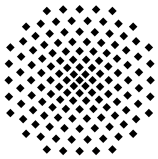

Universität Stuttgart
Fakultät Informatik

A Protocol for Preserving the Exactly-Once Property of Mobile Agents

Kurt Rothermel, Markus Straßer

Email: {kurt.rothermel, markus.strasser}@informatik.uni-stuttgart.de

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

A Protocol for Preserving the Exactly-Once Property of Mobile Agents

K. Rothermel, M. Straßer

Bericht 1997/18
Oktober 1997

A Protocol for Preserving the Exactly-Once Property of Mobile Agents[1]

Kurt Rothermel and Markus Straßer

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

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

Abstract

Mobile agents are autonomous objects that can migrate from node to node of a computer network. Mobile agent technology has been proposed for various application areas, including electronic commerce, systems management and active messaging. Many of these applications - especially those for electronic commerce - require agents to be performed “exactly once”, independent of communication and node failures. In other words, once a mobile agent has been launched, it must never be lost before its execution is finished. Moreover, each “portion” of the agent performed at the visited nodes is performed exactly once.

Due to the autonomy of mobile agents, there is no “natural” instance that monitors the progress of an agent’s execution. As a result of that agents may be blocked due to node crashes or network partitioning even if there are other nodes available that could continue processing. In this paper, we will describe a protocol that ensures the exactly once property of agents and additionally reduces the blocking probability of agents by introducing so-called observer nodes for monitoring the progress of agents. This protocol is based on conventional transactional technology, such as defined by X/Open DTP or CORBA OTS. It is implemented in the Mole, a mobile agent system developed at Stuttgart University.

1. Introduction

Over the last few years, the concept of mobile agents has drawn a lot of attention in both academia and industry. Today many prototypes of mobile agent systems exist, most of them based on the Java programming language. Moreover, various efforts to standardize mobile agent technology are already underway (e.g., OMG MAF, CSELT FIPA). However, despite of all these activities, only few “real” applications based on mobile agents exist today. One reason for that might be that current mobile agent platforms are in a rather early stage. Application critical functions, such as security mechanisms, are often incomplete or missing at all. Moreover, only little work has been done so far in studying the problem of integrating agent technology with legacy systems, such as TP-Monitors and transactional resource managers. In this paper, we will show how agent technology can be integrated with transactional technology to improve fault-tolerance.

Mobile agents are autonomous objects that are able to migrate from node to node in a computer network. When an agent decides to migrate to another node, the agent’s code, data and execution state[2] is captured and transferred to the next node, where it is initiated after arrival.

1. This work was funded by Tandem Computers Inc.; Cupertino (CA)

Agent execution proceeds in so-called stages [Sch97], where the operations of a stage are performed at a single node. Whenever an agent moves to a new node, this ends the current stage and begins a new one. The assignment of stages to nodes can be defined by means of a user-defined itinerary [LO97][GM] before the agent is launched, or on the fly by the agent logic taking into account the current system state [PS97][SBH96].

The use of mobile agents has been proposed for many application areas, including electronic commerce, systems management, or active messaging. In electronic commerce scenarios, for instance, agents autonomously go shopping on a user's behalf, do 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 applications require an agent to be executed "exactly once". For example, assume a user that launches a mobile agent 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. Of course, the user will only delegate this job to an agent if it is guaranteed that the agent does it "exactly once". In other words, independent of node and communication failures it must be ensured that the agent is never lost and hence will eventually get its job done. Moreover, failures may not cause the agent to perform operations more than once (e.g., to reserve and pay two seats instead of one).

The exactly once property has been already defined for RPC systems [Spe82], where it defines the failure semantics of a single remote procedure. In the context of mobile agents, a sequence of agent stages are to be considered rather than a single procedure. An agent execution is defined to be "exactly once" if the entire sequence of its stages is eventually performed, and all operations of each stage are executed exactly once.

In this paper, we will first describe a simple protocol based on transactional message queues (e.g. IBM MQSeries, see [BE97][Bla95]). This protocol already provides the "exactly-once" semantics as defined above. However, for many applications it is not sufficient to get the job "eventually" done but as fast as possible or even up to a certain deadline. In our reservation example above, the agent's status message should arrive at least before the date the business trip is scheduled. The problem with our simple protocol is that an agent may be blocked due to a node crash or network partitioning even if there are other nodes, where it could continue processing. Therefore, we propose an extension of this simple protocol to reduce the probability of agents to be blocked. The extended protocol allows a number of so-called observer nodes to be assigned to each stage. The observers monitor the stage node currently executing the agent and take over agent execution when this node becomes unavailable. A voting procedure integrated in commit processing ensures the "exactly-once" semantics. The protocol is currently implemented in Mole [Mole][SBH96], a mobile agent system developed at Stuttgart University.

The remainder of the paper is structured as follows. In the next section, we will describe the simple protocol and discuss the problems associated with it. Sec. 3 introduces a model for agent processing and gives an overview of the extended algorithm, which is subdivided in a voting protocol and a so-called selection protocol. These two protocols are described in detail in Sec. 4 and 5. Related work is discussed in Sec. 6, before the paper concludes with a brief summary.

