Design and Evaluation of A Fault Tolerant Mobile Agent System

Michael R. Lyu, Xinyu Chen and Tsz Yeung Wong

Abstract

Improving the survivability of mobile agents in the presence of agent server failures with unreliable underlying networks is a challenging issue. In this paper, we address a fault tolerance approach of deploying cooperating agents to detect server and agent failures as well as to recover services in mobile agent systems. Three types of agents are involved, which are the *actual agent*, the *witness agent* and the *probe*. We introduce a failure detection and recovery protocol by employing a message-passing mechanism among these three kinds of agents. Different failure scenarios and their corresponding recovery procedures are discussed. Further, Stochastic Petri Net models for the proposed approach are developed and survivability evaluations through simulation are conducted.

Index Terms

Mobile agent, Witness agent, Probe, Witnessing dependency, Fault tolerance, Stochastic Petri Net

Design and Evaluation of A Fault Tolerant Mobile Agent System

I. Introduction

Mobile agents are autonomous software objects capable of actively migrating from one server to another in a computer network and executing on behalf of a network user. When an agent travels to another server, its code, data as well as execution state are captured and transferred to the next server. Because a mobile agent moves to resource-containing servers to access them locally, it is not required to transfer multiple requests and responses across congested network links, thus the overall performance becomes more efficiently. Consequently, mobile agents create a new paradigm for data exchange and resource sharing in rapidly growing and continuously changing computer networks. It has being exploited in electronic commerce, information retrieval, network and workflow management, etc. Many academic and commercial systems provide mobile agent execution environments, such as Aglets, Agent TCL, Concordia, Grasshopper, Mole, Odyssey and Voyager.

In a distributed system any software or hardware components may be subject to failures. A mobile agent is lost when its hosting agent server crashes during its execution, or it may be dropped in congested networks. Therefore, survivability as well as fault tolerance are vital issues for the deployment of mobile agent systems. A number of research work has been done in these areas. Pleisch *et al.* adopt the utilization of *replication* as well as *masking* [1]. The proposed idea employs replicated servers to mask failures. Dalmeijer *et al.* [2] utilize a checkpoint manager to monitor all agents, which is responsible to keep track of all the agents and to restart the lost agents. Osman *et al.* [3] analyze the execution model of agent platforms to develop a pragmatic framework for agent systems fault tolerance, which deploys a communication-pair-independent checkpointing strategy. Pears *et al.* [4] utilize two exception handling approaches to maintain the availability of mobile agents, which operate at different servers. Silva *et al.* [5] present a set of fault tolerance techniques, such as fault detection, checkpointing and restart, software rejuvenation, and reconfigurable itinerary. The authors also discuss the issues of network partitions.

Our approach [6] is rooted from the approach suggested in [7]. We distinguish three types of agents. One type is the common mobile agent which performs the required computations for its owners. We name it the *actual agent*. The second type of agent monitors the actual agent and detects whether it is lost. We call this type of agent the *witness agent*. The last type is the *probe* who is responsible to recover the failed actual agent and witness agents. A peer-to-peer message passing mechanism stands between the actual agent and the witness agents to perform failure detection and recovery through time-bounded information exchange. In addition to the introductions of the witness agent, the probe and the messages passing mechanism, we need to *log* the actions performed by the actual agent. Because when failures occur, we need to abort uncommitted actions when we perform *rollback recovery* [8]. Moreover, we employ *checkpointed data* [2] to recover the lost actual agent.

II. System Architecture and Protocol Design

Different server failure detection and recovery strategies have been exploited in the literature, which could bring the failed server back to work; however, it cannot recover the lost agent if the actual agent resides on the failed server when the failure occurs. Therefore, we need a more advanced approach to re-initialize the lost agent.

Fig. 1 shows the overall design of the agent server architecture which is capable of recover a lost agent. The agent server should provide three types of stable storage for logs, checkpoints and messages, respectively. First, every server logs the actions performed by an agent. The logged information is vital for failure detection as well as recovery. Also, the hosting servers log which objects have been updated. When a server failure occurs, we should recover the lost agent due to the failure; however, an agent contains its internal data, which may be lost due to the failure. Moreover, if we allow the agent to renew its computation from the starting point of its itinerary, the *exactly-once* property will be violated. Therefore, we have to checkpoint the data of an agent, thus require a permanent storage to store the checkpointed data. Furthermore, our agent failure detection and recovery protocol is based on message passing as well as message logging. In order to detect and recover the failures of an actual agent, we designate another type of agent, namely the witness agent, to monitor whether the actual agent is alive or dead. When the actual agent completes its dedicated work on a server and starts to continue its journey to the next server, it spawns a witness agent at the current server. In addition to the witness agent, we design

