Fault Tolerance Issues in Mobile Agents

Qiao Xin(qxin@cs.ucsd.edu)
Yang Yu(yyu@cs.ucsd.edu)
Yu Xu(yxu@cs.ucsd.edu)
Zhanhai Qin(zqin@cs.ucsd.edu)

Computer Science & Engineering Department
Univ. of California, San Diego

1. Introduction

Mobile agents are programs which are dispatched from a source computer and run among a set of networked servers until they are able to accomplish their task. Mobile agents is an extension of client/server computing in which the client sends a portion of itself to the server for execution. The distinguishing feature is that they can migrate from server to server.

Agents roam on uncertain networks. During the life cycle, a lot of unpredicted things may happen. Errors may happen on the server, it may also happen during the network communication. The longer the path of the agent, the higher the possibility that it gets some troubles.

Thus it is important to make a mobile agent fault-tolerant. That is to say, the errors can be detected and recovered. In the following sections, we will discuss several ideas to implement fault-tolerant in mobile agents: primary-backup model, exactly-once mobile agents, sliding window, replicated agents with voting. We will also introduce a new architecture in this area, the ideas of the above models are used in different part of our new architecture.

2. Models overview

2.1 Primary-backup models

2.1.1 Traditional Primary-Backup protocol

In the traditional Primary-Backup protocol, a fault-tolerant service is implemented through the use of multiple servers. The state information of the service is fully replicated at each server. One of the servers is designated as the primary and the others are designated as backups. [3]

Normally, a client sends its request to the primary, which processes the request and sends a response to the clients. In order to preserve the consistency of the replicated service, the primary also sends to all the backups the database update that occur when each request is processed. When the primary fails, one of the backups takes over as the primary and notices the clients so that subsequent requests will be sent to new primary. In this model, the clients are driven by the servers in the system, they play no role in determining the next primary or in identifying faulty servers. [10]

2.1.2 Active clients primary-backup model

The active clients primary-backup protocols extends the traditional primary-backup model, each clients maintains an ordered list of servers and uses it to detect faulty servers and elect new primary server. In [10] they argue that their AC model can tolerate three types of failures: crash failures, send-omission failures and receive-omission failures of servers and clients and using minimal replication.

2.2 Exactly-once model

2.2.1 Introduction

An important issue of fault tolerence is to keep the exactly-once property. Obviously many of mobile agent applications require an agent to be executed exactly once. For example, assume a user that launches a mobile agent to make a hotel and flight reservation for a forthcoming business trip. The agent is expected to make both reservations if possible, and in any case to return a status message back to the user. Of course, the agent must also guarantee that it makes each reservation exact once and cannot be caught by a network partitioning or node failure [12].

2.2.2 Agent execution model

In their agent execution model, tasks are assigned to agents, which perform them autonomously. To execute its task a mobile agent may exploit the services provided by the various nodes of a computer network. Although there could be a couple of candidates, an agent moves to one node before accessing the node’s resources, i.e., agents only interact with local services.

Agent execution proceeds in steps, where a new step is initiated whenever an agent migrates to the next node in its itiner-
A step of an agent is defined to be the set of operations performed by the agent while it visits this node. In the execution model, resources are encapsulated in resource managers. So each step may change the agent’s state as well as the state of the local resources.

Now we give the definition of exactly-once. Let \( L(I) \) be the number of nodes in the agent’s itinerary \( I=\{N_1, N_2, ..., N_{L(I)}\} \) and \( S_i \) be the step to be performed on node \( N_i \) \((1 \leq i \leq L(I))\). Then the execution of an agent is defined to be exactly-once if

- the agent executes step \( S_i \) before step \( S_{i+1} \), \( 1 \leq i < L(I) \), and
- each step \( S_i \), \( 1 \leq i \leq L(I) \), is executed exactly once, independent of communication and node failures.