2. Actually we distinguish between *strong* and *weak migration* [GV97]. While weak migration only transfers the code and data, strong migration also transfers the agent's execution state.

2. A Simple Solution

The exactly once property of mobile agents can be achieved in a simple way by using transactional message queues (e.g., see [GR94]). Message queues provide for asynchronous communication between processes residing on the same or different nodes, where the sender of a message *Puts* this message on a queue and its receiver *Gets* it from that queue. Transactional message queues provide for persistent messages and ensure the exactly-once delivery, i.e. once a queue manager has accepted a message, it will be delivered once, independent of node and communication failures. Moreover, the *Put* and *Get* operations can be performed within ACID transactions [HR93]. A message is only is only placed on or removed from a queue if the transaction including the corresponding *Put* resp. *Get* operation is committed. Transactional message queues are supported by a wide range of middleware technology (e.g., see IBM MQSeries, TUXEDO[GR94], Encina[Encina][GR94]).

Fig. 1 depicts how transactional message queues can be used to implement exactly once agents. Assume that an agent moves from node to node along route N_1->N_2->...->N_{k-1}->N_K. As an agent may visit the same node several times N_i and N_j ($1 \le i,j \le k$) may denote the same or different nodes. Assume further that an agent is stored in a message queue when it is accepted by the agent system for execution. Once the agent has been stored in the initial queue (Q_1 in our example), the owner of the agent can be informed that this agent - no matter what happens - will be eventually performed exactly once.

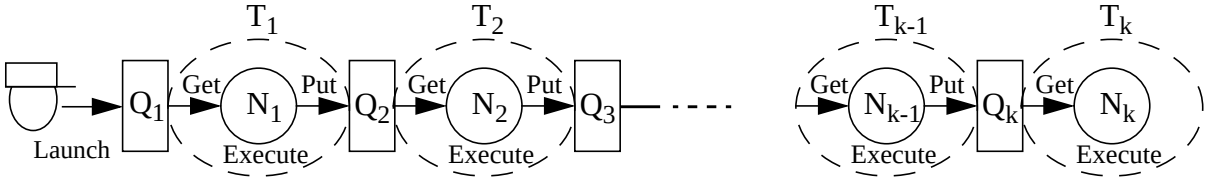

Figure 1: Simple implementation of exactly-once agents using message queues

Except N_k each other node is performing 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 on the input queue of node visited next. All three operations are performed within a transaction and hence build an atomic unit of work. So, if for instance transaction T_i aborts due to a node or transaction failure, recovery undoes all of the agent's effects at N_i and restores the agent in its original state in Q_i. Any effects in Q_{i+1} are undone also. After recovery is finished, N_i continues normal processing and will eventually execute this agent and then hand it over to its successor. Of course, the last node in the agent's itinerary, N_k, does not have to perform *Put*, it simply destroys the agent when the execution is finished.

The problem with this simple solution stems from the autonomy of agents. Due to this property there is no "natural" instance that monitors the progress of an agent. If a node crashes after the agent has been placed in its (local) input queue and before it is moved to the next queue, the agent is "caught" as long as the node is down. A partitioning of the underlying network may have similar effects. Note that this is different in client/server systems, where a client calling the operations of a server monitors the availability of this server. When it detects a server failure,

the client can continue processing by using alternative servers offering the same or similar services.

With the above protocol, there is no system entity that will notice that an agent is “caught”. Of course, the end user might notice it when the agent misses a deadline. This is a serious drawback since agent processing is blocked even if alternative nodes providing the needed services are available. Even if those nodes do not exist, some sort of exception handling should be performed, e.g., informing the user that the agent will most probably miss the deadline. Below we will extend the simple protocol described above to reduce the probability of agents to be blocked due to failures.

3. Models and Protocol Overview

In our model, agents are performed by nodes, that are interconnected by means of a communication network. Each node is assumed to have volatile and stable storage [Lam81], where the latter is never lost, independent of failures. Moreover, nodes are assumed to suffer from crash failures [Jal94] only. Network partitioning may occur due to communication failures. Both, the network and nodes are assumed to eventually recover from failures

The network provides for reliable message delivery: A message is delivered to its receiver, provided that the sender and receiver are in the same network partition and the receiver is up. Messages are authentic, and there are no lost, duplicated or out of sequence messages. Note that this type of failure semantics is provided by most reliable, connection-oriented communication protocols, such as TCP or LU 6.2.

The execution of an agent proceeds in a sequence of so-called ***stages***. The operations associated with a stage are entirely performed at a single node, and 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. This set initially includes a so-called ***worker*** node, which is responsible for executing the agent in this stage, the other nodes are ***observers*** monitoring the availability of the stage’s worker. When the worker fails, 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***[3], which defines a total ordering between the nodes belonging to the same stage. The initial worker of a stage will become the node with the highest priority. Fig. 2 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) failed and the node with priority 2 was elected the new worker.

