A Progressive Fault Tolerant Mechanism in Mobile Agent Systems

Michael R. LYU and Tsz Yeung WONG
Computer Science & Engineering Department
The Chinese University of Hong Kong
Shatin, Hong Kong
{lyu,tywong}@cse.cuhk.edu.hk

Abstract

We present the approach of deploying cooperating agents to detect failures as well as recover services in a mobile agent system. In additional to server failure detection, we use cooperating agents to handle agent failure detection. Two types of agents are involved. One is the agent performing the computation delegated by the owner, which we call the *actual agent*. Another is the agent that monitors the actual agent, namely the *witness agent*. We introduce a protocol by using a message passing mechanism between these two kinds of agents to detect agent failures and recover agent services. This approach can handle server failures, agent failures, and failures in message passing. It is capable of detecting and recovering most failure scenarios in mobile agent systems. We describe the design of our witness agent approach to mobile agent systems, and conduct reliability evaluation for our approach. The evaluation results show our approach is a promising technique in achieving mobile agent system reliability.

Keywords*:* Software Agents, Fault Tolerance, Recovery, Modeling, Measurement.

1 INTRODUCTION

Mobile agents are autonomous objects capable of migrating from one server to another server in a computer network [1]. When an agent travels to another server, the agent's code, data as well as execution state are captured and transferred to the next server. It is re-instantiated after arrival at the next server. The ability to roam the net is provided by a middle-ware platform, a mobile agent execution environment (e.g. Aglets [2], Concordia [3] and Mole [4]).

Reliability as well as fault-tolerance are vital issues for the deployment of a mobile agent system. A number of research work is done in these areas. Some researchers adopt the use of *replication* as well as *masking* [5, 6]. The idea is to use replicated servers to mask the failures. When one server is down, we can still use the results from other servers in order to continue the computation. The advantage of this approach is that the computation will not be *blocked* when a failure happens.

However, this fault-tolerance scheme is expensive since we have to maintain multiple physical servers for just one logical server. Since a failure is a rare event, it is not cost-effective to maintain multiple servers. Moreover, every replicated server has its own data, and the data in all the replicated servers must be consistent among themselves. On the other hand, the computation on different servers may not produce the same and correct result. Thus, it is a tough task in preserving server consistency, especially when the servers are widely separated, since the latency of the network will affect the speed of consistency checking as well as preservation. Alternatively, our approach focuses on the agent failure detection and recovery.

Our approach is rooted from the approach suggested in [7]. We distinguish two types of agents. One type is performing the required computation for the user. We name it the *actual agent*. Another type is to detect and recover the *actual agent*. We call it the *witness agent*. These two types of agents communicate by using a peer-to-peer messages passing mechanism. In addition to the introduction of the *witness agent* and the messages passing mechanism, we also need to *log* the actions performed by the *actual agent* since when failures happen, we need to abort uncommitted actions when we perform *rollback recovery* [8]. We also use *checkpointed data* [9] to recover the lost agent.

The key difference between the protocol suggested in [7] and our protocol is that the former depends on a *reliable broadcast*, while we allow the network to be unreliable. That is, we can handle the failures in transmission of messages as well as the lost of the agent in the network, for example in network partitioning. In [7], the protocol uses message broadcasting with a lot of redundant messages. Our message passing mechanism, on the other hand, is a peer-to-peer one, so we can save a lot of redundant messages. Moreover, our protocol handles the failures of the *witness agents*.

2 SERVER FAILURE DETECTION AND RECOVERY

The server failure is much easier to be detected and recovered than the agent failure. Nevertheless, server failure detection as well as recovery are important issues in the design of a reliable mobile agent system. An agent requires a server to be hosted and an environment to execute. If the hosting server fails, the agent will be lost as an agent is just a piece of running program. On the other hand, the agent has manipulated objects (or data) in the server. These objects in the server will become inconsistent if the modifications done by the agents are not handled properly. We have to tackle this inconsistency problem. Moreover, if the server to which the agent migrates fails, the agent cannot travel to that server.

Since a server hosts an agent and the agent manipulates objects on the server, we have to log every action of the agent involving the modifications of the objects in the server. If a failure happens, all the *uncommitted* transactions done by the agent should be *aborted*. Hence, while the server is restarting, we have to inspect the log on the permanent storage, and undo all the uncommitted changes. During the recovery of the server, we cannot recover any lost agents since it is impossible for a server to re-instantiate an agent that is foreign to it.