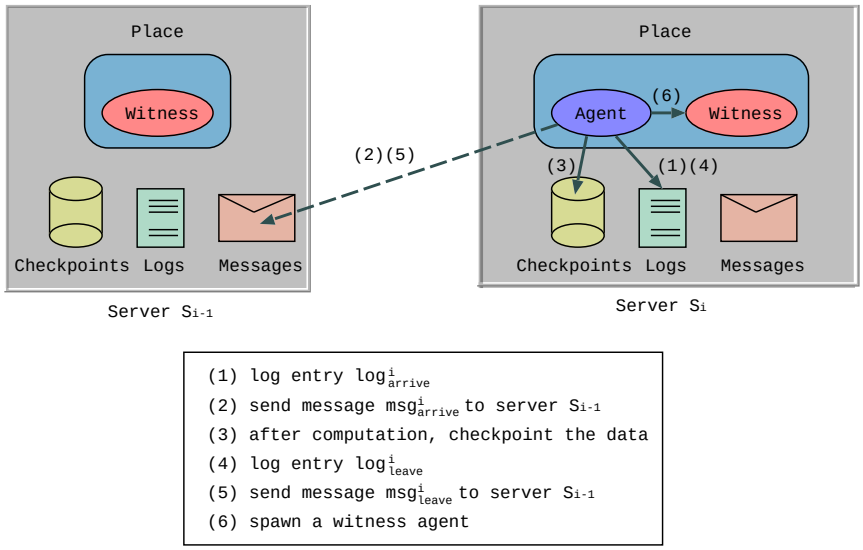

Fig. 1. Fault tolerant mobile agent server framework

a communication mechanism between agents and servers.

Assume that, currently, the actual agent has just arrived at server S_i while the witness agent has been spawned at server S_{i-1} before the actual agent leaves server S_{i-1} . We denote the actual agent as α and the witness agent as ω_{i-1}.

As the actual agent plays an active role in our proposed protocol, we discuss its activity first. Fig. 1 shows the action flow performed by the actual agent α at server S_i. After α has arrived at S_i, it immediately writes an arrival entry, log^i_{arrive}, into the logs on the permanent storage in S_i [Step (1)]. The purpose of this log entry is to provide an evidence 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 S_{i-1} [Step (2)]. S_{i-1} keeps the received message in its message box. Then, α performs its dedicated tasks at S_i. When it finishes, it immediately checkpoints its internal data [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 important step since this property guarantees that the checkpointed data will be available if the actual agent has already finished computing. Moreover, it is essential for the recovery of the lost actual agent. After that α logs another entry log^i_{leave} in S_i [Step (4)]. This log entry expresses that α has completed its computation and is ready to travel to the next server S_{i+1}. In

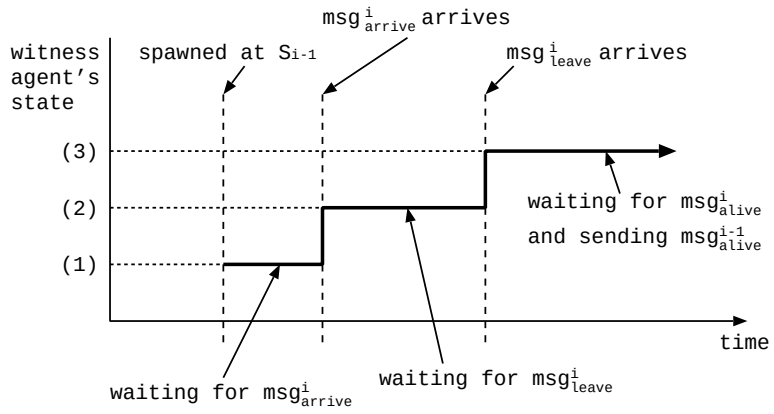

Fig. 2. Life scenario of witness agent ω_{i-1}

the next step, α sends ω_{i-1} another message, msg^i_{leave}, in order to inform ω_{i-1} that α is ready to leave S_i [Step (5)]. After sending the leave message, α spawns a new witness agent at the current server [Step (6)]. At last, α leaves S_i and travels to S_{i+1}. The procedure goes on until α reaches the last destination in its itinerary.

On the other hand, the witness agent ω_{i-1} is more passive than the actual agent in this protocol. It does not send any messages to the actual agent. Instead, it simply waits to receive messages from the local mailbox first. Two messages are expected: one is msg^i_{arrive} and the other is msg^i_{leave}. One advantage of receiving these two types of messages through a mailbox is that the mailbox provides a history record that these messages have arrived at this server. Additionally, the mailbox provides a mechanism to shuffle messages and only lets msg^i_{arrive} pass before msg^i_{leave}. Therefore, if the messages are out-of-order, msg^i_{leave} will be kept in the permanent storage and will not be consumed by ω_{i-1}. The message record in the mailbox will be utilized in recovering the lost witness agent and actual agent. After receiving these two indirect messages, ω_{i-1} waits for the direct heartbeat message, msg^i_{alive}, which is sent by the witness agent at server S_i. This message testifies the liveness of ω_i. Therefore, a witness agent will undergo three states after being spawned, shown in Fig. 2.