What are the functional capabilities expected from observers? Ideally, an observer provides the same set of services an agent expects to find at the initial worker (e.g., a flight reservation service). However, an observer that offers no more than an environment for running agents is also acceptable. At such a node an agent can perform the exception handling mentioned above. For example, it can use the infrastructure services to find alternative servers, it can change its travel plans, or it can just move back to the user’s machine to report the problems and receive new directions.

To allow an observer to take over agent execution, it obviously needs a copy of the agent.

3. Node priorities are required for the voting and selection process.

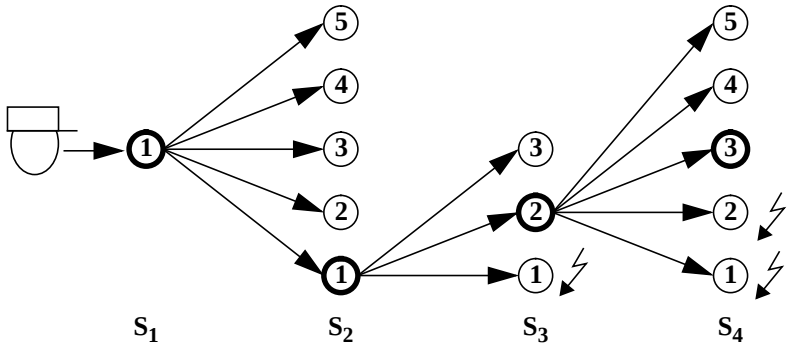

Figure 2: Execution of an agent in 4 stages

Therefore, in our scheme, a worker sends the agent not only to the (initial) worker but to all nodes of the next stage when it has finished processing. However, only the worker initiates agent processing, while the observers just do the monitoring for this stage. As in our simple protocol above, we use transactional message queues to move agents from one stage to another. Stage processing has the following structure (see Fig. 3): *Begin_Transaction*; *Get*(Agent); *Execute*(Agent); *Put*(Agent) *to* (AllNodesOfNextStage); *Commit*.[4]

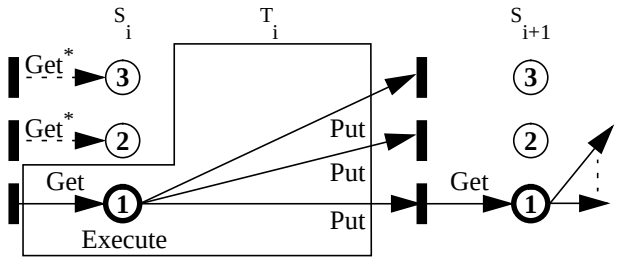

Figure 3: The transactional processing of an agent in a stage

The monitoring protocol ensures that an observer eventually recognizes when a worker becomes unavailable. In such a case, the observers of that stage select a new worker, which initiates a ***new*** stage processing transaction comprising the sequence of operations described above. Now, there is an obvious problem with this approach. Since the observers in general cannot decide whether an unavailable worker has crashed or is still active in a different partition of the network, it may happen that two or more nodes of the same stage execute the agent at the same time. However, the exactly once property of agents requires that exactly one stage transaction is committed per stage. In order to achieve this, we integrate a voting protocol into the two-phase commit (2PC) processing [GR94] of stage transactions: a transaction can only commit if a majority of stage nodes agree. It is also the responsibility of this voting protocol to make sure that all observers of a stage forget about the agent when a worker's stage transaction commits (see the *Get** operations in Fig. 3).

In Sec. 4, we will propose our voting protocol and show how it can be integrated in standard 2PC processing. We will assume an (X/Open Distributed Transaction Processing [X/O91] like)

4. Note that the *Get* operations of the observer nodes are not part of the transaction. If they would be included in the transaction, this would require all nodes of a stage to be available to execute an agent at that stage. Clearly, this increases the probability that an agent becomes "caught" rather than decreasing it. It is important to notice that having several *Puts* instead of one in the transaction does not increase the "caught" probability as the observers for the next stage can be determined on the fly from the set of available nodes.

architecture, which consists of transaction managers running the 2PC protocol and resource managers maintaining the recoverable data. Fig. 4 only depicts the components and interactions

