Design and Evaluation of a Fault-Tolerant Mobile-Agent System

Michael R. Lyu, Xinyu Chen, and Tsz Yeung Wong, *Chinese University of Hong Kong*

This fault tolerance approach deploys three kinds of cooperating agents to detect server and agent failures and recover services in mobile-agent systems.

When mobile agents travel from one server to another in a network, they transfer their code, data, and execution state to the server. Because they access servers locally, transferring multiple requests and responses across congested network links isn't necessary, thus making overall performance more efficient. Consequently, mobile agents create a new paradigm for data exchange and resource sharing in rapidly growing and continually changing computer networks.

In a distributed system, failures can occur in any software or hardware component. A mobile agent can get lost when its hosting server crashes during execution, or it can get dropped in a congested network. Therefore, survivability and fault tolerance are vital issues for deploying mobile-agent systems. The "Related Work" sidebar discusses various approaches in these areas.

Our method,[1] rooted in the approach of Dag Johansen and his colleagues,[2] employs three types of agents to detect server and agent failures and recover services in mobile-agent systems. An *actual agent* is a common mobile agent that performs specific computations for its owner. *Witness agents* monitor the actual agent and detect whether it's lost. A *probe* recovers the failed actual agent and the witness agents. A peer-to-peer message-passing mechanism stands between each actual agent and its witness agents to perform failure detection and recovery through time-bounded information exchange; a log records the actual agent's actions. When failures occur, the system performs *rollback recovery* to abort uncommitted actions.[3] Moreover, our method uses *checkpointed data* to recover the lost actual agent.[4]

System architecture and protocol design

Researchers have exploited various server failure-detection-and-recovery (FDR) strategies to bring failed servers back online. However, these strategies don't recover a lost actual agent if it resides on the failed server when the failure occurs. Therefore, we need a more advanced approach to reinitialize lost agents.

Figure 1 shows the overall design of our agent server architecture, which can recover lost agents. The agent server should provide three types of stable storage—for logs, checkpoints, and messages. Every server logs the actions that an agent performs. The logged information is vital for failure detection and recovery. Moreover, the hosting servers log which objects the system has updated. When a server failure occurs, the system should recover the agent that was lost due to the failure. However, each agent contains its internal data, which could also be lost due to the failure. In addition, if the agent renews its computation from the starting point of its itinerary, it will violate the exactly-once property. Therefore, the system must checkpoint each agent's data, thus requiring a way to permanently store that checkpointed data. Furthermore, our protocol for agent failure detection and recovery is based on message passing and message logging. To detect and recover an actual agent's failures, the witness agent monitors whether the actual agent is alive or dead (that is, lost). When the actual agent completes its dedicated work on a server and resumes its journey to the next server, it spawns a new witness agent at the current server. We've also designed a communication mechanism between agents and servers.

Assume an actual agent has just arrived at server S_i. Also, assume that a witness agent was spawned at server S_{i-1} before the actual agent left that server. We denote the actual agent as α and the witness agent as

1541-1672/04/$20.00 © 2004 IEEE
Published by the IEEE Computer Society

ω_{i-1}. Because the actual agent plays an active role in our proposed protocol, we discuss its activity first.

Figure 1 shows the action flow that α performs at server S_i. After α arrives at S_i, it immediately writes an arrival entry, log^i_{arrive}, into the logs of the permanent storage in S_i (Step 1). This log entry provides evidence that α has successfully reached this server. Next, α informs ω_{i-1} that it has safely arrived at S_i 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 actual agent's computations. So, if the checkpointing action fails, the actual agent aborts the entire transaction. This step is important because it guarantees that the checkpointed data will be available if the actual agent has already finished computing. Moreover, it's essential for recovering a lost actual agent. Next, α logs another log^i_{leave} entry in S_i (Step 4). This entry expresses that α has completed its computation and is ready to travel to the next server, S_{i+1}. In the following step, α sends ω_{i-1} another message, msg^i_{leave}, 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). Finally, α leaves S_i and travels to S_{i+1}. This procedure continues until α reaches the last destination in its itinerary.