III. Agent Failure Detection and Recovery

The purpose of the introductions of the log entries, log^i_{arrive} and log^i_{leave}, and the messages, msg^i_{arrive} and msg^i_{leave}, is to guarantee that the actual agent has finished up to a certain point of

its execution. If a server failure occurs between a log entry and its corresponding message, we can determine when and where the actual agent fails. We assume that there will be no hardware failures such that the log entries cannot be recorded in the permanent storage. However, other kinds of failures like software faults in the mobile agents or in the mobile agent platforms may occur. In the following subsections, we will cover different types of failures including the loss of the actual agent and the loss of the witness agents.

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

The cases that the witness agent at server S_{i-1}, ω_{i-1}, fails to receive msg^i_{arrive} include:

(a) The message is lost due to an unreliable network;

(b) The message arrives after the timeout period of ω_{i-1};

(c) α is dead when it is ready to leave S_{i-1} and heading for S_i;

(d) α is dead when it has just arrived at S_i without logging;

(e) α is dead when it has just arrived at S_i with logging.

By utilizing the arrival entry logged in S_i, log^i_{arrive}, we can solve the first two problems. In these two cases, the actual agent does not die and log^i_{arrive} is a proof for the existence of α inside S_i. The witness agent can then send out a *probe* ρ_i, another agent, to search for log^i_{arrive} in S_i. If found, ρ_i re-transmits msg^i_{arrive} in order to recover the lost or delayed message.

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 case (e), the probe can find log^i_{arrive} and wrongly determines that the actual agent is still alive, and thus terminates itself prematurely. This case will be discussed in the next subsection.

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

Fig. 3 shows the execution steps to detect agent failures when the witness agent fails to receive msg^i_{arrive}. ω_{i-1} waits for the message msg^i_{arrive} with a configurable 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 entry log^i_{arrive} [Step (3)]. If log^i_{arrive} is found, it re-transmits msg^i_{arrive} [Step (4)]. If the log entry is not found, ρ_i will recover α in S_i by using

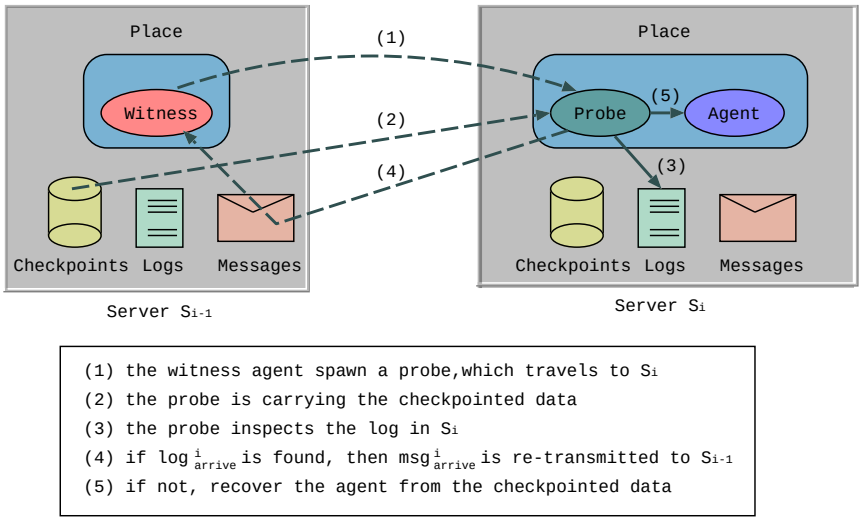

Fig. 3. Recovery steps when ω_{i-1} fails to receive msg^i_{arrive}

the piggyback checkpointed data [Step (5)]. Finally, the recovered actual agent at S_i will send 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 an agent recovery is required. This introduces a risk of losing the critical message.

When ω_{i-1} sends out ρ_i, it waits for another timeout period. This is important since probe ρ_i may lose, the message that is re-transmitted from S_i may be lost again, or another successive 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.

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

The reasons that ω_{i-1} fails to receive msg^i_{leave} are listed as follows:

(a) The message is lost due to an unreliable network;

(b) The message arrives after the timeout period of ω_{i-1};

(c) α is dead when it has just sent the message msg^i_{arrive};

(d) α is dead when it has just logged the entry log^i_{leave};

(e) α is dead when it has spawned the witness agent ω_i.

If the failure occurs because of the first two reasons, it can be solved in a similar way as that in the previous subsection. ω_{i-1} will send a probe, again denoted as ρ_i, with a different task to search for log^i_{leave} in the log file of S_i. We may also face the missing detection problem if the reason of the failures is cases (d) or (e). The solution to case (d) will be discussed in the subsection III-C by utilizing the witness agent monitoring mechanism. For case (e), it becomes the same case as ω_i cannot receive msg^{i+1}_{arrive}, which has been discussed in the previous subsection.

