launches a mobile agent

This research was funded by Tandem Computers Inc., Cupertino (CA) and the German Research Community (DFG, RO 1086/4-1)

to make a flight and hotel reservation for a forthcoming business trip. The agent is expected to make both reservations if possible, and in any case return a status message back to the user. The user will delegate this job to an agent only if it is guaranteed that the agent does it "exactly once". In other words, it must be ensured that the agent is never lost independent of node and communication failures and hence will get its job done eventually. Moreover, failures must not cause the agent to perform operations more than once (e.g., to reserve and pay for a seat twice instead of once).

The concept of a mobile agent allows for asynchronous operation of arbitrarily complex tasks. For example, in the scenario above the user can forget about the agent after launching it. Instead of controlling the progress of the agent, the user just synchronizes with the agent and receives the results the day before his business trip. Clearly, this type of agent autonomy is very attractive for applications in the field of electronic commerce, in particular if users are mobile. Unfortunately, exactly this property may cause problems: If an agent is "caught" on a node due to a node crash or a network partition, no instance will detect this fact. This situation is even worse if there are alternative nodes that could continue the agent processing.

Consequently, in the context of electronic commerce, agent-based systems should provide for mechanisms ensuring the "exactly-once" property as well as an increased level of fault-tolerance. As we will see later, these requirements imply the integration of agent systems with transactional technology. Unfortunately, none of the currently available agent systems provide this type of functionality yet. In [9], a protocol for preserving the "exactly-once" property of mobile agents has been proposed. With this protocol, agents are performed in so-called stages. Each stage consists of a number of nodes. Each of these nodes adopts one of two possible roles, either the role of a worker or the role of an observer. While the worker node actually performs the agent, the observers monitor the worker. When the worker becomes unavailable, one observer of this stage takes over the agent and executes it instead. To ensure the "exactly-once" property, a voting mechanism is integrated into the 2-phase commit processing.

The main focus of this paper is the construction of stages. In particular, we will investigate how the number of nodes per stage influence the probability of an agent to be blocked due to failures, and what types of stage nodes must be considered. We will propose two types of nodes, those that actually can execute the agent during normal processing, and those that just can execute exception handling routines. We will analyse which nodes should be selected when forming a stage to minimize the communication overhead caused by the protocol. Also a flexible itinerary concept is proposed that gives agent systems the freedom to choose from a set of possible optimizations to maximize efficiency when determining the next node and constructing a stage. More rigid types of itineraries are used in the Aglet system [7], in Telescript [2] and in Concordia [11].

The paper is structured as follows. In the next Section, a novel type of itinerary is introduced, which allows for the flexible specification of an agent's travel plans. Based on the notion of an itinerary we will then define the notion of "exactly once" execution in the context of mobile agents. In Section 3, an overview of the protocols for preserving the exactly-once property is given. Subsequently, in Section 4, the various aspects of constructing a stage are investigated in detail. The paper concludes with a brief summary.

2 The Exactly-Once Property of Mobile Agents

If we want to define the notion of the exactly-once semantics of mobile agents, we need a model describing the execution of a mobile agent. This model describes the execution of an agent in terms of the nodes the agent visits and the actions it takes on a node. The actions performed by the agent on a node are called a *step*. A facility that allows to specify the nodes to be visited is called *itinerary*, which is presented in this section. We will then use this concept of an itinerary to define the exactly-once property of mobile agents.

2.1 The Itinerary

While performing a job, a mobile agent often has to visit several nodes to use services offered locally. In many cases, some (or all) of these nodes are either known before agent initialization or can be determined by the agent several steps in advance. However, as in real life, no strict order exists in which the nodes have to be visited. For example, an agent having to buy a CD, one pound of beef and a theatre ticket, may perform these tasks in any sequence. On the other hand, if there are several branches of a music shop, the agent needs only to visit one of these branches. To exploit the possible benefits given by a flexible travel plan (e.g. by calculating the shortest path) and to provide a powerful facility to the agent developer, an *itinerary concept* is provided. This *itinerary concept* allows a very flexible specification of an agent's travel plan as well as the dynamic adaptation and expansion of the travel plan during the execution of the agent.