On the other hand, witness agent ω_{i-1} is more passive than the actual agent in this protocol. It doesn't send any messages to the actual agent. Instead, it simply waits to receive messages from the local mailbox. Two messages are expected: msg^i_{arrive} and msg^i_{leave}. One advantage of receiving these two types of messages through a mailbox is that the mailbox provides a history record that they've arrived at this server. Additionally, the mailbox provides a mechanism to shuffle messages, and it only lets msg^i_{arrive} pass before msg^i_{leave}. If the messages are out of order, the mailbox detains msg^i_{leave} in permanent storage, and ω_{i-1} will not consume this message. The mailbox's message record helps recover the lost witness agent and the actual agent. After receiving these two indirect messages, ω_{i-1} waits for the direct heartbeat message, msg^i_{alive}, which the witness agent at server S_i sends. This message confirms the liveness of ω_i. Thus, a witness agent undergoes three states after being spawned, as Figure 2 shows.

Related Work

Extensive research has occurred in the areas of survivability and fault tolerance. Stefan Pleisch and André Schiper adopt the use of replication and masking,[1] employing replicated servers to mask failures. Manfred Dalmeijer and his colleagues use a checkpoint manager to monitor all agents.[2] This manager is responsible for tracking all agents and restarting those that have failed. Taha Osman, Waleed Wagealla, and Andrzej Bargiela analyze an execution model for agent platforms to develop a pragmatic framework for fault tolerance in agent systems.[3] This framework deploys a communication-pair, independent-checkpointing strategy. Simon Pears, Jie Xu, and Cornelia Boldyreff use two exception-handling approaches operating on different servers to maintain mobile agents' availability.[4] Luis Moura Silva, Vitor Batista, and Joao Gabriel Silva present a set of fault tolerance techniques, including fault detection, checkpointing and restart, software rejuvenation, and reconfigurable itinerary.[5] They also discuss issues regarding network partitions.

References

1. S. Pleisch and A. Schiper, "Fault-Tolerant Mobile Agent Execution," *IEEE Trans. Computing.*, vol. 52, no. 2, pp. 209–222.
2. M. Dalmeijer et al., "A Reliable Mobile Agents Architecture," *Proc. 1st Int'l Symp. Object-Oriented Real-Time Distributed Computing*, IEEE CS Press, 1998, pp. 64–72.
3. T. Osman, W. Wagealla, and A. Bargiela, "An Approach to Rollback Recovery of Collaborating Mobile Agents," *IEEE Trans. Systems, Man and Cybernetics*, Part C, vol. 34, no. 1, pp. 48–57.
4. S. Pears, J. Xu, and C. Boldyreff, "Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach," *Proc. 6th Int'l Symp. Autonomous Decentralized Systems*, IEEE CS Press, 2003, pp. 115–122.
5. L.M. Silva, V. Batista, and J.G. Silva, "Fault-Tolerant Execution of Mobile Agents," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2000, pp. 135–143.

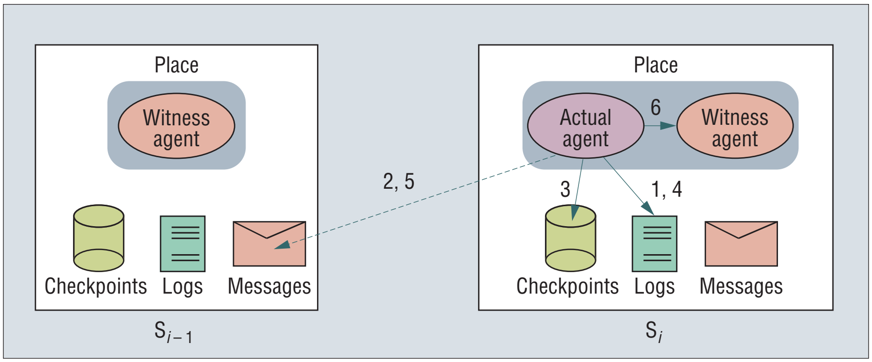

Figure 1. Steps in a fault-tolerant mobile-agent server framework: (1) Log entry log^i_{arrive}. (2) Send message msg^i_{arrive} to server S_{i-1}. (3) After computation, checkpoint the data. (4) Log entry log^i_{leave}. (5) Send message msg^i_{leave} to server S_{i-1}. (6) Spawn a witness agent.