### 2.2.3 Protocol overview

The exactly-once property of mobile agents as defined above can be achieved in a simple way by using transactional message queues. Message queues provide for asynchronous communication between processes residing on the same or different nodes. While an agent is going to complete the operations on a node (sender), it will wrap up its code, data and execution state into a message package and put it on a queue. The successor of the sender will get it from that queue. Transaction message queues 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.

Figure 1 depicts how transactional message queues works. Assume that an agent moves from node to node along route \( N_1 \rightarrow N_2 \rightarrow ... \rightarrow N_{k-1} \rightarrow N_k \) where \( N_i \) and \( N_j \) \((1 \leq i, j \leq k)\) may denote the same or different nodes. Once the agent has been stored in queue \( Q_i \), it is guaranteed that this agent will be performed exactly once eventually.

![Figure 1. Simple implementation of exactly-once agents using message queues](image)

Every node, except for \( N_k \), performs the following sequence of operations: \textit{begin\_transaction}; \textit{Get(Agent)}; \textit{Execute(Agent)}; \textit{Put(Agent)}; \textit{Commit}. \textit{Get} removes an agent from the node’s input queue, \textit{Execute} performs the received agent locally, and \textit{put} places it directly into the input queue of the node to be 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 also undone. After recovery is finished, \( N_i \) continues normal processing and will execute this agent eventually and then hand it over to its successor.

In contrast to client/server systems, where a client calling the operations of a server monitors the availability of this server, there is no “natural” instance that monitors the progress of an agent. In the above protocol, if a node crashes after an agent has been placed in the node’s (local) input queue and before the agent is moved to the next queue, this agent is “caught” as long as the node is down even if there are other candidates available. In the next part we will introduce the enhanced protocol to reduce the probability of agents to be blocked due to failures.

#### 2.2.3.1 Enhanced protocol

To reduce the probability of agents to be blocked due to node or communication failures, they introduced the concept of \textit{stages}: for each step there is a non-empty set of nodes, called a stage, which can perform that step alternatively. Each stage node is associated with a priority, which defines a total order between the nodes belonging to the same stage. Each stage initially selects the node with the highest priority as \textit{worker} node. The other nodes are \textit{observers} monitoring the availability of the stage’s worker. When observers recognize that a worker becomes dead, they will select a new worker among themselves by applying selection protocol. Figure 2 shows a 2-stage execution of an agent. For example, stage \( S_j \) is associated with one worker, and 4 observers. In \( S_2 \), the node with the highest priority 1 failed and the node with priority 2 was selected to be the new worker.

![Figure 2. Execution of an agent in 2 stages](image)

To allow an observer to take over agent execution, 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.
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. In order to achieve the exactly-once property, a voting protocol was integrated into the two-phase commit (2PC) processing of step transactions: a transaction can only commit if a majority of stage nodes agree.

The transaction processing architecture they built on consists of transaction managers (TM) running the 2PC protocol and resource managers (RM) maintaining the recoverable data. 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.

2.2.3.2 Voting protocol

Resource managers implement an interface with the following operations:

- **rm_prepare**
  As the first phase of 2PC, it returns either rm_yes or rm_no, depending on whether or not the resource manager is able to prepare for commitment.

- **rm_commit & rm_rollback**
  In the second phase, the TM issues 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 voter of a stage. Phase 1 of the voting protocol is initiated when an orchestrator receives rm_prepare from its local TM. First, the orchestrator sends a VOTE request to each voter of its stage: VOTE(StageId, OrchId, TId). This request includes the id of the stage currently processed (StageId), the orchestrator’s id (OrchId), and the id of the transaction the orchestrator is currently involved in (TId). The orchestrator waits for the answers, and whenever the voter returns a YES vote, the identifier of the receiving orchestrator is recorded in OrchSet maintained by each voter.

A voter receiving a VOTE request 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 several orchestrators competing for the vote. 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 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.

Once after applying the COND_YES semantics YES votes are a majority of votes, the orchestrator returns rm_yes.

In phase 2, if the TM commits the transaction, it issues rm_commit for each local participating resource manager.