For case (c), we handle it by detecting whether log^i_{leave} exists or not. If log^i_{leave} is absent, it implies that the actual agent is lost while performing its computation. Case (e) of the previous subsection can be categorized as this case. Because we could expect that the witness agent ω_{i-1} will not receive msg^i_{leave} after the loss of α. In this case, since the actual agent is lost, its partially-completed task 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 actions which have been executed in this transaction. We employ 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 entry log^i_{leave}, it can immediately use the checkpointed data to recover the actual agent. After the recovery is completed, the recovered actual agent continues to perform its computation in S_i. This simplifies the implementation of the agent failure detection mechanism. The execution steps of the probe when msg^i_{leave} is missing is very similar to the steps in Fig. 3. Note again the recovery of the actual agent takes place in the server where the actual agent is expected to be hosted, i.e., S_i.

C. Failures of witness agents and the recovery strategy

The witness agent at server S_i, ω_i, is spawned by the actual agent α after it logs the entry log^i_{leave} and before it moves to the next server S_{i+1}. The reason of engaging this witness agent

spawning strategy instead of letting the the lagged witness ω_{i-1} moves forward to server S_i is to reduce the network communication, thus minimizing the chances of agent loss introduced by link failures, and to create a chain of witness agents.

Before the actual agent completes its itinerary, there are witness agents spawned along the itinerary of the actual agent. The most recently created witness agent is monitoring the actual agent; on the other hand, the older witness agents are responsible for monitoring the witness agent that is just one server closer to the actual agent in its itinerary. That is

$$\omega_0 \rightarrow \omega_1 \rightarrow \omega_2 \rightarrow \ldots \rightarrow \omega_{i-1} \rightarrow \omega_i \rightarrow \alpha,$$

where "$\rightarrow$" represents the monitoring relation. 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 travelling as well as for receiving agents when they finish travelling on the network. This home server is denoted as S_0. Therefore, ω_0 denotes the witness agent resides at the home server. We name the above dependency the *witnessing dependency*. This dependency cannot be broken, otherwise, no witness agent will monitor the actual agent eventually. In order to preserve the witnessing dependency, the witness agents that are not monitoring the actual agent periodically receive heartbeat messages from its forward witness agent. That is, ω_i sends a periodic message, msg^i_{alive}, to ω_{i-1} in order to let ω_{i-1} know that ω_i is alive. When ω_{i-1} cannot receive msg^i_{alive} from ω_i, the reasons may be:

(a) The network is congested or unreliable;

(b) The system load of S_i is too high;

(c) ω_i is not created or is dead.

No matter what the reason of the failure is, ω_{i-1} can always assume that ω_i is dead. These cases have included case (d) in Subsection III-B. After timeout, ω_{i-1} will send a probe ρ_i to S_i to spawn a new witness agent in order to replace the lost witness agent in S_i. Since there is no special data stored in the witness agent except the agent itinerary, this type of probe does not need to carry the checkpointed data. Because there exists a probability of false detection due to cases (a) and (b), when ρ_i arrives at S_i, it first checks whether the witness agent is still alive. If no witness agent exists, it initializes a new witness agent. This newly-created witness agent will re-send the message msg^i_{alive} to ω_{i-1}; otherwise, ρ_i just disposes itself.

Fig. 4 illustrates a recovery procedure of witness agent failure. If α is in S_i, then ω_{i-1} is

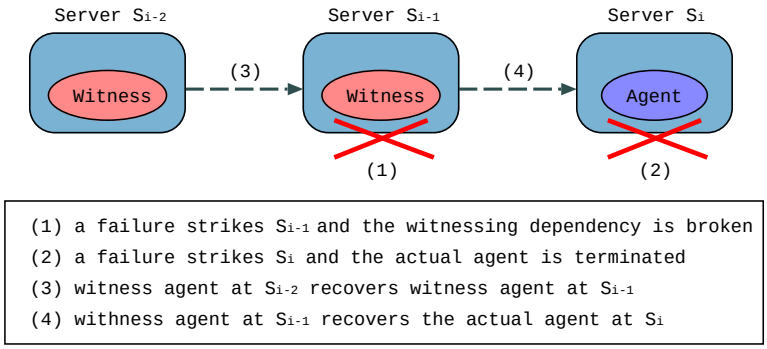

Fig. 4. Witness agent failure scenario

monitoring α, and ω_{i-2} is monitoring ω_{i-1}. Assuming we have the following failure sequence: S_{i-1} crashes first and then S_i crashes. Since S_{i-1} crashes, ω_{i-1} is lost, hence no one monitors α. If no one recovers ω_{i-1}, then no one can recover α after S_i crashes. This is not desirable. Therefore, we need a mechanism to monitor and recover the failed witness agents. This is achieved by preserving the witnessing dependency: the recovery of ω_{i-1} can be performed by ω_{i-2}, so that eventually α can be recovered by ω_{i-1}. Note that there are other more complex scenarios, but as long as the witnessing dependency is preserved, the agent failure detection and recovery can always be achieved.