The *itinerary* is composed using different types of *itinerary entries*. The simplest form of an entry is a simple pair (*node*, *method*) specifying a node which has to be visited and the step (defined by *method*) which has to be executed on this node (see [11]). The other possible entries, called *sequence*, *set* and *alternative* contain several other entries (recursively). A *sequence* is a list $[e_1,...,e_n]$ of n entries ($n \geq 1$) defining that the nodes specified by entry e_i ($1 \leq i < n$) must have been visited before the nodes of entry e_{i+1} are visited. A *set* is a set of entries $\{e_1,...,e_n\}$ specifying that the elements $e_1,...,e_n$ can be handled in any order as long as each element is handled exactly once. An *alternative* $(e_1,...,e_n)$ allows to specify that exactly one of the entries $e_1,...,e_n$ have to be chosen.

To clarify this definition, 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 reserve 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 node of the flower service, one of the two nodes offering the ticket service (unfortunately, there is no central ticket service for both theatres), and, depending on the chosen theatre, the node of the restaurant.

The itinerary i specifying the travel plan of our concierge agent is defined using the notation introduced above by

i = { (BestFlowers, buyFlowers),
([(CentralTheatre, buyTicket), (KingsInn, reserveTable)],
[(ModernArts, buyTicket), (BeefHouse, reserveTable)]
)}.

A graphical representation of the itinerary is shown in Figure 1a. The top level entry of the itinerary is a *set* specifying that the agent has to go and buy flowers on node "BestFlowers" (using the method buyFlowers) and to follow the specification in the *alternative*. This can be performed in any order. The *alternative* specifies the two possible ways of buying a theatre ticket and making a reservation for a table. Each alternative is defined using *sequences*. The *sequences* specify that the agent first has to go to the theatre to buy a ticket using the method "buyTicket", and afterwards to go to the restaurant to make a reservation for a table (a sequence is used here instead of a set because the agent needs the information when the theatre play ends to make the reservation).

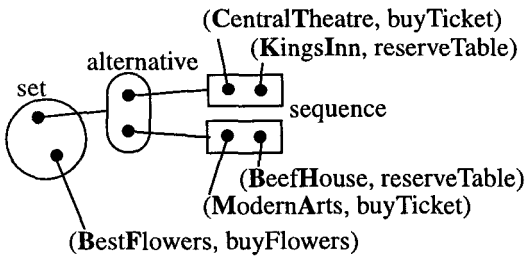

Fig. 1a. An itinerary...

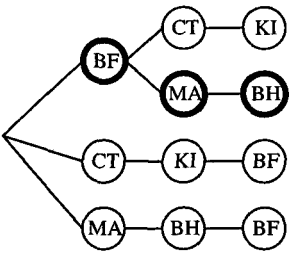

Fig. 1b. ... and the corresponding tree of possible paths

Given this itinerary, the system can decide which node has to be visited next provided there are alternatives. For the first step, it has the possibility to visit either one of the theatre nodes or the flower shop. The possibilities for the next step depend on the alternative chosen. Using the information given in the itinerary, a tree containing all possible paths of the agent can be constructed. The tree in Figure 1b shows all possible paths that can be taken by Pauls 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 with bold circles.

The itinerary provides operations to query and to change its content. The query methods allow to navigate through the entries in the itinerary, to provide information about which nodes already have been visited and which nodes, according to the itinerary, may be visited next. The change methods allow to insert new entries and to delete entries in the part of the itinerary not yet processed. This allows the agent to gain an overview over the current state of its execution and to change the itinerary dynamically during its execution.

2.2 The "Exactly-Once" Property

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 on the visited nodes.