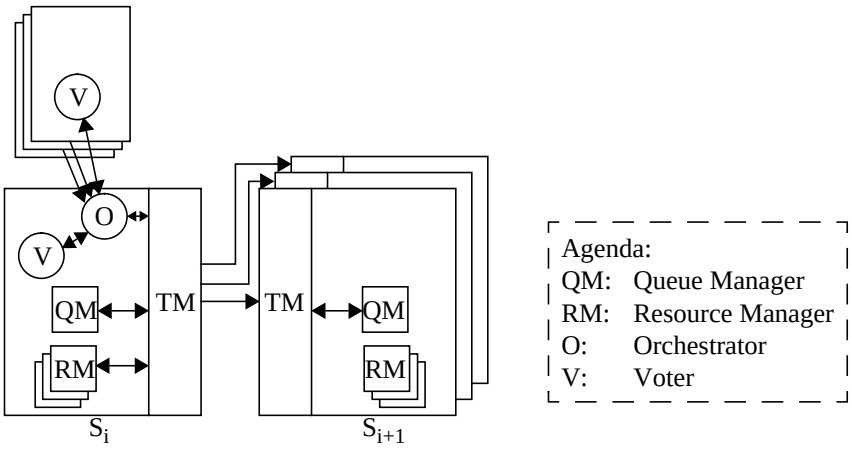

Figure 4: Components and interactions relevant to process a stage

relevant for the processing of stage S_i. When the worker of S_i calls *Commit*, the local TM initiates 2PC processing, which involves the worker itself and all nodes of S_{i+1}. During the commit procedure, each involved TM interacts with those local resource managers that were involved in stage processing. For example, at the worker node this is the queue manager associated with the worker's input queue and the other local resource managers (e.g. a DBMS) that have been involved in agent execution. In addition, the worker's TM interacts with another type of resource manager called ***orchestrator***. The orchestrator, which communicates with the so-called ***voters*** belonging to its stage, is responsible for orchestrating the voting procedure. Each stage node runs a voter, which determines and communicates the node's vote. The orchestrator and the voters of a stage communicate according to the voting protocol presented in the next section.

It is important to notice that the proposed architecture nicely separates voting and 2PC processing. From a TM's point of view, the orchestrator is just another resource manager, which provides the same interface as all other resource managers (e.g., an XA interface [X/O91][BE97]). Consequently, the voting procedure can be easily integrated in existing middleware systems, such as CORBA [OMG96] or X/Open compliant systems, just by implementing a new resource manager, or a new recoverable server to use the CORBA terminology.

Besides the voting procedure, a ***selection protocol*** is needed allowing the observers of a stage to select a new worker when they recognize that the old one failed. Since the voting during 2PC processing already ensures that only one stage transaction commits, the exactly once property is not jeopardized even if more than one new worker is selected. Actually, each observer that recognizes a worker failure could select itself without talking to the other observers. Consequently, the problem of selecting a new worker differs from the well-known election problem as defined in the literature (e.g., see [GM82]). For that reason we are using the term "selection" rather than "election" throughout this paper.

The selection protocol proposed in Sec.5 is a "light-weight" protocol, which usually selects one new worker, but also can end up with multiple workers in rare situations. Each worker and observer node runs monitor processes that do the monitoring and the selection of new workers

if needed.

4. Voting Protocol

In this section, we will focus on the voting protocol and its integration into 2PC processing. Instead of describing the well-known 2PC procedure, we will confine ourselves on presenting the interactions between the transaction manager (TM) and the local voting orchestrator (see Fig. 4).

As already stated in the previous section, from the TM's point of view the orchestrator looks like an ordinary resource manager. We assume that resource managers implement an XA-like interface with the following operations: *rm_prepare*, *rm_commit* and *rm_rollback*. The first operation, called in the first phase of 2PC, returns either *rm_yes* or *rm_no*, depending on whether or not the resource manager is able to prepare for commitment. In the second phase, the TM calls either *rm_commit* or *rm_rollback* depending on the transaction's outcome. Upon such a call a resource manager terminates the transaction accordingly and returns *rm_ack* to the TM.

The voting protocol is run between the orchestrator and the voters of a stage. While the orchestrator is located at a worker node only, there exists a voter at each stage node. When 2PC processing is started at the orchestrator (i.e., when *rm_prepare* is called), it issues vote requests to the voters of its stage and then collects the returned votes. Only if it receives a majority or yes votes, the orchestrator returns a *rm_yes* to its local TM, and a *rm_no* otherwise. In other words, only if majority of voters vote yes, the transaction can be committed. That is why only one transaction can commit per stage even if there is more than one worker.

We distinguish between two types of stable states, namely ***transaction states*** and ***stage states***. Both are stored in stable storage and thus are supposed to survive node failures. Transaction states are maintained by orchestrators, while voters maintain stage states. A transaction's state can be "Unknown", "Ready" or "Committed", while a stage's state may be "Unknown" or "Active". For both types of states "Unknown" means that no state information is stored in stable storage for the corresponding transaction or stage.

The state information of an "Active" stage is stored in a so-called ***stage record*** on stable storage. It contains the following information:

- Identifier of the stage, which consists of an (*AgentId*, *HopCount*) pair. *AgentId* is a globally unique agent identifier and *HopCount* is incremented whenever the agent is moved to the next stage.
- List of nodes participating in the stage. For each node the node's identifier and priority is included.

When an agent moves to the next stage, not only the agent itself but also the stage record of the next stage is *Put* into the input queues of the nodes associated with the next stage. Each stage node reads the stage record without actually removing it. Once the stage record has been *Put* into the message queue (on stable storage), the stage becomes "Active" at the corresponding node. Since all *Put* operations are performed in a single transaction (see Fig. 3), either all stage nodes are "Active", or none of them. Initially, the stage node with the highest priority becomes the worker, while all other nodes take over the observer role.

Voters and orchestrators are identified by globally unique node ids, i.e., a voter and orchestrator

residing on the same node have the same id. In analogy to 2PC processing, our voting protocol proceeds in two phases.