D. Simplification of the witnessing dependency

To keep the witnessing dependency, witness agents are created in the itinerary and heartbeat messages are exchanged between witness agents. This procedure consumes a lot of resources along the itinerary of the actual agent. If, however, we assume that no k or more servers can fail at the same period of time, we can simplify our mechanism by shortening the witnessing dependency through keeping the witness length less than or equal to k. If the actual agent α is now at server S_i, the simplified dependency then becomes:

$$\begin{aligned} &\text{if } (i \leq k) && \omega_0 \rightarrow \omega_1 \rightarrow \ldots \rightarrow \omega_{i-1} \rightarrow \alpha \\ &\text{else} && \omega_{i-k} \rightarrow \omega_{i-k+1} \rightarrow \ldots \rightarrow \omega_{i-1} \rightarrow \alpha, \end{aligned}$$

where "$\rightarrow$" represents the monitoring relation. Since no more than k servers can fail simultaneously, k witness agents are sufficient to guarantee the availability of the actual agent. When a failure occurs in S_i, ω_{i-1} can recover α after the server is restarted. When a failure strikes S_j,

$i-k<j<i$, ω_{j-1} will recover ω_j. When a failure occurs in S_{i-k}, as ω_{i-k} cannot be recovered, the length of witnessing dependency is reduced by 1. However, when α travels to S_{i+1}, a new witness agent ω_i will be created and a new dependency involving ω_{i-1}, ω_i, and α will be formed, thus the witnessing dependency resumes its length of k. Finally, when α successfully logs the entry log^{i+1}_{arrive}, we can terminate ω_{i-k} by sending a message, msg^{i+1}_{kill}, from S_{i+1} to S_{i-k}.

IV. Stochastic Petri Net Models and Experimental Results

With the proposed agent failure detection and recovery mechanism, we evaluate its improvement on agent survivability through Stochastic Petri Net (SPN) and simulation [9]. We denote a mobile agent system without any fault tolerance as Level 0. For comparison, we introduce a server failure detection and recovery mechanism. Before the actual agent leaves the current server, it tests whether or not its next destination server is alive. If yes, it moves to it; otherwise, it will stay at the current server until the next server comes back to work. If a mobile agent system engages this server failure detection and recovery strategy, it is at Level 1. If it additionally embeds the agent failure detection and recovery, it goes up to Level 2. We define the metric, the *agent survivability*, as the successful ratio of actual agents in completing their scheduled round-trip journeys in a network of agent servers.

A. SPN models

Fig. 5 shows the SPN that models the mobile agent system at Level 2. SPNs for Levels 0 and 1 are subsets of the SPN for Level 2, so we omit them. The right dashed-line box manifests the state transitions of the actual agent at a server. Transitions t_a_m, t_l_a, t_a_p, t_l_l and t_w_n are *timed transitions*, which model the time spent on travelling between two servers, the time to log the arrival entry, the required computation time inside a server, the time to log the leave entry and the time to spawn a witness agent, respectively. The right un-boxed places and transitions are for the server itself. t_s_f models the time to a failure and t_s_r is the time required to perform a recovery. Here instant failure detection is assumed. A more realistic round-robin failure detection approach could also be modelled [6]. When a *token* is inside place p_s_u, the server is available; however, if there is no token inside that place, the server fails, and all agents inside that server are lost. We utilize inhibitor arcs and guard arcs to model these phenomena. The guard arc from p_s_u to t_a_m prevents the actual agent from moving to the server when it fails. The upper-left