If the agent cannot detect whether the target server is available or not, we may lose it when sending it to a failed server. Therefore, we have to implement the ability to detect the availability of a server for the mobile agent. We have implemented a method similar to *ping* for this purpose. With this implementation, an agent decides to wait in the current server if the target server ahead is failed. The agent continues waiting until the target server becomes available. In this implementation, the agent can continue its itinerary. However, while the agent is waiting, there is a chance that a failure happens to the server where the agent resides. In this case, we require an agent failure detection and recovery mechanism. This is covered in Section 3.3

Our mechanism to detect and recover a server failure is to launch a daemon in a machine, which is not error-prone. This daemon is to monitor the availability of all the servers. We name this daemon the *server monitor*. The server hosting this daemon is not a server responsible for receiving and executing any agent; it is an independent server which is not vulnerable to failures. The advantage of this approach is that it is easy to implement. However, we may encounter the problem of *single point failure*. In order to ease this problem, we can introduce more backup worker servers. The worker servers will monitor the primary server. If the primary one fails, one of the workers will replace the primary one, by launching the daemon and replacing the primary server.

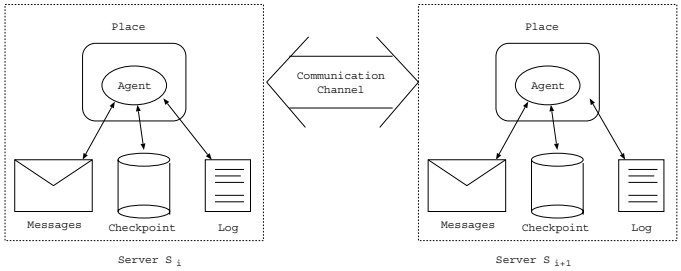

Figure 1. The server design

3 AGENT FAILURE DETECTION AND RECOVERY

3.1 System Architecture

In our agent system design, in order to detect the failures of an *actual agent* as well as recover the failed *actual agent*, we designate another type of agents, namely the *witness agent*, to monitor whether the *actual agent* is alive or dead. In addition to the *witness agent*, we have to design a communication mechanism between both types of agents. In our design, agents are capable of sending messages to each other. We call this type of messages the *direct messages*. The *direct message* is a peer-to-peer message. Since a *witness agent* always lags behind the *actual agent*, the *actual agent* can assume that the *witness agent* is at the server that the *actual agent* just previously visited. Moreover, the *actual agent* always knows the address of the previously visited server. Therefore, the peer-to-peer message passing mechanism can be established.

There are cases that the *actual agent* cannot send a direct message to a *witness agent* for several reasons, e.g., the witness agent is on the way to the target server. Then, there should be a *mailbox* at each server that keeps those unattended messages. We call this type of messages the *indirect messages*. These indirect messages will be kept in the permanent storage of the target servers.

Every server has to log the actions performed by an agent. The logging actions are invoked by the agent. The information logged by the agent is vital for failure detection as well as recovery. Also, the hosting servers have to log which objects have been updated. This log file is required when performing the rollback recovery.

Last but not the least, when a server failure happens, we have to recover the lost agent due to the failure. However, an agent has its internal data, which may be lost due to the failure. Moreover, if we allow the agent to start computing from the starting point of the itinerary, the *exactly-once* [10] property will be violated. Therefore, we have to *checkpoint* the data of an agent as well as *rollback* the computation when necessary [8]. We require a permanent storage to store the checkpointed data in the server. Moreover, we log messages in the log of the server in order to perform rollback of executions. The overall design of the server architecture is shown in Figure 1.

3.2 Protocol Design

Our protocol is based on message passing as well as message logging to achieve failure detection. Assume that, currently, the *actual agent* is at server S_i while the *witness agent* is at server S_{i-1}. Both the *actual agent* and the *witness agent* have just arrived at S_i and S_{i-1}, respectively. We label the *actual agent* as α, and the *witness agent* as ω_{i-1}.

We discuss the behavior of the *actual agent* α first. The actual agent plays an active role in this protocol. After α has arrived at S_i, it immediately logs a message, log^i_{arrive}, on the permanent storage in S_i. The purpose of this message is to let the coming *witness agent* know that α has successfully landed on this server. Next, α informs ω_{i-1} that it has arrived at S_i safely by sending a message, msg^i_{arrive}, to ω_{i-1}.

α performs the computations delegated by the owner on S_i. When it finishes, it immediately *checkpoints* its internal data to the permanent storage of S_i. Then, it logs a message log^i_{leave} in S_i. The purpose of this message to let the coming *witness agent* know that α has completed its computation, and it is ready to travel to the next server S_{i+1}. In the next step, α sends ω_{i-1} a message, msg^i_{leave}, in order to inform ω_{i-1} that α is ready to leave S_i. At last, α leaves S_i and travels to S_{i+1}.