Normal Processing: Phase 1

Phase 1 of the voting protocol is initiated when an orchestrator receives an *rm_prepare* call from its local TM. First, the orchestrator sends a VOTE request to each voter of its stage. This request includes several globally unique identifiers: the id of the stage currently processed, the orchestrator's id, and the id of the transaction the orchestrator is currently involved in[5]. Then, the orchestrator waits for the answers, periodically resending the VOTE request to all stage nodes that have not yet answered.

To record its votes already given to orchestrators, a voter maintains a list called *OrchSet* in stable storage. Whenever the voter returns a yes vote, the identifier of the receiving orchestrator is recorded in *OrchSet*. Normally, *OrchSet* ends up with one node identifier. In the presence of failures, however, there might be several orchestrators competing for a node's vote.

A voter receiving a VOTE(*StageId, Tid, OrchId*) request for stage *StageId* determines its reply based on its *OrchSet*. If *OrchSet* is empty, the voter has not voted yes before. In this case, *OrchId* is added to *OrchSet* and a YES(*StageId, TId, VoterId*) reply is sent back to the orchestrator, where *VoterId* identifies the voter.

If *OrchSet* is not empty instead, there are obviously several orchestrators competing for the vote. To make sure that one of the them will eventually receive a majority of votes, our voting protocol prefers the orchestrator with the highest priority. Assume that *N* is the node with the highest priority in *OrchSet*. If *OrchSet* is not empty and *OrchId* has a lower priority than *N*, then the voter has already voted yes for a node with a higher priority. In this case, the voter replies with NO(*StageId, TId, VoterId*), i.e., *OrchId* loses the competition.

If *OrchSet* is not empty and *OrchId* has a higher priority than *N*, then the voter has already voted but only for orchestrators with a lower priority. If *N* is not the voter's node, the voter immediately sends back a COND_YES(*StageId, Tid, OrchSet, VoterId*) and then adds *OrchId* to its *OrchSet*. The semantics of this vote is that *VoterId* votes yes, provided that all nodes in *OrchSet* also vote YES.

If *N* equals the voter's node, there exists a local orchestrator, which has already initiated a competing voting procedure. Since *OrchId* has a higher priority than the local orchestrator, the latter one is supposed to give up. This, however, is only possible (and desirable) before the stage transaction at the orchestrator has entered the "Ready" state (i.e., before the orchestrator has got a majority of votes). To check the transaction's state, the voter sends a HIGHER_PRIO request to the local orchestrator, which returns either GAVE_UP to indicate its stage transaction has been aborted, or ALREADY_DONE if the transaction state is already "Committed" or "Ready". If ALREADY_DONE is returned, the voter sends a NO(*StageId, Tid, VoterId*) message to *OrchId*, and a COND_YES(*StageId, TId, OrchSet-{N}, VoterId*) message otherwise.

5. The transaction identifier is received in the *rm_prepare* call and is used here to match VOTE requests with the corresponding votes. Due to node failures it may happen that the same orchestrator starts several rounds of voting.

To record the received votes matching the curren *TId*, the orchestrator maintains three sets in volatile storage: *YesVotes*, *NoVotes* and *CondYesVotes*. When it receives a YES or NO vote, it includes the voter's id in *YesVotes* or *NoVotes*, respectively. If it receives a COND_YES, it adds the (*VoterId*, *OrchSet*) pair included in this message to *CondYesVotes*. Note that this conditional yes becomes a "real" yes after all nodes in *OrchSet* voted yes. In other words, if *OrchSet*-*YesVotes* equals the empty set, *VoterId* can be added to *YesVotes* and (*VoterId*, *OrchSet*) can removed from *CondYesVotes*. Obviously, this check has to be performed when the (*VoterId*, *OrchSet*) pair is added to *CondYesVotes* and whenever *YesVotes* is changed.

Once *YesVotes* contains a majority of votes, the orchestrator moves into the "Ready" state and then returns *rm_yes* to its local TM. Then it waits for the TM's commit or abort decision. Note that the *rm_yes* response is only a prerequisite for commitment rather than a commit decision. If a majority becomes impossible (i.e., at least half of the voters voted NO), the orchestrator returns a *rm_no* to its local TM, sends an UN_VOTE(*StageId*, *TId*, *OrchId*) message to all voters recorded in its *YesVotes* and *CondYesVotes* set, and then forgets the transaction. Note that the *rm_no* response forces the TM to abort the stage transaction.The orchestrator's node then changes from the worker to the observer role (see Sec. 5).

When the orchestrator receives a HIGHER_PRIO message, it replies ALREADY_DONE if its stage transaction is already "Ready" or "Committed". If the transaction is still in the "Unknown" state, it sends back GAVE_UP to the local voter and returns *rm_no* to the local TM. Furthermore, it sends UN_VOTE messages to all voters recorded in its *YesVotes* and *CondYesVotes* set before it forgets the transaction. As above, orchestrator's node then changes from the worker to the observer role.