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 nodes 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 node $N_{i,j}$ of path P_i ($1\leq i\leq n$, $1\leq j\leq L(P_i)$). Then the execution of an agent is defined to be exactly-once if

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

In the above scenario, 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 nodes are performed in one of the orders defined by the tree of possible paths in Figure 1b. Each of this steps is executed exactly once.

3 Protocols for Providing the "Exactly-Once" Property

To preserve the exactly once property of an agent, a simple protocol, based on transactional message queues, can be used. As we will see, this solution unfortunately is very prone to node and network failures possibly trapping the agent until the node or the network recovers from failure. If, for example, one of the ticket service nodes of the example in Section 2 becomes unavailable for several hours while Paul's concierge agent visits this node, it may happen that the agent will not be able to fulfil its task in time even though this would be possible by just using the other ticket service. Therefore, an extension of this protocol has been proposed in [9] increasing the degree of fault tolerance. Here, we will confine ourselves to describing the basic principles of this protocol.

3.1 The Simple Protocol

The exactly-once property of mobile agents can be achieved in a simple way by using transactional message queues (e.g., see [4]). Transactional message queues provide for persistent messages and ensure the exactly-once delivery. 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 [5].

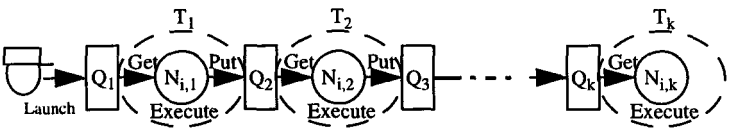

Fig. 2. Simple implementation of exactly-once agents using message queues

Figure 2 depicts how transactional message queues can be used to implement exactly once agents. At each migration and at the start of the agent, the runtime system chooses one of the possible destinations of the agents. The nodes 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*. As a result, the agent moves from node to node along one of the possible paths P_i ($N_{i,1} \rightarrow N_{i,2} \rightarrow ... \rightarrow N_{i,k-1} \rightarrow N_{i,k}$ ($k=L(P_i)$) of the agent. At the start, the agent is put in the input queue of the first node 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 node recovers eventually - will eventually be performed exactly once.

All nodes (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 node's input queue, *Execute* performs the received agent locally, and *Put* places it in the input queue of the next node. 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 node 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 undone also. After recovery is finished, $N_{i,j}$ continues normal processing and will eventually execute this agent and then hand it over to its successor. In the scenario in Section 2, the ticket node that is visited by the concierge agent gets the agent from its input queue, performs the agent, enabling the agent to use the ticket service of the node, and finally puts the agent in the input queue of the restaurant node. If this transaction is aborted (e.g. due to a node crash), all effects of this transaction, including the purchase of the ticket, are undone, resulting in the agent residing in the input queue of the (same) ticket service node (until the ticket node recovers).

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 node 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, due to the autonomy of mobile agents, no "natural" instance monitoring the progress of an agent. Therefore, a novel protocol was proposed in [9] that enables the system to monitor the agents execution and, if necessary, allows it to react on failures by executing the agent on alternative nodes. A short sketch of this protocol is given below.

3.2 The Fault-Tolerant Protocol

To allow for fault-tolerance, the execution model described above is extended by the concept of *stages* [10]: The execution of an agent proceeds in a sequence of stages. An agent enters a new stage whenever it moves to the next node. For each stage there exists a non-empty set of nodes, which alternatively can perform that stage. One node of the stage, holding the role of the *worker* node, executes the agent while the other nodes of the stage, the *observers*, monitor the availability of the stage's worker. When the worker becomes unavailable (caused by either a node failure or a network partition), this will be detected by the observers, which then will elect a new worker from the set of available stage nodes. Each stage node is associated with a *priority* that defines a total ordering between the nodes belonging to the same stage (which is required for the voting and selection process). The initial worker of a stage will become the node with the highest priority. Figure 3 shows a 4-stage execution of an agent. For example, stage S_1 has no observers, while stage S_2 is associated with one worker, and 4 observers. In S_3, the node with the highest priority (1) fails and the node with priority 2 is selected to be the new worker.