On the other hand, the *witness agent* is more passive than the *actual agent* in this protocol. It will not send any messages to the *actual agent.* Instead, it only listens to the messages coming from the *actual agent.* We assume that the *witness agent*, ω_{i-1}, arrives at S_{i-1}. Before ω_{i-1} can advance further in the network, it waits for the messages sent from α. When ω_{i-1} is in S_{i-1}, it expects receiving two messages: one is msg^i_{arrive} and another one is msg^i_{leave}. If the messages are out-of-order, msg^i_{leave} will be kept in the permanent storage of S_{i-1}. That is, msg^i_{leave} is considered as unattended, and becomes an indirect message until ω_{i-1} receives msg^i_{arrive}. When ω_{i-1} has received both msg^i_{arrive} and msg^i_{leave}, it *spawns* a new *witness agent* called ω_i. The reason of spawning a new agent instead of letting ω_{i-1} migrate to S_i is that originally ω_{i-1} is witnessing the availability of α. If a server failure happens just before ω_{i-1} migrates to S_i, then no one can guarantee the availability of the *actual agent.* More details about this problem will be discussed in Section 3.3. Note that the new *witness agent* knows where to go, i.e. S_i, because msg^i_{arrive} or msg^i_{leave} contains information about the location of S_i where α has just visited.

Figure 2 shows the flow of the protocol. The *actual agent* α arrives at S_i and the *witness agent* ω_{i-1} also arrives at S_{i-1}. First, α logs the message log^i_{arrive} in S_i [Step (1)]. Then, α sends the message msg^i_{arrive} to ω_{i-1} [Step (2)]. α then performs the computation. After α has finished all the tasks, it checkpoints its data in S_i [Step (3)]. We assume that the checkpointing action is one of the computations of the *actual agent.* That is, if the checkpointing action fails, the *actual agent* will abort the whole transaction. This is an

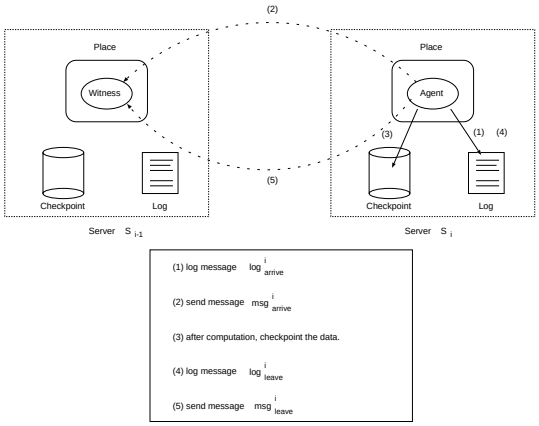

Figure 2. Steps in the witness protocol.

important step since this property guarantees that the checkpointed data will be available if the *actual agent* has already finished computing. Also, it is important for the recovery of the lost *actual agent.* Then, α logs the message msg^i_{leave} in S_i [Step (4)]. Before α leaves S_i, it sends the message msg^i_{leave} to ω_{i-1} [Step (5)]. Finally, α leaves S_i and travels to S_{i+1}. Upon receiving msg^i_{leave}, ω_{i-1} spawns ω_i, and ω_i travels to S_i. The procedure goes on until α reaches the last destination in its itinerary.

3.3 Failure and Recovery Scenarios

The purpose of the logs and the messages is to guarantee the *actual agent* has finished up to a certain point of the execution of the actual agent. If a server failure occurs in between a log and a message, we can determine when and where the *actual agent* fails. We assume that there will be no hardware failures such that the log message cannot be recorded in a the permanent storage. However, other kinds of failures like the software faults in the mobile agents or in the mobile agent platforms can happen.

In following subsections, we will cover different kinds of failures including the loss of the *actual agents*, and the loss of the *witness agents.* We describe several scenarios as follows.

3.3.1 ω_{i-1} fails to receive msg^i_{arrive}

The reasons that ω_{i-1} fails to receive msg^i_{arrive} can be:

1. The message is lost due to an unreliable network;
2. The message arrives after the timeout period of ω_{i-1};
3. α is dead when it is ready to leave S_{i-1};
4. α is dead when it has just arrived at S_i without logging; or
5. α is dead when it has just arrived at S_i with logging.

For the first two reasons, i.e., the *actual agent* does not die, and the message logged in S_i, log^i_{arrive}, can help solving this problem, as log^i_{arrive} is a proof for the existence

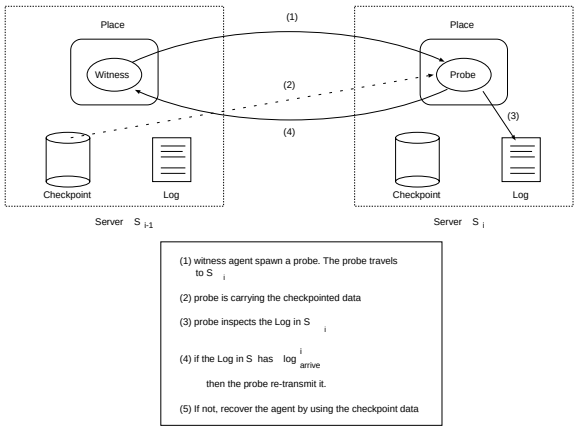

Figure 3. ω_{i-1} **fails to receive** msg^i_{arrive}

of α inside S_i. The *witness agent* can send out a *probe* ρ_i, another agent, to search for log^i_{arrive} in S_i. If found, ρ_i can re-transmit msg^i_{arrive} in order to recover the lost messages.