An orchestrator receives a GIVE_UP request from the local voter if another stage node already committed its stage transaction (see below). Clearly, this message can only arrive while the receiving orchestrator resides in the "Unknown" transaction state. When GIVE_UP arrives, it immediately forgets the transaction, and returns *rm_no* to the local TM.

Normal Operation: Phase 2

If the TM commits the transaction, it issues *rm_commit* for each local participating resource manager. When *rm_commit* is called, the orchestrator atomically enters the "Committed" state and returns an *rm_ack* to the local TM. Subsequently, it sends a FORGET(*StageId*, *OrchId*) message to all voters of its stage and then waits for the acknowledgements to arrive. It periodically resends FORGET until it received an ACK from each voter. When the ACKs are complete, it moves to the "Unknown" transaction state before it forgets the transaction.

A voter receiving FORGET atomically goes into the "Unknown" stage state, i.e. the stage's stage record (together with the agent) is removed from the voter's transactional input queue in an atomic fashion. Subsequently, the voter sends back an ACK(*StageId*) message to the sender of FORGET and removes the stage's *OrchSet* from stable storage. If there happens to be a local orchestrator different from *OrchId*, the voter sends GIVE_UP to this orchestrator, causing the locally initiated stage transaction be aborted.

If the orchestrator receives *rm_abort* instead of *rm_commit* from its TM, it enters the transaction state "Unknown" and then sends UN_VOTE requests to all voters recorded in its *YesVotes* or *CondYesVotes* set. Then orchestrators node restarts the transaction. Voters receiving an UN_VOTE remove *OrchId* from their *OrchSet*, i.e., they withdraw their votes previously given

to *OrchId*. Note that this “unvote” mechanism is needed to allow a lower priority node to achieve a majority after some higher priority node gave up. Remember that a voter only votes (cond)yes if it has not already voted (cond)yes for some other node with a higher priority.

Failure Recovery

Once a voter has returned a vote to an orchestrator, it can expect either a FORGET or UN-VOTE response. When the voter times out while waiting on the response, it sends an INQUIRY message to the corresponding orchestrator. INQUIRY messages are sent periodically until FORGET (or GAVE_UP) is received, or each orchestrator recorded in the voter’s *OrchSet* returned a UN_VOTE response.

An orchestrator’s response on an incoming INQUIRY(*StageId*, *VoterId*) request depends on its current transaction state. If the transaction state is “Ready”, the orchestrator adds *VoterId* to its *YesVotes* set if it is not already included. This ensures that the identified voter will be notified accordingly as soon as the TM issues *rm_commit* or *rm_abort*. If the orchestrator is in the “Committed” state when receiving an inquiry, it responds with a FORGET message. If the orchestrator resides in the “Unknown” state, two cases must be distinguished: If there is no active transaction belonging to the stage identified by *StageId*, the orchestrator returns an UN_VOTE message. If there is an active transaction instead (i.e., voting is still in progress for the stage) the INQUIRY can be ignored. Let us briefly argue why. If *VoterId* is already in *YesVotes* or *CondYesVotes* or will be included at a later point in time, the identified voter will be informed during phase 2. Even if this is not the case, a future INQUIRY will eventually find no locally active transaction, causing an UN_VOTE to be returned to the voter.

When a node recovers from a failure, it reads the transaction and stage states recorded in stable storage. Orchestrator recovery only takes place if the transaction state is “Committed” or “Ready”. If the transaction is “Ready” after restart, the orchestrator waits until it is informed by the local TM about the transaction’s outcome, and then proceeds as described above. If the transaction is already “Committed” instead, the orchestrator sends FORGET to all voters of the stage and collects the ACKs. After having received all ACKs, it can enter the “Unknown” transaction state.

A voter only performs recovery if its stage state is “Active”. In this case, the voter periodically sends INQUIRY request to all orchestrators recorded in its *OrchSet.* It continues to send inquiries until it receives FORGET from some orchestrator, or it got an UN_VOTE from each orchestrator in *OrchSet*. It acts upon the received responses as described above.

Correctness Arguments

In the following, we will give some informal correctness arguments for the voting protocol described above. We will assume that the selection protocol ensures that there eventually exists a non-empty set of orchestrators (or workers). The objective of the voting protocol is to guarantee that exactly one of these orchestrators will commit its state transaction.

Let us first show that - given a non-empty set of orchestrators - exactly one of them will eventually enter the “Ready” transaction state. If there is only one orchestrator, it will get YES votes from all available voters. As soon as a majority of voters is available, it can enter the “Ready” state.

Now assume that there are several competing orchestrators, and O_1 is the one with the highest priority. When another orchestrator, say O_2, receives O_1's vote request, it has either already entered the "Ready" or "Committed" state, or it gives up. In the first case, O_2 has got a majority of votes, which allows no other orchestrator to move into the "Ready" state. In the latter case, O_2 sends UN_VOTE to its voters, allowing its local voter to send a YES vote back to O_1. All other voters either return a YES or COND_YES (O_2) message back to O_1, depending on the sequence O_1's and O_2's vote requests arrived. Since O_2 returned a YES vote, the COND_YES(O_2) votes can be interpreted as yes votes. Consequently, O_1 will eventually receive a majority of votes and thus can enter the "Ready" state.