The execution of an agent on the worker node is performed inside an ACID transaction. To start the agent, the worker node begins a new transaction, gets the agent from

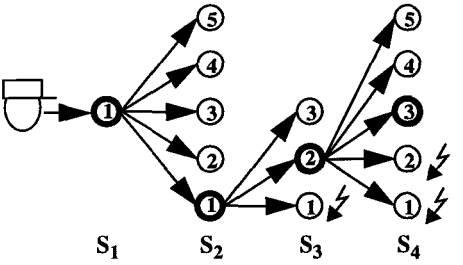

Fig. 3. Execution of an agent in 4 stages

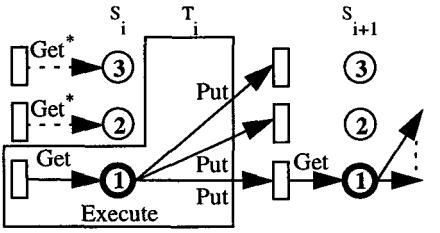

Fig. 4. Transactional processing of an agent in a stage

its input message queue[1] 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 node, the worker node puts the agent into the message queues of the nodes of the next stage and commits the transaction. Figure 4 shows the transactional execution of an agent in stage S_i. Please note that the *Get* operations of the observer nodes are excluded from the transactions, because including them would require all stage nodes to be available to commit the stage.

In our shopping scenario in Section 2, the nodes of the first stage might be the ticket service nodes and the flower shop node, with one of the ticket shops having the highest priority, the other ticket shop the medium priority and the flower shop the lowest priority. After an abort of the transaction (e.g. due to a node crash) on the ticket service node with the highest priority (which becomes the worker node first), all effects of our concierge agent on this node are undone and the other ticket service node is selected as the new worker. In this case, the concierge agent is able to start over at the beginning of the stage instead of being blocked.

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 node crash or network partitioning) and initiates the selection protocol.

The selection of a new worker node is performed by the *selection protocol*[2]. This protocol is a variant of the bully algorithm described in [3]. An observer detecting the failure of the worker node sends an *Are_You_There* message to all stage nodes with a higher priority. Available nodes (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 stage nodes about it. If the initiator receives a reply instead, it cancels its selection procedure and starts monitoring the new

1. Transactional message queues are used for the transport of agents between nodes.
2. We do not use the term election protocol here since the problem of choosing a new worker (possibly resulting in two worker nodes) differs from the well-known "election problem" (electing exactly one node). For details see [9].

worker. This protocol results in the node 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 node may commit its transaction. This voting protocol has to be integrated into the 2-phase commit protocol (2PC) of the transaction. When the transaction manager issues the prepare request, voting requests are sent to all stage nodes. A stage node receiving a voting request responds depending on the fact if it has already voted for another node in the stage or not. Only if a worker node gets a majority of votes from the other stage nodes, the transaction on this node commits and the other stage nodes are notified about the commit. Otherwise, the transaction is aborted on this node. In [9], the protocol is described in detail.

4 Stage Construction

The careful choice of the nodes of a stage is essential to the gain in reliability as well as to the performance of the protocol. The reliability of the protocol particularly depends on the number of nodes in a stage while the performance is influenced by which nodes are chosen. This section examines how many nodes should be assigned to a stage, which nodes should be used for a stage and how the priorities of the nodes should be determined.

4.1 Number of Stage Nodes

The worker node needs to collect a majority of votes during 2PC processing to be able to commit the transaction of a stage. Therefore, a transaction can only be committed if more than half of the stage nodes (including the worker) are available. This fact can be used to give a (simple) metric for the availability A_s of a stage which is the probability that a majority of stage nodes is available so that an agent can finish a step and proceed with the next step.