If ω_{i-1} fails to receive msg^i_{arrive} because of the loss of the *actual agent*, we may have the problem of *missing detection* when, in the fifth case, the *probe* will wrongly determine that the *actual agent* is still alive. This cases will be discussed in the next subsection.

If the failure is caused by the third or the forth cases, the probe will not be able to find log^i_{arrive} in S_i. Then, we can use the *checkpointed data* stored in S_{i-1} to recover the lost *actual agent*. Therefore, the probe is required to carry along the checkpointed data when it travels to S_i.

Figure 3 shows the execution steps of the probe ρ_i to detect agent failures when the witness fails to receive log^i_{arrive}. ω_{i-1} waits for the message, msg^i_{arrive}, for a timeout period. If the timeout period is reached, it creates the probe ρ_i. ρ_i then travels to S_i [Step (1)]. Since it may be required to recover a lost agent, it travels with the *checkpointed data* [Step (2)]. Upon arriving at S_i, it searches the log file in S_i for the message log^i_{arrive} [Step (3)]. If log^i_{arrive} is found, it re-transmits msg^i_{arrive} in order to recover the lost message [Step (4)]. However, *missing detection* may happen at this step. If the log message is not found, ρ_i will recover α in S_i by using the checkpointed data [Step (5)]. At last, ρ_i re-transmits the message msg^i_{arrive}. Note that we recover the lost *actual agent* in S_i instead of S_{i-1} because when ρ_i detects that a recovery is required, we can immediately recover that *actual agent* in S_i. If we perform the recovery in S_{i-1}, ρ_i has to send a message to S_{i-1} in order to inform ω_{i-1} that a recovery is required. There is a risk of losing such message.

In the meanwhile, ω_{i-1} waits for another timeout period. This is important since the message that is re-transmitted from S_{i-1} may be lost again. Or, another failure may strike S_i. Such a failure may terminate both the probe ρ_i and the just-recovered actual agent. Therefore, ω_{i-1} should wait until the message msg^i_{arrive} arrives.

Note that it is possible that ρ_i reaches S_i while α is still on the way. However, the occurrence probability of this case should be low. Since both α and ρ_i have to travel from S_{i-1} to S_i in the same network, they suffer from more or less the same network latency. Although there may be many routes from S_{i-1} to S_i, we can set the timeout of ω_{i-1} to be large enough to overcome the difference of speeds among these routes.

3.3.2 ω_{i-1} fails to receive msg^i_{leave}

The reasons that ω_{i-1} fails to receive msg^i_{leave} can be:

1. The message is lost due to an unreliable network;
2. The message arrives after the timeout period of ω_{i-1};
3. α is dead when it has just sent the message msg^i_{arrive}; or
4. α is dead when it has just logged the message log^i_{leave}.

As it is mentioned in the previous subsection, the fifth case of the previous subsection will be investigated here. This case will result in *missing detection* and the probe will re-transmit the expected message, msg^i_{arrive}, again regardless of the availability of the *actual agent*. Thus, we can expect that the *witness agent* is not able to receive msg^i_{leave}. Therefore, the last case of the previous subsection can be categorized as the third case of this subsection.

If the failure happens because of the first two reasons, it can be solved by the similar way as the previous subsection. ω_{i-1} can send a probe, again ρ_i, to search for log^i_{leave} in the log file of S_i. However, we may also have the problem of *missing detection* if the reason of the failures is the fourth or the fifth cases. That is, the actual agent is dead but we have not detected it. These two cases can be covered. When ρ_i re-transmits msg^i_{leave}, ω_{i-1} assumes that α has *successfully left* S_i. Therefore, ω_{i-1} spawns ω_i, and, eventually, ω_i travels to S_i. However, ω_i will never receive msg^{i+1}_{arrive} from α since α is already dead and does not exist in S_{i+1}. Consequently, we can successfully detect the agent failure by the third case of the previous subsection.

For the third case, we can handle it by detecting if log^i_{leave} exists. Since log^i_{leave} is absent, this implies that the actual agent is lost while it is performing its computation. In this case, since the actual agent is lost, the partially completed task by the actual agent should be undone. Therefore, it is required to *rollback* those operations by the method proposed in [8] in order to preserve the data consistency in S_i. We treat the whole computation process as a single transaction. Since the transaction is not committed, we have to abort all the uncommitted actions. We can use the log in S_i to recover the data inside S_i. The rollback recovery is not done by the probe, ρ_i. Instead, it is performed during the recovery of the server. Therefore, when the probe cannot find the log message log^i_{leave}, it can immediately use the checkpointed data to recover the *actual agent*. After the recovery is completed, the recovered *actual agent* can start performing its computation in S_i.