If the orchestrator receives rm_abort instead of rm_commit from its TM, it sends UN_VOTE requests to all voters to whom it voted YES or COND_YES. Then the orchestrator’s node restarts the transaction. Voters receiving an UN_VOTE remove OrchId from their OrchSet, i.e., they withdraw their votes previously given to OrchId. This “unvote” mechanism allows a lower priority node to achieve a majority after some higher priority node gave up.

2.2.3.3 Monitoring and selection protocol

The worker of a stage periodically sends I_AM_ALIVE messages to the observers. 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.
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 to 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 step transaction comprising the sequence of operations sketched above. If the initiator receives a reply instead, it waits for the I_AM_SELECTED of the new worker to arrive. When this message arrives, it starts monitoring the new worker as other observers do.

2.2.4 Comments

From the approaches we introduced above, Kurt and Markus [1]'s exactly-once model does tolerate network partitioning. But they didn’t consider the case that besides the crash of workers, observers may also crash. And since in general the observers in one stage are more than the workers, the possibility of the crash of the observers is much larger than that of the worker’s crash. On the other hand, when orchestrators return rm_commit when they decide that the YES is the majority, the total number of voters is supposed to be fixed. But actually it may be less because some of the voters may have crashed. So we propose a remedy as follows: The observers of a stage periodically send I_AM_ALIVE messages to the worker. When the worker times out while waiting on the observers’ I_AM_ALIVE messages, it assumes that the observer is not available any more and when the worker’s TM calls rm_prepare it won’t consult those dead observers any more. Those observers alive can be maintained dynamically in a set in volatile memory.

2.3 Sliding window model

2.3.1 Introduction of Sliding window model

As introduced at the beginning of the paper, Mobile agents are itinerant programs that move from host to host in a network. In the mobile agent world, the agent can predefine the order of the hosts that it will visit, or the order can be calculated dynamically. Sometimes, mobile agents can fork multiple copies of itself on a host. But in most cases, the path of the agents is a loop. It starts from the client, travel through a set of service provider, and ends at the client. Ma suggested a fault-tolerance model which works when there is a no forking during the itinerance of the agent[9]. In this model, the mobile agent keep an original copy at the client, and during the itinerance, it also keeps a certain number of backups on the hosts that it has recently visited. When the mobile agent travels to the next host, the last backup is discarded. Since the number of backups is limited, and the backups move when the agent moves, this model is called sliding window model.

2.3.2 Architecture of sliding window model

2.3.2.1 Assumption

The path of the mobile agent is a loop.

2.3.2.2 How this model works

When only the fault-tolerance properties of a mobile agent is considered, the mobile agent (and its backups) can be represented by the following tuple: MA(H, L, O, P, T, W). H is the name (or identification of the mobile agent), L is the length of the sliding window, O is the order of the host in the window, the higher the order, the more responsibility that this backup takes for recovery. P records the hosts that the agent has visited, say, if H=Vi, P = [Vi, V2, ..., Vi-1]. T is the maximum time allowed for an agent to work on one host. W is the time that backup waits, by default is T*(L-O), if there is no messages from succeeding hosts in W time, the backup assumes that they all crashed and will starts recovery.

The host that the mobile agent is currently running on is called active host. The hosts that the backups reside are called backup hosts. When the mobile agent is created by the client, one original copy is saved there in case that every backups in the path all crashed. (Fortunately in most cases, this copy will not be used.)

Now we only consider the general backup and recovery procedure. When the mobile agent finished its job on host Vi, suppose the next host that it will visit is Vi+i (predetermined or dynamically computed), Vi build the connection to Vi+i, and send the mobile agent to Vi+i. If Vi get a RECEIVE_OK message from Vi+i, it shows that the transmission is successful, thus Vi saves the state of mobile agent (H = Vi, O = L), and send a RECEIVE_OK message to all hosts in the sliding window. Those hosts who received a RECEIVE_OK message deduct its order by 1. The waiting time W is also extended. If O is deducted to 0, the backup is removed from the host (discarded out of the sliding window). On the other hand, if Vi cannot get a RECEIVE_OK message from Vi, it rebuilds the connection with Vi+i and send out the agent again. Depends on the policy of the agent, it continue trying or select another host to visit if it fails to connect Vi+i again. If one host wants more time than T to do its job, it sends a REACTIVATE message to all the hosts in the current window before T comes, so that the backup agents on those hosts restart the timer. When the destination is arrived, the destination host (the client) sends a DESTINATION message to all the hosts in the sliding window, indicating that the travel is finished, so that all the backups could be discarded.
2.3.2.3 Recovery