If O_1 aborts its transaction, the "unvote" mechanism ensures that all voters will eventually withdraw their votes given to O_1 (or forget the stage). Therefore, also a lower priority orchestrator will get the chance to collect a majority of votes.

As shown above, if there are several orchestrators, exactly one of them will eventually enter the "Ready" state. This orchestrator's transaction will either commit or abort. In the case of commitment, all stage nodes forget the stage, and thus no other transaction of this stage will be able to commit any more. In the case of abort, the transaction becomes "Unknown" and its orchestrator starts a new transaction. In the latter case, as shown above, this or another orchestrator will eventually become "Ready". Consequently, exactly one orchestrator will eventually perform commitment.

5. Selection Protocol

In the previous section, we already pointed out that in addition to the agent also the stage record of the stage to be performed next, say *S*, is *Put* into the input queues of the nodes associated with *S*. Remember that all these *Put* operations are performed within the transaction of the pervious stage and thus are "all or nothing". Each stage node reads the stage record without removing it from its input queue and decides its initial role depending on the priorities recorded in the stage record. The node with the highest priority becomes the worker node, which then performs the sequence of operations already outlined in Sec. 3: *Begin_Transaction*; *Get*(Agent); *Execute*(Agent); *Put*(Agent, StageRecord) *to* (AllNodesOfNext-Stage); *Commit*. The other stage nodes are observers, which monitor the worker.

A worker, say *W*, periodically sends I_AM_ALIVE messages to the observers of its stage. If it receives an I_AM_ALIVE or I_AM_SELECTED (see below) message from another node, then there obviously exists a competing worker, say *W*'. If *W*' has a higher priority than *W*, *W* sends a HIGHER_PRIO request to the local orchestrator. If the response is GAVE_UP (see Sec. 5), *W* becomes an observer monitoring *W*'.

When an observer times out while waiting on the worker's I_AM_ALIVE messages, it assumes that the worker is not available any more and initiates the procedure for selecting a new worker. The selection protocol described below adopts the basic principles of the bully algorithm [GM82].

A node initiating the selection procedure sends ARE_YOU_THERE messages to all stage nodes with a higher priority. Available nodes (observers as well as workers) reply on this message with an I_AM_THERE message. If no reply arrives within a reasonable time, the

initiator is selected to be the new worker. The newly selected worker sends an I_AM_SELECTED message to all other stage nodes, and starts a new stage transaction comprising the sequence of operations sketched above. If the initiator receives a reply instead, it waits on the I_AM_SELECTED (or I_AM_ALIVE) of the new worker to arrive. When this message arrives, it starts monitoring the new worker.

In the presence of network partitioning, the protocol presented so far selects a worker in each partition, if two partitions are joined, two workers remain in the resulting partition. Note that this is not a problem since our voting protocol ensures that only one worker will commit.

Starting a transaction in a partition that does not include a majority of nodes is at least questionable. With a little modification of the protocol, starting transactions in partitions without a majority of nodes can be avoided: Observers getting an I_AM_SELECTED message are supposed to reply with an ACK, and the initiator of the selection protocol becomes the new worker only if it receives a majority of ACKs. Therefore, the initiator periodically sends I_AM_SELECTED messages until it receives either a majority of ACKs or an I_AM_SELECTED from a higher priority node. In the first case, it becomes the new worker, while it continues to be an observer in the latter case.

6. Related Work

In the field of mobile agents, only few researchers have considered aspects of transaction management and fault tolerance so far. In [Sch97], a stage model similar to the one in this paper is proposed. However, the paper focuses on a different aspect of fault-tolerance. Nodes are not assumed to be fail-stop but potentially non-deterministic. Fault tolerance is achieved by processing the agent on each stage node (in parallel) and to send the migrating agent to all nodes of the following stage. Stage nodes perform voting on incoming agents to determine a majority of equal agents. Only an agent from this majority is processed further. In [SK97], an agent-based transaction model is presented. Similar to our model, an agent executes a transaction while moving from node to node. To prevent the blocking of agents due to long lasting failures, the use of monitoring components is proposed. However, this paper purely concentrates on modelling aspects, and hence protocols or algorithms are not given.

In addition, there has been some related work in the field of transaction processing. The ConTract model [WR92][RSS97] also aims at the exactly-once property and similar to our approach only allows for forward recovery. A ConTract is defined by a script which is performed by a ConTract manager. A first prototype implementation, APRICOTS [Sch93], will allow the migration of scripts between ConTract managers, even in the case that the ConTract manager processing the script crashed, by using logging information written during the execution of the script to recover the state of the script on another node. However, there is currently no component which autonomously (and reliably) initiates the migration of a script to another ConTract manager if the ConTract manager executing the script crashes.

7. Summary

We have investigated how the exactly once property can be ensured in mobile agent systems. We presented a protocol guaranteeing this property, while reducing the probability for agent blocking. Moreover, we proposed an architecture that allows to integrate the protocol in standard transactional technology. In other words, the proposed mechanism can be realized on

top of conventional TP-monitors and transactional message queues. Currently, the protocol is under implementation in the Mole system. Future work will be to evaluate this protocol in terms of performance. Results are expected to be available in spring next 1998.