The execution steps of the probe when log^i_{leave} is missing is very similar to the steps in Figure 3. Note the recovery

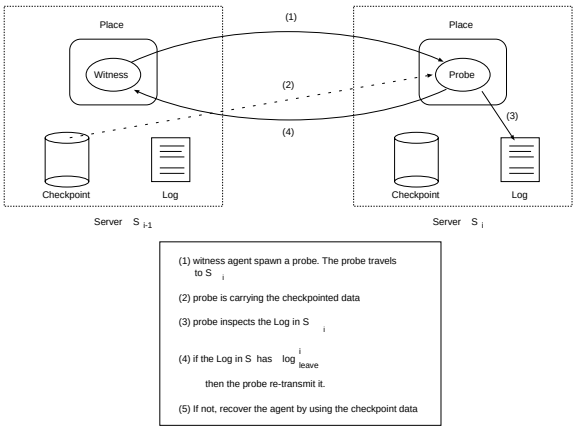

Figure 4. ω_{i-1} fails to receive msg^i_{arrive}

of the actual agent takes place on the server where the *actual agent* is expected to be hosted, i.e., in S_i. Moreover, when the actual agent is recovered, it immediately performs the computation in S_i regardless of the state before the failure occurs. This simplifies the implementation of the agent failure detection mechanism.

3.3.3 Failures of witness agent and recovery scenarios

Before the *actual agent* completes its itinerary, there are *witness agents* spawned along the itinerary of *actual agent*. The youngest (i.e., the most recently created) *witness agent* is witnessing the *actual agent*. On the other hand, the elder *witness agents* are neither idle nor terminated; they have another important responsibility: an earlier *witness agent* monitors the *witness agent* that is just one server closer to the actual agent in its itinerary. That is :

$$\omega_0 \rightarrow \omega_1 \rightarrow \omega_2 \rightarrow \ldots \rightarrow \omega_i \rightarrow \alpha$$

where "$\rightarrow$" represents the monitoring relation.

We name the above dependency the *witnessing dependency*. This dependency cannot be broken. For instance, if α is in S_i. ω_{i-1} is monitoring α, and ω_{i-2} is monitoring ω_{i-1}. Assuming we have the following failure sequence : S_{i-1} crushes and then S_i crushes. Since S_{i-1} crashes, ω_{i-1} is lost, hence no one monitoring α. If no one recovers ω_{i-1} in S_{i-1}, then no one can recover α after S_i has crushed. This is not desirable. Therefore, we need a mechanism to monitor and to recover the failed *witness agents*. This is achieved by the preserving the *witnessing dependency*: the recovery of ω_{i-1} can be performed by ω_{i-2}, so that α can be recovered by ω_{i-1}. Figure 5 illustrates this scenario.

Note that there are other more complex scenarios, but as long as the witnessing dependency is preserved, agent failure detection and recovery can always be achieved. In order to preserve the *witnessing dependency*, the *witness agents* that are not monitoring the *actual agent* receive message from the *witness agent* that is monitoring it. That is, ω_i sends a periodic message to ω_{i-1} in order to let ω_{i-1} knows that ω_i is alive. We label this message msg^i_{alive}. When ω_{i-1} cannot receive msg^i_{alive} from ω_i, the reasons

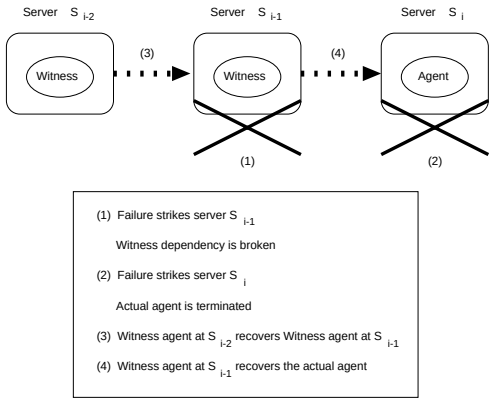

Figure 5. Witness agent failure scenario

can be:

1. The network is congested or unreliable;
2. The system load of S_i is high; or
3. ω_i is dead.

No matter what the reason of the failure is, ω_{i-1} can always assume that ω_i is dead. ω_{i-1} will spawn a new witness agent, namely ω_i, in order to replace the lost witness agent in S_i. Since there is no special data stored in the witness agent, only initializing the *states* of the new witness agent is required (see Figure **??**). When ω_i arrives at S_i, it re-transmits the message msg^i_{alive} to ω_{i-1}. If it is a false-detection, i.e., the message is lost, but the witness agent is still in S_i, we can prohibit multiple instances of ω_i from executing by exchanging messages between 2 instances of ω_i.

3.3.4 Catastrophic failures

The witness agent protocol cannot guarantee that all failures are detected and recovered. First of all, the *witnessing dependency* cannot be always preserved. The weakness is at the head of the *witness dependency*, ω_0, which is not monitored by any agents. Hence, when S_0 fails, ω_0 cannot be recovered. This will shorten the *witness dependency*.