Due to the complexity of the network environment, the mobile agent or the hosts that it visit may crash during the itinerance of the mobile agent. In the sliding window model, when fault happens, no actions are performed immediately. However, since each backup agents keep a timer, they will wake up when the time comes. The higher the order, the shorter the time that they wait(by default, W=T*(L-O)). If one backup agent has not received any RECEIVE_OK message or the REACTIVATE message during the past W time, it assumes that not only the current active agent crashed, but all the backup agents with a higher order crashed too(otherwise, it should receive the MS_LoST message from one of them). Thus this backup agent will reactive itself again, then reset the order O and waiting time W. It will also send MA_LOST message to all the backups behind it, so that they can readjust their position in the sliding window.

2.3.3 Comments

This model works more than one hosts crash. Only when all the L backups crash, does the source need to be involved. The tradeoff is the space spent to save the state of backups on the hosts. The larger the sliding window, the better the recovery performance, however, it also require more space on the hosts.

In this model, there are only a few additional communication messages. When fault happens, there are some additional computations needed to resend the agent. Since always the newest backup is used for the recovery(the newest the backup, the shortest the wait time), these computations are necessary and have to be done in any other models.

Although this model assumes that the path of the agent is a loop, it also works when the path is a chain(the destination needs not to be the source). However, when there are branches and merge points, or one host appears more than once in the path, this model should be modified.

2.4 Replicated Agents with Voting

2.4.1 Introduction

Replication and voting can be used to mask the effects of executing an agent on a faulty processor. However, this needs the authentication of the votes since faulty processors that do not execute agents may spoof and cast bogus votes. The authentication can be implemented by hardware, where each component that can vote has an independent and separate connection to the voter. But there may be no correspondence between the physical communications lines, the replicas, and the voter in the network, authentication must be implemented in other ways.

2.4.2 Protocols based on Authentication Chains

Let's call the sequence of processors on which the agent is executed trajectory, denoted by S(source), P2(stage2), P3(stage3), ..., D(destination, may be the same as S). To tolerate faulty processors(other than the source and destination) in a trajectory, we replicate each stage except the source and destination. Each processor in stage i takes as its input the majority of the inputs that it receives from the replicas comprising stage i-1 and each processor in stage i sends its output to all of the processors that it determines comprise stage i+1. For voters in stage i processors to determine their electorate, the processors comprising stage i-1, each agent carries a privilege, bogus agents can be detected and ignored by the destination.

In this scheme, the privilege is based on pedigree: all processors are assumed to know a priori the identity of the source, agents carry(unforgeable) certificates describing their trajectories, and voters use that information in order to reject bogus agents.

Let [A]p denote a text A that has been cryptographically signed by processor p. A digit certificate [(p:P)->(q,Q)]p, called a forward, will accompany the agent sent by a processor p, a member of electorate P, to a processor q, a member of electorate Q. Notice that if p is non-faulty, then p is the only processor that can construct forward [(p:P)->(q:Q)]p whose validity can be checked by any processor.

The processors in stage i-1 know what its electorate Pi is since any agent executing on a stage i-1 processor is supposed to be sent to all processors in Pi. Therefore, if each agent carries with it the forward [(r:Pi-1)->(p:Pi)]r when sent to p by r, Pi becomes to all processors in stage i and beyond. Sender authentication can be implemented with digital signatures. Once the voter at a processor q in stage i+1 (i) knows Pi and can authenticate the sender of each message that q receives, it can select some agent that was sent by a non-faulty processor p of stage i. The selection is done as follows:

1) The voter uses sender-authentication to reject any agent that is not from a processor in electorate Pi. This foils attacks by faulty processors that are not in the previous stage.