Let n be the number of the nodes of a stage and p be the availability of an individual node (i.e. the probability that the node is available). Then the probability that exactly m out of these n nodes are available can be calculated using the binomial probability function $f(n, m) = \binom{n}{m} p^m (1-p)^{(n-m)}$ [6]. The availability $A_s(n, p)$ of a stage can then be calculated by

$$A_s(n, p) = \sum_{i=\left\lceil \frac{n+1}{2} \right\rceil}^{n} \binom{n}{i} p^i (1-p)^{(n-i)} .$$

The blocking probability, i.e. the probability that the agent is blocked in the stage, is calculated by $B_s(n, p) = 1 - A_s(n, p)$. The relative blocking probability $B_r(n, p)$ is calculated by $B_r(n, p) = B_s(n, p)/B_s(1, p)$. A relative blocking probability of $B_r(n, p)$=0.4 means that the probability of an agent blocking in a stage with n nodes (node availability

p) is only 40% of the probability of an agent blocking on one node with availability p.

Table 1 shows the availability A_s of a stage and the relative blocking probability B_r depending on the availability p of a node and the number n of stage nodes. In the case of using only 2 stage nodes, both nodes need to be available resulting in a $B_r > 1$. Using three or more nodes generally decreases the blocking probability considerably. However, observe that the relative blocking probability of a stage using an even number of stage nodes is bigger as if using one node less. This may even result in a relative blocking probability $B_r>1$ for small n (e.g. p=0.75, n=4, B_r = 105%). Therefore, the number of nodes to be used in a stage should be an odd number bigger or equal to three.

Table 1. Availability and Relative Blocking Probability of a Stage

n	p					
	0.75		0.9		0.99	
1	0.75	100%	0.9	100%	0.99	100%
2	0.5625	175%	0.81	190%	0.9801	199%
3	0.8438	62%	0.972	28%	0.9997	3%
4	0.7383	105%	0.9477	52%	0.9994	6%
5	0.8965	41%	0.9914	9%	~1	~0%
6	0.8306	68%	0.9842	16%	~1	~0%
7	0.9294	28%	0.9973	3%	~1	~0%

4.2 Types of Stage Nodes

There are two different types of stage nodes, *regular nodes* and *exception handling nodes* (short: *exception nodes*). Let j be the node currently executing the agent, then *Nextj* defines the set of nodes that can be visited next according to the agent's itinerary. In the example depicted in Figure 1b the *Next* set associated with node *BF* includes nodes *MA* and *CT*. Let node *j* be the worker of stage *i*-1 then a node of stage *i* is called a *regular node* if it is member of *Nextj*. All other nodes of the stage are called *exception nodes*. In other words, regular nodes provide the services needed to perform the "regular" steps of an agent, while exception nodes are only expected to provide a runtime environment for agents. An agent is only moved to an exception node if in the stage no regular node is available due to failures. Each agent is supposed to provide a method *NoRegularNodeAvailable()*, which is initiated when the agent arrives on an exception node. As mentioned above, on regular nodes the method specified for the node in the itinerary is performed.

4.3 Performance Considerations

Unfortunately, the fault-tolerant protocol also introduces some overhead in the error-free case. Figure 5 shows the additional communications needed. During the execution of a stage, the worker node periodically sends *I_Am_Alive* messages to all other nodes (dashed arrows). After the agent has finished its stage-specific work, the agent, consisting of code, data state and execution state, has to be transferred to all nodes of the next stage (solid arrows). Then, the transaction has to be committed. Resource managers affected due to the protocol are the local input queue of the worker, the input queues of the nodes of the next stage (solid arrows) and the local instance performing the voting protocol. Additionally, the voting protocol (dashed arrows) is executed during the 2PC.

Finally, the observers of the "old" stage have to be informed about the termination of the stage (dashed arrows).