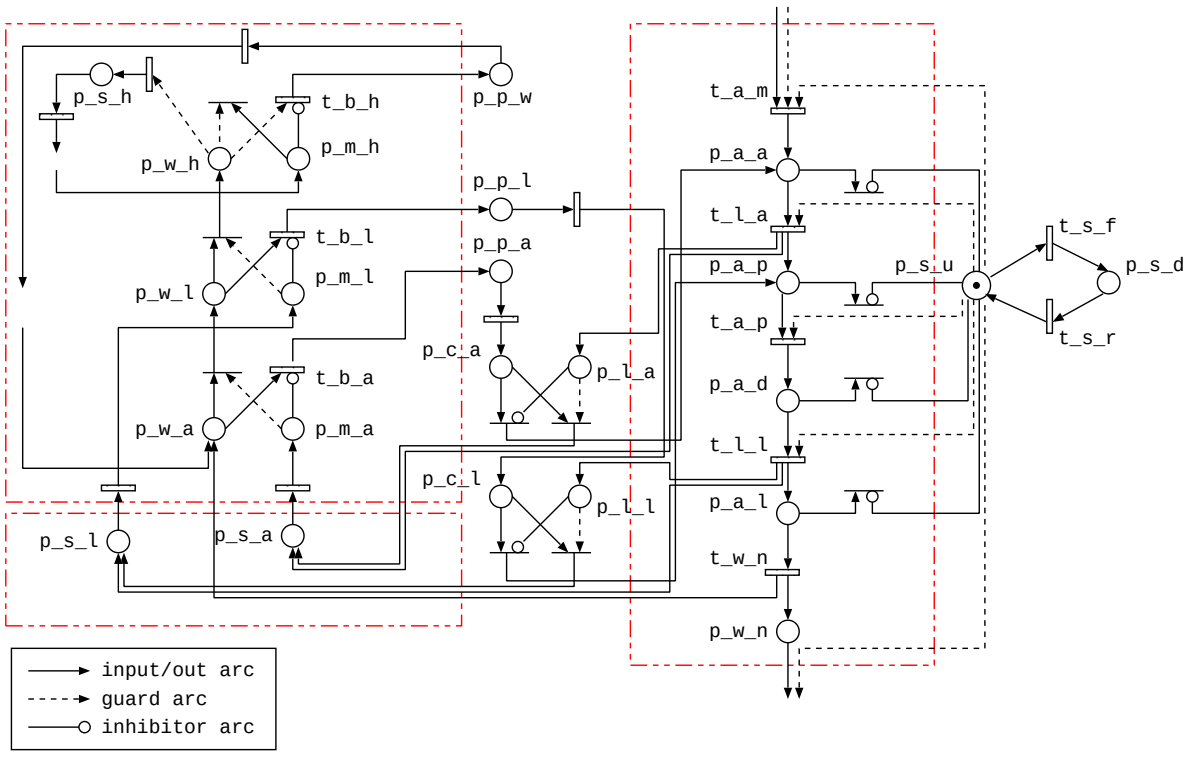

Fig. 5. SPN model for Level 2

dashed-line box shows the witness agent's state transitions. Three states, waiting for the arrival, leave and heartbeat messages, are clearly represented. The middle non-boxed area is for the probe. Three types of probes will be dispatched, which are for retrieving the arrival message, for retrieving the leave message, and for recovering a witness agent. The places in the lower-left dashed-line box represent sending the arrival and leave messages to a witness agent, which are shared by the actual agent and the probe. After a server recovers from a failure, places p_l_a, p_l_l, p_m_a and p_m_l will be initialized with a token if their corresponding logs and messages are present.

Fig. 5 only shows different agents' behaviors in one server. We can put several servers together to form a chain, which represents the itinerary of an actual agent and the witnessing dependency.

B. Experimental results

Our experiments are carried out by simulations developed with C-Sim [10]. Some parameters are given here: the network transmission rate for all agents is 100, and for messages is 200;

the server repair rate $t\_s\_r = 0.1$; all message log rates are 100; the arrival, leave and heartbeat messages bound times are 1, 100, and 20, respectively; and the heartbeat interval is 5. We carry out the experiments by using different itineraries with various number of servers. These experiments illustrate how the agent survivability is improved.

The results of using the C-Sim implementation with different server failure rates and job completion rates are shown in Fig. 6. For each parameter pair, we conduct six simulations, one for Level 0, one for Level 1, and the other four for Level 2 with different k, the length of witness agents. All sub-figures (a) show that the agent survivability decreases progressively as the number of servers increases. It is reasonable since the chance of waiting for the recovery of a failed server increases, the agent loss probability will also go up while the agent is waiting. The agent failure detection and recovery mechanism achieves relatively higher survivability than other two levels. The improvement becomes more significant as the number of servers and the length of witness agents increase. Therefore, to achieve high percentage of successful round-trip agent travels with more servers, we should increase the number of witness agent correspondingly.

An unexpected result is that after engaging the server failure detection and recovery approach, the completed agents percentage is, however, less than that of without engaging any fault tolerance mechanisms. We know that in both these levels, when the actual agent is lost, no way is provided to recover it. Therefore, if the agent finishes its journey more quickly, its loss probability is less. After engaging the server failure detection and recovery, the actual agent spends more time in the system because it should wait at its current server when its next server is unavailable. Consequently, its loss chance becomes higher. Even the agent failure detection and recovery with small number of witness agent is employed, this statement is also true. By comparing Fig. 6(I-a) and (II-a), we note that the higher the failure rate is, the higher the agent loss probability is. However, if an agent could complete its dedicated work at each server more quickly (Fig. 6(III-a)), the survivability will increase. This implies that under unreliable systems, the actual agent should complete its task as fast as it can.