2) Of those agents that remains, the voter selects any agents for which it has received |pi|/2 equivalent replicas(replicas of an agent that differ only in the sets of forwards they carry). This masks the effects of faulty processors from the previous stage.

The protocol is as follows.
1) Source processor $p$ sends agents to each of the processor in stage $P_2$, and the agent sent to processor $q$ carries the forward $[(p:S)->(q:P_2)]p$.

2) The voter at each processor $p$ of stage $i$:

   i) receives agents from processors comprising stage $i-1$ (and perhaps from faulty processors elsewhere);

   ii) discards any agents that do not carry a suitable set of forwards. An agent will carry enough forwards so that a voter can check that the trajectory of the agent started at the source and passed through a sequence of stages, where a majority of the processors in each stage agreed on the electorate of the next stage and what agent to send to the processor on that stage. Fault processors can not produce an agent with such a set of forwards.

   iii) determines whether a majority of the remaining agents are equivalent and, if so, augments that agent's set of forwards to include the forwards of all agents in the majority.

3) When an agent is ready to move from a processor $p$ to next stage $i+1$, for each processor $q$ in the next stage, the forward $[(p:P_i)->(q:P_{i+1})]p$ is added to the set of forwards carried by that agent, and the agent is digitally signed and sent by $p$ to $q$.

### 2.4.3 Comments

In the above protocols, the message size grows linearly with the number of replicas and stages. In addition, there is more to implementing fault-tolerant agents than the a protocol used by voters authenticate agents comprising an electorate. The agent replicas must somehow find independent processors running the services they require, for example, the question of ensuring that replicas don't diverge must addressed.

### 3. Proposal

After studying all the above models, We give our models for fault-tolerance in mobile agent.

#### 3.1 problem definition

We define the problem we want to solve as the following:

A client needs to get $n$ different kinds of services from $n$ hosts. The client sends out a mobile agent to visit these $n$ hosts. After the mobile agent visit all the host and gets all the services, it will return the result to the client.

During the computation on $n$ hosts, the hosts can crash. We need to figure out a fault-tolerance model under this situation.

The client can specify:

(1) whether a certain service can be skipped. If a certain service can't be skipped, when the mobile agent can't get the service in the internet, it'll report to the client and will not continue to run to get other service. If a certain service can be skipped, when the service is unavailable due to the crash of the hosts, the mobile agent can skip the service and continue to run on other machines to get other services.

(2) Whether a certain service must be first finished before another, that is, the order between the $n$ services. This can be expressed by an acyclic graph.

(3) alternative hosts for services. If a host providing a certain service is unavailable, the mobile can go to other alternative hosts providing the same service specified by the client.

(4) The maximum time that can be allowed on a certain service.

We can provide a script language with that the client can easily write its rules. We call the rules the clients write a specification.

### 3.2 our models

To solve this fault-tolerance problem, we propose three models, which vary in their complexity and fault-tolerance capability.

(1) Naive home-based model

The first one is quite straightforward. Whenever the mobile agent gets a service, it'll send a copy of its current state back to the home of client.

The disadvantage is also straightforward. It requires the home machine online during the whole computation; the home machine also has the fault-tolerance problem during the whole computation, it could be the bottleneck.

(2) Agent-based model

We extend the first model a little further. Each time the mobile finishes with a service, it'll send a copy of the new states back to an agent machine representing the client instead of directly sending back to the client.

The advantage is that the client machine is freed of keeping states for its mobile agent, and can be disconnected during the whole computation. Also, the agent machine provided by some trusted social organization has good performance than
the ordinary client machine. The disadvantage is still the fault-tolerance issues on the agent machine.

(3) Active mobile with sliding-window model

In this model we use a group of machines to form a mothership which provides secure backup mechanism for the mobile agent.