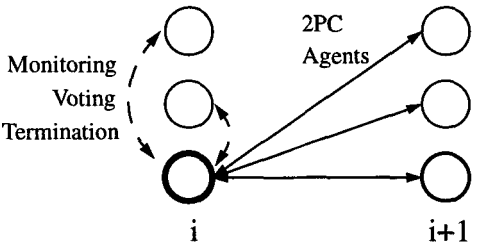

Fig. 5. Communication Patterns

For the performance considerations we use a (simplified) cost model that assumes that communication costs for a message of a fixed size are the same between all involved nodes. Therefore, only the amount of data transfers over the net is considered.

One of the possibilities to reduce the amount of data being sent over the net is the reduction of the nodes of a stage. The more nodes a stage contains, the more data has to be sent over the net during migration, for monitoring purposes and during the commit. On the one hand, this optimization contradicts the aim of the protocol to increase the availability. On the other hand, the application developer himself has the possibility to weigh up the increase of availability against communication costs by deciding how many nodes to use in a stage.

The amount of messages needed for monitoring a fixed number of stage nodes is fixed. In the two phase commit protocol, messages are sent between the current worker and the nodes of the next stage resulting in a total of $4n$ messages where n is the number of nodes in the next stage. If the current worker is also a member of the next stage (see algorithm below), the amount of messages for 2PC reduces to $4(n\text{-}1)$.

The only other possibility to reduce the amount of data being sent over the net for a fixed number of stage nodes is to reduce the amount of data being sent during the migration. This can be achieved by carefully choosing the nodes of a stage so that the intersections of successive stages are as big as possible resulting in a reduced amount of code being transferred. If, for example, some nodes of the current stage are also part of the next stage, no code has to be transferred onto these nodes when the agent migrates. If, additionally, the current worker is also part of the next stage (as exception node), no data and execution state has to be transferred to this node.

4.4 Stage Construction Algorithm

The algorithm described in this section aims at constructing a stage such that communication overhead is minimized during normal operation. The basic idea of the algorithm is to use as much as possible regular nodes to construct a stage and, if there is any freedom in the choice of nodes, to ensure that consecutive stages have as much nodes in common as possible. As an input this algorithm takes the number of stage nodes, say n, and the agent's itinerary. Assume node w is the worker of stage i. Then w performs the algorithm to determine the nodes of stage i+1, say S_{i+1}, together with the nodes' priorities.

Before describing the algorithm, we have to introduce some terminology. S_i is defined to be the nodes of stage i that are available from w's point of view. The *Next* set is used as defined above, i.e., $Next_w$ defines the set of nodes that potentially can follow w according to the agent's itinerary.

Case 1: In the simplest case, the cardinality of $Next_w$ ($|Next_w|$) equals n. In this case, S_{i+1} is equal to $Next_w$, which by definition includes regular nodes only. The way how priorities are assigned to these nodes will be described below.

Case 2: If the cardinality of $Next_w$ is bigger than n, then the resulting stage also contains only regular nodes. The choice of stage nodes is performed in two steps: In the first step, $S_{i+1}=S_i \cap Next_w$ is computed. If $|S_{i+1}|\geq n$, the priorities of the nodes in S_{i+1} are determined (see below) and the n nodes with the highest priorities remain in S_{i+1}. In this case, no second step is needed. If $|S_{i+1}|<n$, then $n-|S_{i+1}|$ nodes are selected from $Next_w \backslash S_i$ according to their priorities (see below). Subsequently, the final priorities of the nodes in S_{i+1} are calculated.

Case 3: If the cardinality of $Next_w$ is smaller than n, $m=n-|Next_w|$ exception nodes have to be chosen. Good candidates for this choice are the nodes in S_i, particularly w. By using these nodes, the number of code transfers for migrating the agent from stage i to stage $i+1$ can be reduced by $|S_i \cap S_{i+1}|$. In addition, using w as an exception node of stage $i+1$ reduces the number of data and execution state transfers and saves 4 messages during 2PC processing.