On the system level, the current protocol allows for forward recovery only. In other words, if a user wants to abort an agent, the potentially required compensation operations are not automatically triggered on the system level. Instead, the logic to perform compensations must be provided in the agent by the agent programmer. This goes in line with the experiences made with today's workflow systems that many operations can only be compensated in a very application specific manner and often requires the intervention of human users. However, some compensation can be done automatically, and hence future work will be to investigate, what concepts and protocols are needed to support compensation on the system level. We are confident that we can learn from the research conducted in the field of transaction models (e.g., Sagas [GMS87], open nested transaction [Wei91][WS91], etc.).

Bibliography

[BE97] Bernstein, P.; Newcomer, E.: "Principles of Transaction Processing". Morgan Kaufmann Publishers Inc, 1997

[Blak95] Blakeley, B.: "Messaging and Queuing Using the MQI", McGraw-Hill series on computer communications, 1995

[Encina] http://www.transarc.com/afs/transarc.com/public/www/Public/ProdServ/Product/Encina/index.html
http://www.transarc.com/afs/transarc.com/public/www/Public/ProdServ/Product/Whitepapers/index.html

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

[GM82] Garcia-Molina, H.: "Elections in a Distributed Computing System". In: IEEE Transactions on Computers, Vol. C-31, No. 1, January 1982

[GMS87] Garcia-Molina, H.; Salem, K.: "Sagas". In: Proc. ACM Conf. on Management of Data, pp. 249-259, May 1987

[GR94] Gray, J.; Reuter, A.: "Transaction Processing - Concepts and Techniques". Morgan Kaufmann Publishers Inc, 1994

[GV97] Ghezzi, C.; Vigna, G.: "Mobile Code Paradigms and Technologies: A Case Study". In: Mobile Agents, Proc. 1st Int. Workshop, MA'97. Springer, 1997

[HR93] Haerder, T.; Reuter, A.: "Principles of Transaction-Oriented Database Recovery.". ACM Computing Surveys, 15(4), 1993

[Jal94] Jalote, P.: "Fault Tolerance in Distributed Systems". Prentice Hall Inc., 1994

[Lam81] Lampson, B.: "Atomic Transactions". In: Lampson, B. et al (eds): "Distributed Systems - Architecture and Implementation", Springer-Verlag, 1981

[LO97] Lange, D.; Oshima, M.: “Java Agent API: Programming and Deploying Aglets with Java”. To be published by Addison-Wesley, Fall 1997; a working draft, “Programming Mobile Agents in Java, is avail-able at http://www.trl.ibm.co.jp/aglets/whitepaper.htm.

[Mole] Project Mole, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[OMG96] Object Management Group: “CORBA 2.0 specification”, ptc/96-03-04, 1996

[PS97] Peine, H.; Stolpmann, T.: “The architecture of the Ara platform for mobile agents.”. In: Mobile Agents, Proc. 1st Int. Workshop, MA’97. Springer, 1997

[RSS97] Reuter, A.; Schneider, K.; Schwenkreis, F.: “ConTracts Revisited”. In: S. Jajodia and L. Kerschberg (ed.): Advanced Transaction Models and Architectures (ATMA), Kluwer Verlag, 1997

[SBH96] Straßer, M.; Baumann, J.; Hohl, F.: “Mole - A Java Based Mobile Agent System”. In: Proc. 2nd ECOOP Workshop on Mobile Object Systems, dpunkt-Verlag 1996

[Sch93] Schwenkreis, F.: “APRICOTS - Management of the Control Flow and the Communication System”. In Proc. of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, October 1993

[Sch97] Schneider, F.: “Towards Fault-tolerant and Secure Agentry”. In: Proc. 11th Int. Workshop on Distributed Algorithms, 1997

[SK97] Morais de Assis Silva, F.; Krause, S.:”A Distributed Transaction Model based on Mobile Agents”. In: Mobile Agents, Proc. 1st Int. Workshop, MA’97. Springer, 1997

[Spe82] Spector, A.: “Performing remote operations efficiently on a local computer network”, Communications ACM, vol. 25, pp 246-260, Apr. 1982

[Wei91] Weikum, G.: “Principles and realization strategies of multi-level transaction management”. ACM Transactions on Database Systems, 16(1): pp. 132-180, March 1991

[WR92] Wächter, H.; Reuter, A.: “The ConTract Model”. In: A. Elmagarmid (ed), Database Transaction Models for Advanced Applications, Morgan-Kaufmann, 1992

[WS91] Weikum, G.; Scheck, H.: “Multi-level transactions and open nested transactions”. IEEE Data Engineering Bulletin, March 1991

[X/O91] X/Open DTP: “X/Open Common Application Environment”
“Distributed Transaction Processing:Reference Model”
“Distributed Transaction Processing: The XA Specification”
Reading, Berkshire, England: X/open Ltd, 1991