We know that Level 2 is achieved by engaging witness agents and probes. All sub-figures (b) and (c) show what the cost it has to pay. All sub-figures (b) show the number of created witness agent with Level 2, which increases linearly with the number of servers. The higher the number of required witness agents k is, the more witness agents will be created. Sub-figures (c) show the number of probes generated during agent execution. Note that the higher percentage of

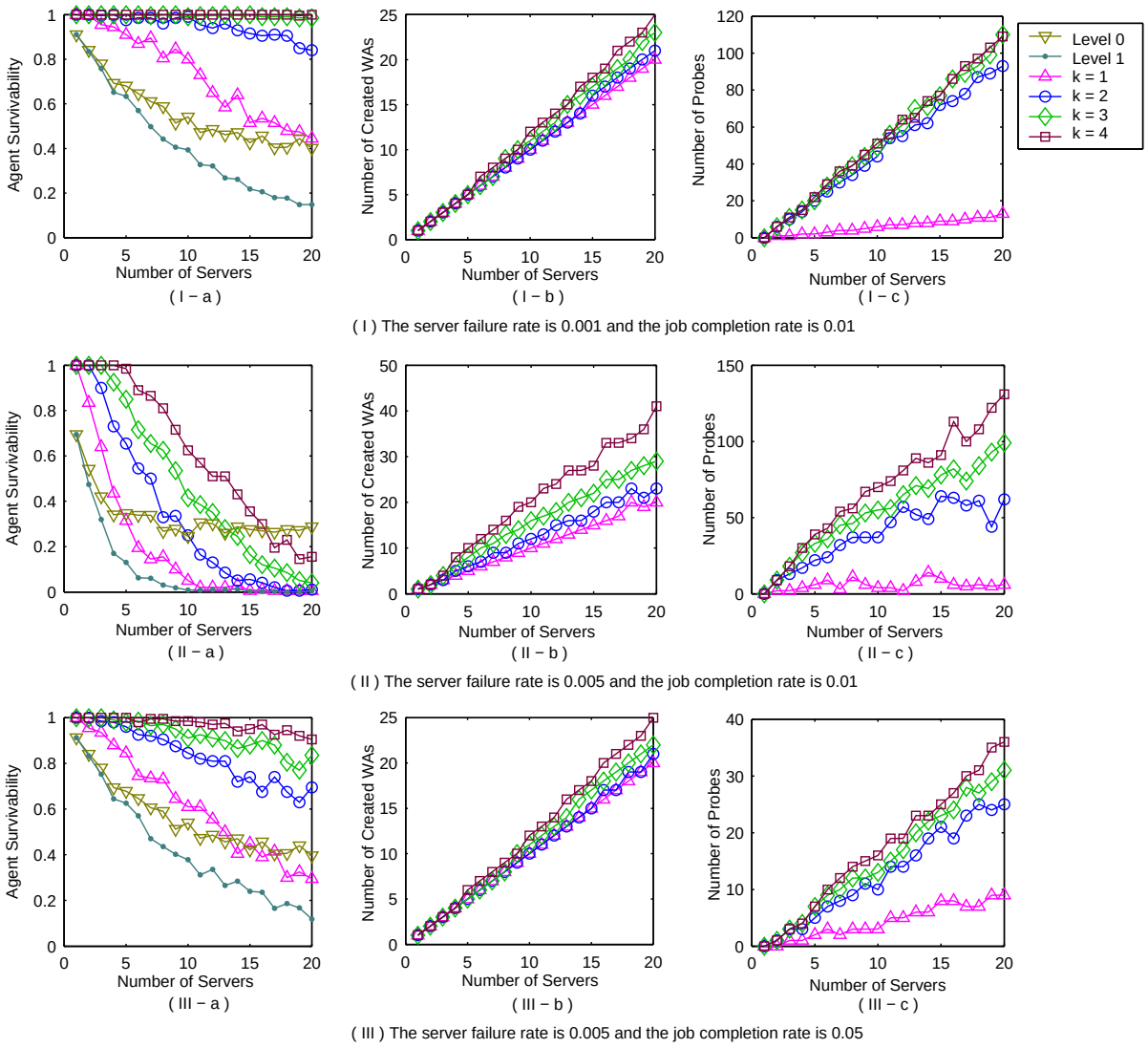

Fig. 6. Simulation results with different server failure rates and job completion rates

completed agent is achieved at the cost of more witness agents and probes spawned. It indicates that as the itinerary becomes longer, more extra witness agents and probes will be required, and consequently the complexity of the system is increased. With these simulation results, we also note there exists a trade-off between the achieved survivability and the overhead cost.

V. Conclusion

Enhancing the agent survivability in a failure-prone mobile agent system should be exploited in order to create a more reliable agent deployment environment. Different approaches have been

studied, such as server replication, checkpointing and rollback recovery, as well as message-based mechanisms. We propose a fault tolerant mobile agent framework which not only integrates those traditional fault tolerance strategies but also employs witness agents and probes to detect and recover server and agent failures. Spawned witness agents form a witnessing dependency which is maintained by heartbeat messages sent backward. Failures of agents are recovered through utilizing probes. An SPN model for the proposed mechanism is constructed and simulations are conducted to evaluate the agent survivability and the numbers of witness agents and probes created for failure recovery. Simulation results show that our proposed agent detection and recovery approach improves the agent survivability. However, the improvement in agent survivability is achieved by spending more time and space resources. Therefore, how to achieve the expected agent survivability with affordable cost is a trade-off issue and should be investigated in the future.