Secondly, if the above *shortening* process goes on, the whole *witnessing dependency* will *collapse* when a series of failures completely destroy the *witnessing dependency*. Though the possibility of such failure series is extremely small, if it happens, the protocol will fail.

In order to handle this failure series, the owner of the *actual agent* can send a *witness agent* to the first server, S_0, in the itinerary of the agent with a timeout mechanism. The effect of sending this *witness agent* is similar to the case when a *witness agent*, ω_i, fails to receive msg^{i+1}_{alive}. This method can recover ω_0 and the *witness dependency* effectively with an appropriate timeout period.

3.3.5 Simplification

Note the *witnessing dependency* is useful only when several servers fail in a short period of time. However, this dependency uses a lot of resources along the itinerary of the *actual agent*. If we assume that *no two or more servers can fail at the same period of time*, we can simplify our mechanism by shortening the *witnessing dependency*. The dependency then becomes:

$$\omega_{i-1} \rightarrow \omega_i \rightarrow \alpha$$

where "$\rightarrow$" represents the monitoring relation.

Since no two servers can fail simultaneously, two *witness agents* are sufficient to guarantee the availability of the *actual agent*. When a failure occurs in S_i, ω_{i-1} can recover ω_i after the server is recovered. When a failure happens in S_{i-1}, we can let the dependency to be further shortened. It is because when α travels to S_{i+2}, a new dependency involving ω_i, ω_{i+1}, and α will be formed, and the simplified protocol resumes. Finally, when ω_i spawns ω_{i+1}, we can terminate ω_{i-1} by sending a message from S_i to S_{i-1}.

4 RELIABILITY EVALUATION

The reliability evaluation of our protocol is conducted by Stochastic Petri Net simulation [11, 12] using SPNP [13] as well as agent code implementation by using Concordia [3]. Reliability in our experiment is measured by the successful ratio of actual agents in completing their scheduled round-trip travels in a network of agent servers. We introduce a server called *home*, i.e., the machine of the agent owner. The *home* server is responsible for transmitting agents when the agents start traveling as well as for receiving agents when they finish traveling on the network. We carry out the experiment by using different itineraries with various lengths. We assume that the *home* is *error-free* while the other servers are *error-prone*. We inject failures into every server. In each server, we create a daemon running together with the agent server (or the agent platform). The daemon will randomly kill the process of the agent server. When the *server monitor*, another daemon that monitors all the servers, discovers that an agent server is dead, it restarts the agent server process within a specified time.

4.1 Server Failure Detection Analysis

Figure 6 shows the Stochastic Petri Net that models the server failure detection mechanism for one server. The shaded part on the left describes the *states of an agent* inside a server. The transitions on that part are mainly *timed transitions*. They model the time spent on traveling between two servers and the time required for the computation of an agent. The shaded region on the right is the *server monitor*. It also contains timed transitions. These transitions model the time spent on detecting the availability of a server and

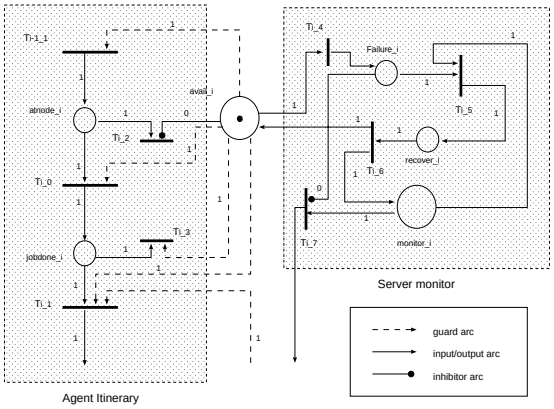

Figure 6. A server model with server failure detection

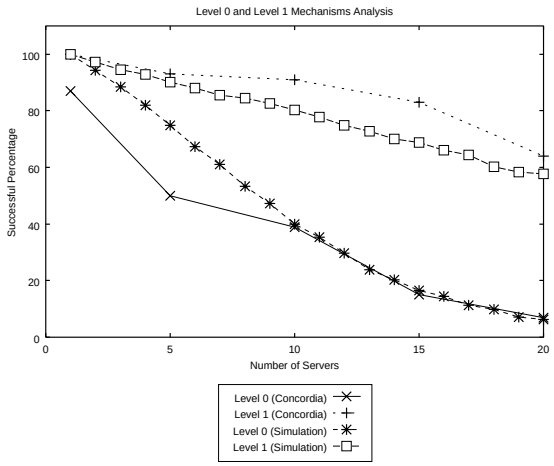

Figure 7. Evaluation result of server failure detection

the time required to perform a recovery. The non-shaded *place* in the middle states the *availability of the server*. When a *token* is inside that place, the server is available. However, if there is no token inside that place, the server fails, and all agents inside the server are lost. Figure 6 only shows the model of one server. We can put several servers together to form a chain. That chain represents the itinerary of the agent. Our experiment is carried out by connecting different numbers of these modules to represent different numbers of servers in the agent itinerary.