If $m\leq|S_i \backslash Next_w|$, m nodes - including w - are taken from $S_i \backslash Next_w$ as exception nodes according to their priorities. If $m>|S_i \backslash Next_w|$, then all nodes in $S_i \backslash Next_w$ are used as exception nodes. In order to select the yet missing exception nodes, future destinations specified in the itinerary can be taken into account. In sum, S_{i+1} includes $Next_w$ and a set of exception nodes selected as described above. Since $Next_w$ includes regular nodes, the exception nodes are assigned a lower priority than the ones in $Next_w$.

Priorities: To determine the priorities of the nodes, several possibilities exist. A simple approach is to randomly assign the priorities just ensuring unique priorities per stage. A more effective strategy is to exploit knowledge about node reliability. In this case, the priorities are assigned in accordance with the reliability, i.e., the higher the reliability, the higher the priority. Priorities between nodes with equal reliability can be chosen randomly. Obviously, with this heuristic more reliable nodes are preferred for agent execution.

A third strategy to determine the priorities is to take into account not only the next stage but also the ones following the next stage. Unfortunately, the nodes which can be visited in a stage depend on the worker of the previous stage, making the computations rather complex and time consuming. Therefore, this path hasn't been investigated further here.

The possible reductions of the protocol overhead gained by the presented algorithm are shown in Table 2. Assuming a number of stage nodes of n=3, the use of e.g. one node of the current stage as stage node of the next stage (regular or exception node) already reduces the number of code transports by one third. The example itinerary shown in Figure 6 represents the optimal case for the stage construction algorithm.

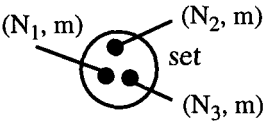

Fig. 6. Simple Itinerary

Table 2. Possible Reductions of the Overhead

	Code Transports	State Transports	Messages for 2PC
$S_i \cap S_{i+1} = \{\}$	n	n	$4n$
$S_i \cap S_{i+1} \neq \{\}$	$n - \lvert S_i \cap S_{i+1} \rvert$	n	$4n$
$S_i \cap S_{i+1} \neq \{\}$ $w_i \in S_{i+1}$	$n - \lvert S_i \cap S_{i+1} \rvert$	n-1	$4(n$-1)

The first stage (n=3) contains nodes N_1 to N_3, ordered by their priority. For this first stage, only the code has to be transported to the nodes (3 code transports) and the transaction of putting the code onto the first stage nodes has to be committed (4*3 messages). In the second stage, the worker of the first stage, (assume N_1), acts as exception node. Here, no code transports, only two state transports and 4*2 messages for the 2PC are necessary. In the third stage, N_1 and the worker of the second stage (assume N_2) act as exception nodes. The number of data transports is the same as in the last stage. Therefore, a total of 3 code transports (no overhead at all!), 4 state transports (overhead: 2 transports) and 28 messages for the 2PC are needed. Without the flexible definition of the itinerary and the optimization of the stage construction algorithm, 9 code transports, 9 state transports and 36 messages for 2PC would have been used.

In our example of Section 2, the first stage (n=3) contains the two ticket service shops and the flower shop (3 code transports + 4*3 messages). If the concierge agent is executed on the flower shop node first, the second stage consists of the two ticket service nodes (as regular nodes) and the flower shop node as exception node (two state transports and 4*2 messages). Then, the third stage consists of one of the restaurant nodes (depending on the ticket node used as worker in stage two) as regular node and the worker node of stage two and one other node of stage two as exception nodes (one code transport, two state transports and 4*2 messages). This results in a total of 4 code transports (overhead: one transport), 4 state transports (overhead: two transports) and 28 messages for 2PC. In this case, each stage contains as much as possible regular nodes. If, on the other side, the worker node of the first stage is one of the ticket service nodes, there can be an additional code transport if the flower shop is not contained in the second stage (the flower shop node is not in $Next_w$ in this case, which is rather an inadequacy of the itinerary concept than of the stage construction algorithm).