REFERENCES

[1] S. Pleisch and A. Schiper, "Fault-tolerant mobile agent execution," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 209–222, Feb. 2003.

[2] M. Dalmeijer, E. Rietjens, D. Hammer, A. Aerts, and M. Soede, "A reliable mobile agents architecture," in *Proc. of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan, Apr. 1998, pp. 64–72.

[3] T. Osman, W. Wagealla, and A. Bargiela, "An approach to rollback recovery of collaborating mobile agents," *IEEE Trans. Syst., Man, Cybern. C*, vol. 34, no. 1, pp. 48–57, Feb. 2004.

[4] S. Pears, J. Xu, and C. Boldyreff, "Mobile agent fault tolerance for information retrieval applications: An exception handling approach," in *Proc. the 6th International Symposium on Autonomous Decentralized Systems*, Pisa, Italy, Apr. 2003, pp. 115–122.

[5] L. M. Silva, V. Batista, and J. G. Silva, "Fault-tolerant execution of mobile agents," in *Proc. of International Conference on Dependable Systems and Networks*, New York, June 2000, pp. 135–143.

[6] M. R. Lyu and T. Y. Wong, "A progressive fault tolerant mechanism in mobile agent systems," in *Proc. of the 7th World Multiconference on Systemics, Cybernetics and Informatics*, vol. IX, Orlando, Florida, July 2003, pp. 299–306.

[7] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov, "NAP: Practical fault-tolerance for itinerant computations," in *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, USA, June 1999, pp. 180–189.

[8] M. Strasser and K. Pothernel, "System mechanisms for partial rollback of mobile agent execution," in *Proc. of the 20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, Apr. 2000, pp. 20–28.

[9] R. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1996.

[10] R. Jokl and S. Racek, "C-Sim version 5.1," Univ. of Wet Bohemia in Pilsen, Tech. Rep. DCSE/TR-2003-17, May 2003.

Michael R. Lyu received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1981, the M.S. degree in computer engineering from University of California, Santa Barbara, in 1985, and the Ph.D. degree in computer science from University of California, Los Angeles, in 1988.

He is currently a Professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong. He was with the Jet Propulsion Laboratory as a Technical Staff Member from 1988 to 1990. From 1990 to 1992, he was with the Department of Electrical and Computer Engineering, The University of Iowa, Iowa City, as an Assistant Professor. From 1992 to 1995, he was a Member of the Technical Staff in the applied research area of Bell Communications Research (Bellcore), Morristown, New Jersey. From 1995 to 1997, he was a Research Member of the Technical Staff at Bell Laboratories, Murray Hill, New Jersey. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, wireless communication networks, Web technologies, digital libraries, and E-commerce systems. He has published over 170 refereed journal and conference papers in these areas. He received Best Paper Awards in ISSRE'98 and ISSRE'2003. He has participated in more than 30 industrial projects, and helped to develop many commercial systems and software tools. He was the editor of two book volumes: Software Fault Tolerance (New York: Wiley, 1995) and The Handbook of Software Reliability Engineering (Piscataway, NJ: IEEE and New York: McGraw-Hill, 1996).

Dr. Lyu initiated the First International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE'96, and has served in program committees for many conferences, including ISSRE, SRDS, HASE, ICECCS, ISIT, FTCS, DSN, ICDSN, EUROMICRO, APSEC, PRDC, PSAM, ICCCN, ISESE, and WWW. He was the General Chair for ISSRE2001, and the WWW10 Program Co-Chair. He has been frequently invited as a keynote or tutorial speaker to conferences and workshops in U.S., Europe, and Asia. He served on the Editorial Board of IEEE Transactions on Knowledge and Data Engineering, and has been an Associate Editor of IEEE Transactions on Reliability and Journal of Information Science and Engineering. Dr. Lyu is a fellow of IEEE.

Xinyu Chen received the B.E. degree in mechanical engineering from Beijing Institute of Technology, Beijing, China, in 1997 and the M.E. degree in signal and information processing from Peking University, Beijing, China, in 2000. Now he is a Ph.D. candidate in the Department of Computer Science and Engineering at The Chinese University of Hong Kong, Hong Kong, China. His research interests include fault-tolerant distributed systems and mathematical modelling.

Tsz Yeung Wong received the B.S. and M.Phil. degrees in computer science from The Chinese University of Hong Kong, Hong Kong, China, in 2000 and 2002, respectively. Now he is a Ph.D. candidate in the Department of Computer Science and Engineering at The Chinese University of Hong Kong, Hong Kong, China. His research interests include distributed algorithms, graph algorithms, networking, and computer and network security.