The results of using both the *Concordia* implementation and the *SPNP* simulation are shown in Figure 7. The experiment compares two scenarios, one without server failure detection (Level 0) and the other with server failure detection (Level 1). This experiment illustrates how much the reliability is improved by the server detection and recovery mechanism with a given server failure rate. The result shows that the successful percentage of an agent with server failure detection and recovery drops much slower than an agent without this implementation. With the measurement of 20 servers in the agent itinerary, the successful ratio of the agents with fault-tolerance server implementation falls

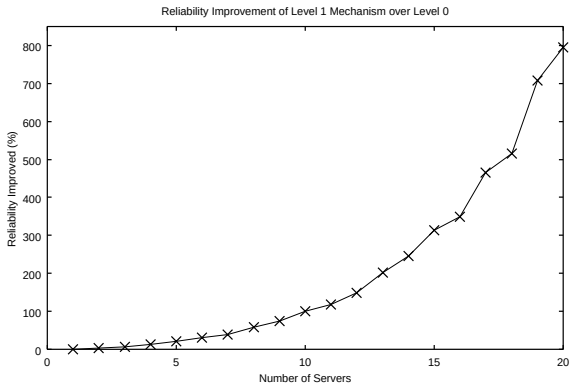

Figure 8. Reliability improvement with server failure detection

between 55 and 60 percents. The successful percentage of agent without fault-tolerance server implementation, on the other hand, falls below 10 percent for both simulation and Concordia implementation. Figure 8 shows the overall improvement of the fault-tolerance server implementation (Level 1) versus the non-fault-tolerance implementation (Level 0). The increasing slope implies that the advantage of the fault-tolerance implementation becomes more significant as the number of servers increases.

The result measured by using simulation shows a monotonic increasing relation between the successful ratio and the number of servers. As the number of servers increases, the number of successful round-trip-travels decreases progressively. It is reasonable since the chance of waiting for the recovery of a failed server increases, the probability of the agent loss while it is waiting will also increase.

4.2 Agent Failure Detection Analysis

We perform the same experiment for the evaluation of the agent failure detection and recovery. In the previous subsection, we can observe that with the server failure detection and recovery, the system still suffers from the loss of agents. Therefore, the goal of the agent failure detection and recovery mechanism is to increase the percentage of successful round-trip travels by witness agent mechanism.

We construct the Stochastic Petri Net that models both the failure detection and recovery for both servers and agents. Our experiment is carried out by simulation with up to 20 servers, which is shown in Figure 9. The primary result indicates that the successful percentage of a round-trip travel in our fault-tolerance mechanism (Level 2, which further incorporates the witness agents) is further improved with respect to that with only the fault-tolerance server implementation (Level 1). Our fault-tolerance mechanism can always recover failed agents, i.e., we have a 100% recovery. Figure 10, depicts the reliability improvement of the agent/server fault-tolerance mechanism (Level 2) over the server-only fault-tolerance mechanism (Level 1). The result shows that the reliability is further enhanced. It reaches about 80% with an itinerary of twenty servers. However,

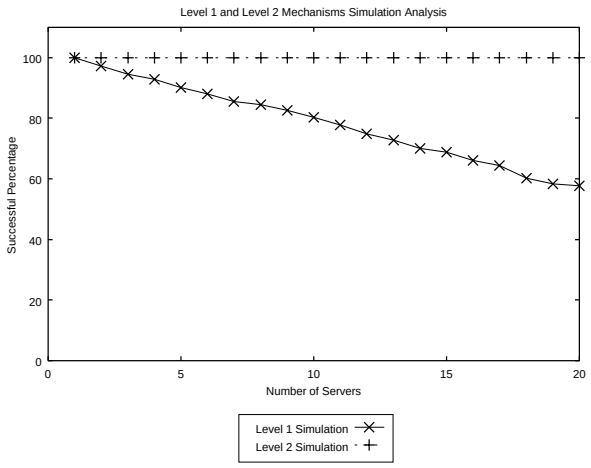

Figure 9. Server vs. agent failure detection simulation result.

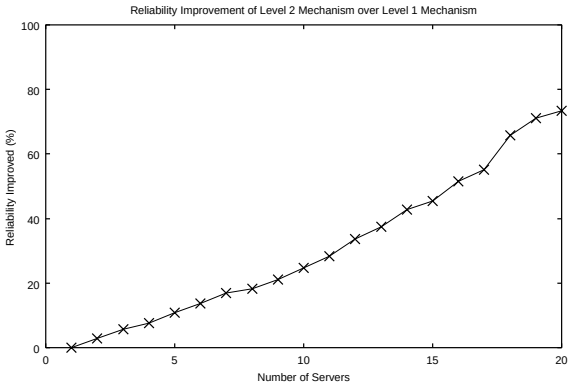

Figure 10. Reliability improvement with agent failure detection and recovery

one side effect is that whenever we have recovered an agent, the new agent may encounter another failure. This generates many extra agents. Figure 11 shows the results of the number of extra agents (in percentage) per successful round-trip travel against the number of servers. It indicates that as the itinerary becomes longer, more extra agents will be required, and consequently the complexity of the system is increased.