5 Conclusions and Future Work

Based on the necessity of being able to represent the travel plan of an agent for defining the exactly-once protocol, an itinerary concept has been introduced. This itinerary concept provides the electronic commerce application developer with a very flexible possibility to specify the travel plans of an agent (e.g. a shopping agent). Besides that, the flexibility of this concept in many cases allows the system to choose from a set of several nodes the agent may visit next, e.g. by enabling the system to optimize the agents way through the net or to visit nodes temporarily not available after the other nodes have

been visited. This can result in considerable reductions of the time and costs needed for electronic commerce transactions.

Then, a solution for one of the most important aspects of electronic commerce transactions, the guaranteed, fault tolerant execution of an agent was tackled: After presenting a simple protocol for the preserving of the exactly-once property of mobile agents, a fault-tolerant protocol was introduced that provides fault-tolerance by monitoring the agents execution by several nodes. One important task of the protocol, the decision on which node to move the agent next and which nodes to take for the monitoring was examined. An algorithm was presented that exploits the flexibility provided by the itinerary to reduce the overhead introduced by the fault tolerant protocol.

Despite being very flexible, the itinerary concept introduced in this paper does not cover all desirable possibilities to specify the travel plans for an agent. Therefore, more work will be invested in improving the itinerary concept to allow a finer specification of the relations of nodes an agent intends to visit. Another subject to investigate will be the possibilities of taking into account more of the future travel plans provided by the itinerary to improve the optimizations performed by the stage construction algorithm.

References

[1] Baumann, J.; Hohl, F.; Rothermel, K.; Straßer, M.: "Mole - Concepts of a Mobile Agent System." accepted for "WWW Journal, Special issue on Applications and Techniques of Web Agents", 1998.

[2] General Magic: "Agent Technology", URL: http://www.genmagic.com/agents/

[3] Garcia-Molina, H.: "Elections in a Distributed Computing System." In: IEEE Transactions on Computers, Vol. C-31, No. 1, January 1982.

[4] Gray, J.; Reuter, A.: "Transaction Processing - Concepts and Techniques." Morgan Kaufmann Publishers Inc, 1994.

[5] Haerder, T.; Reuter, A.: "Principles of Transaction-Oriented Database Recovery." ACM Computing Surveys, 15(4), 1993.

[6] Hughes, A.; Grawoig, D.: "Statistics: A Foundation for Analysis." Addison-Wesley Publishing Company, 1971.

[7] Lange, D.; Oshima, M.: "Mobile Agents with Java: The Aglet API." In "Special issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents." Baltzer Science Publishers, 1998.

[8] Peine, H.; Stolpmann, T.: "The architecture of the Ara platform for mobile agents." In: Mobile Agents, Proc. 1st Int. Workshop, MA'97. Springer, 1997.

[9] Rothermel, K.; Straßer, M.: "A Protocol for Preserving the Exactly-Once Property of Mobile Agents." Technical Report 1997/18, Faculty of Information Science, University of Stuttgart, Germany, 1997. Also submitted for publication.

[10] Schneider, F.: "Towards Fault-tolerant and Secure Agentry." In: M. Mavronicolas and P. Tsigas (eds.), "Distributed Algorithms, 11th International Workshop, WDAG '97." Lecture Notes in Computer Science, Volume 1320. Springer, 1997.

[11] Wong, D.; Paciorek, N.; Walsh, T.; DiCelie, J.; Young, M.; Peet, B.: "Concordia: An Infrastructure for Collaborating Mobile Agents." In: Rothermel, K.; Popescu-Zeletin, R. (eds.): "Mobile Agents. First International Workshop MA '97." Lecture Notes in Computer Science, Vol. 1219, Springer. 1997, pp. 86-97.