Inside the mothership, primary-backup model is used to provide fault-tolerance. Each time the mobile agent finishes a service, it'll send one copy of the new state back to the mothership and then travel to another host to get another new service. The client needs only to launch the mobile agent into the mothership at the very beginning. During the whole computation, the client can be disconnected off the internet, finally it'll get the result from the mothership either by explicit polling the mothership or be noticed by the mothership when the client is connected again.

Obviously, this model's network traffic is twice of the optimal traffic in the normal cases in which there are no host crashes. In case of host failure, this model has very little traffic, the mothership will notice a host failure by not receive message from mobile agent, it can then launch a new mobile agent to continue with the most recent computation. It avoids the mess communication among the same multiple agents running on different hosts in other models.

To reduce the traffic in the normal cases, the client can specify the interval of sending backup copies to the mothership. If the interval is 1, this model is just the above one we discussed. If the interval is n, the model has no fault-tolerance at all. We think that an interval from 2 to 6 is reasonable. Also, the client can specify the interval in its specification.

To reduce the traffic in the normal case further more, we integrate sliding-window model in the mobile agent's traveling. When the mobile agent gets a service, it'll travel to another host and leaves rear guard agents behind it. The size of sliding-window can be specified by the client in the specification. The rear guards are responsible for:

(1) launching a new agent if a failure cause an agent to vanish and,(2) terminating itself when its function is no longer necessary(because the agent it protects is itself ready to terminate).[2].

With sliding-window, we have got two advantages:

(1) if the host machine hx where the mobile agent is running suddenly crashes and the host hy where its rear guard resides is still alive, the rear guard can launch the new mobile agent to the host hx at a later time. This is especially important if the interval the client chooses is relatively big. For example, if the interval is 4, without rear guard, the mothership has to launch a new mobile agent, but the new mobile has to repeat 4 jobs again, a great waist of time.

(2) If the interval is 1, the mothership will have the most recent copy at any time. Even in this case, in the same situation as in (1), if the cost from the nearest rear guard to the crashed machine hx is less than the cost from the mothership to the crashed host hx. We still gain.

As for the relationship between the mobile agent with the mothership. We use two different way to efficiently handle two different cases:

(1) If the machines that form the mothership are all in the same LAN and has good stability, we use traditional communication between the mothership and the mobile agent, that is, we use the restricted client model. Moreover, the group of servers in the mothership uses broadcast.

(2) If the machines that form the mothership are distributed on the internet and their stability are also in question, we use active client model proposed in [10], that is, the mobile agent has an ordered list of the servers in the mothership. In case of primary server failure, the mobile agent will connect an available backup server. The communication between servers will use multi-casting. In paper [3],[10], they use either end-to-end communication or broadcast, in this case, it's sure that multi-casting makes more sense.

To facilitate the client, there are some kinds of name servers. So the client can first connect to the name servers, get the names of the hosts which provide the service it needs, and then writes them into its specification, launches the mobile agent into the motherhsip.

Compared to other models, our model has several advantages.

First of all, the concept is very simple. The mothership provides a level of abstraction to the mobile agent, so the mobile agent will focus on its own computation need, spend less time and effort on fault-tolerance.

Second, the model is very flexible. The client can get different degree of fault-tolerance very easily, just specifies the interval value and the size of the sliding window. Also client has complete control of the behavior of mobile agent, it just writes rules in its specification.

Third, it has high fault-tolerance. The mothership provides high fault-tolerance to the mobile agent. Even it at some time all the hosts crashed at the same time, the mothership still can launch a mobile agent. In other models, either there are no ways to handle this problem, or they have to do the computation from scratch.
Fourth, its overhead (the number of redundant servers, the communication traffic) is very low because the combination of sliding window and the backup interval specified by the client.

Reference

[1]“In Implementing Primary-Backup Protocols”  Navin Budhiraja, Keith Marzullo, 1995 IEEE( 1063-6374/95 )