5 CONCLUSION

In this paper, we propose an approach to enhance mobile agent systems with better reliability. We also analyze different failure scenarios that may happen in the mobile agent systems, and design a progressive fault-tolerance scheme that can detect the server and the agent failures. We further discuss the mechanism, which uses a global daemon, communication messages, and checkpointing techniques, that enables us to detect and recover these failures by employing cooperative witness agents. We conduct reliability evaluation of the proposed mechanism for server failures and agent failures. The result shows that, under the condition

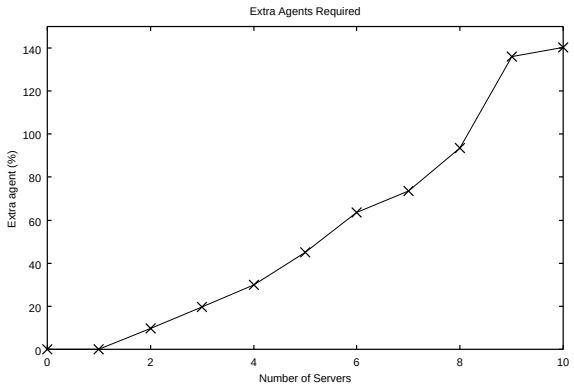

Figure 11. Extra agent per successful round-trip travel.

for up to 20 servers, with the server failure detection only, we achieve a significant improvement for the successful ratio of the agent round-trip travels. In addition to the server failure detection, we further improve the reliability by using the agent failure detection over server failure detection. However, the cost becomes higher when we want to achieve a higher level of fault-tolerance. Quantitative results for trade-off study between cost and reliability of the proposed scheme are provided in this paper.

6 ACKNOWLEDGMENT

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No. CUHK4193/00E).

References

[1] Anthony H.W. Chan, T.Y. Wong, Caris K.M. Wong, and Michael R. Lyu. Design, Implementation and Experimentation on Mobile Agent Security for Electronic Commerce Applications. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.* pp.1871-1878.

[2] D. Lange and M. Oshima. Mobile agents with java: the aglet API. *Special Issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents* 1(3) (Baltzer Science Publishers, 1998), pp.111-121.

[3] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, Concordia: an infrastructure for collaborating mobile agents, in *Mobile Agents, Proceedings of 1st International Workshop, MA'97, Lecture Notes in Computer Science 1219*, pp.86-97.

[4] J. Baumann, F. Hohl, K. Rothermel and M. Strasser. Mole - concepts of a mobile agent system. *Special Issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents* 1(3) (Baltzer Science Publishers, 1998), pp.123-127.

[5] Stefan Pleish and Andre Schiper. Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems. *Proceedings of the 19th IEEE Symposium on Reliable Distributed System 2000* (SRDS-2000), pp.11-20

[6] Stefan Pleisch and Andre Schiper. FATOMAS - A Fault Tolerant Mobile Agent System Based on the Agent-Dependent Approach. *The International Conference on Dependable Systems and Networks, 2001*, pp.215-224.

[7] Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen, and Dmitril Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.*, pp. 180-189

[8] Markus Strasser and Kurt Pothernel. System Mechanisms for Partial Rollback of Mobile Agent Execution. *Proceedings of 20th International Conference on Distributed Computing Systems 2000*, pp.20-28.

[9] Victor F. Nicola. Checkpointing and the Modeling of Program Execution Time. *Software Fault Tolerance, M. Lyu (ed.), John Wiley & Sons, 1994*, pp.167-188.

[10] Kurt Rothermel and Markus Stasser. A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. *Proceedings of 17th IEEE Symposium on Reliable Distributed Systems 1998*, pp.100-108.

[11] Lorrie Tomek and K.S. Trivedi. Analyses Using Stochastic Reward Nets. *Software Fault Tolerance, M. Lyu (ed.), John Wiley & Sons, 1994*, pp.231-248.

[12] Dianxiang Xu and Yi Deng. Modeling Mobile Agent Systems with High Level Petri Nets. *IEEE Systems, Man, and Cybernetics, 2000*, pp.3177-3182.

[13] C. Hirel, B. Tuffin, and K.S. Trivedi. SPNP: Stochastic Petri Nets, Version 6.0. *11th International Conference of Computer performance evaluation: Modeling tools and techniques. Schaumburg, Il., USA, B. Haverkort, H. Bohnenkamp, C. Smith(eds.), Lecture Notes in Computer Science 1786, Springer Verlag, 2000.*