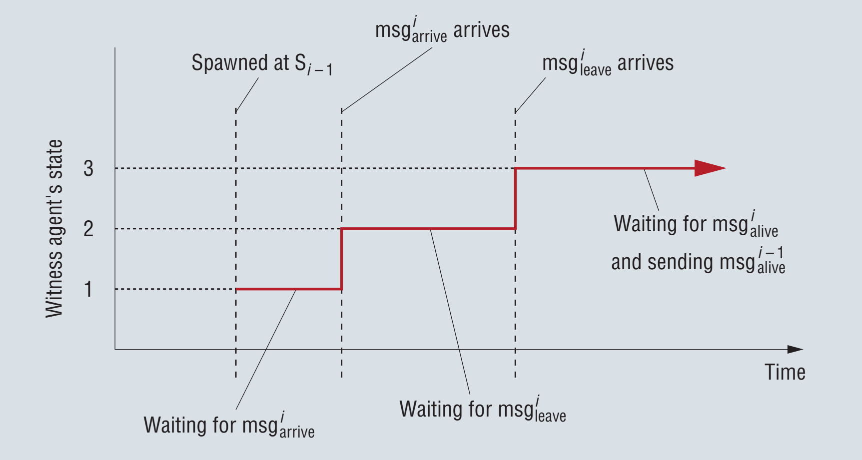

Figure 2. Life scenario of witness agent ω_{i-1}.

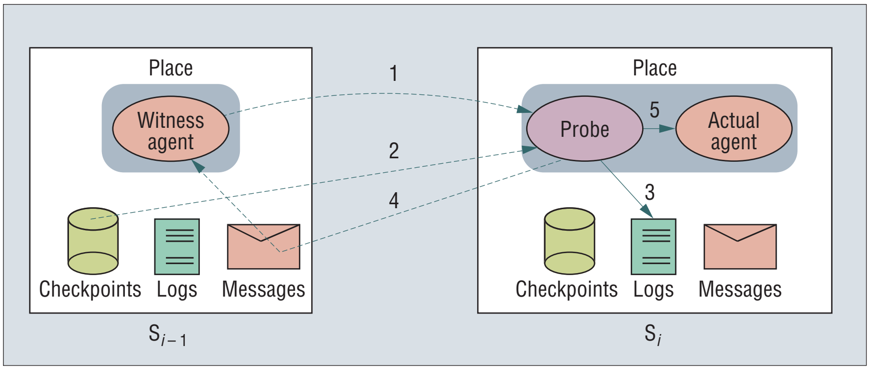

Figure 3. Recovery steps when ω_{i-1} fails to receive msg^i_{arrive}: (1) The witness agent spawns a probe, which travels to S_i. (2) The probe carries the checkpointed data. (3) The probe inspects the log in S_i. (4) If log^i_{arrive} is found, the probe retransmits msg^i_{arrive} to S_{i-1}. (5) If not, it recovers the agent from the checkpointed data.

Agent failure detection and recovery

The purpose of log entries log^i_{arrive} and log^i_{leave} and 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 failed. We assume there are no hardware failures such that the log entries can't be recorded in permanent storage. However, other kinds of failures, such as software faults in the mobile agents or in the mobile-agent platforms, can occur. Here, we discuss different types of failures, including loss of the actual agent and loss of the witness agents.

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

Witness agent ω_{i-1} could fail to receive msg^i_{arrive} at server S_{i-1} for one of the following reasons:

1. The message is lost due to an unreliable network.
2. The message arrives after the timeout period of ω_{i-1}.
3. Actual agent α gets lost when it's ready to leave S_{i-1} and is heading for S_i.
4. Actual agent α gets lost when it arrives at S_i without logging.
5. Actual agent α gets lost when it arrives at S_i with logging.

By using arrival entry log^i_{arrive} logged in S_i, we can solve the first two problems. The actual agent doesn't die, and log^i_{arrive} proves the existence of α inside S_i. The witness agent can then send out probe ρ_i, another agent, to search for log^i_{arrive} in S_i. If found, ρ_i retransmits msg^i_{arrive} to recover the lost or delayed message. If ω_{i-1} fails to receive msg^i_{arrive} because of the loss of α, there could be a *missing-detection* problem: In Case 5 (the fifth reason just listed), the probe might find log^i_{arrive} and, wrongly determining that α is still alive, terminate itself prematurely. If Cases 3 or 4 cause the failure, the probe won't be able to find log^i_{arrive} in S_i. Then, we should recover the lost actual agent by using the checkpointed data stored in S_{i-1}. Therefore, the probe must carry along the checkpointed data when it travels to S_i.

Figure 3 shows the execution steps to detect agent failures when the witness agent fails to receive msg^i_{arrive}. Witness agent ω_{i-1} waits for message msg^i_{arrive} with a configurable timeout period. If the timeout period is reached, ω_{i-1} creates probe ρ_i, which then travels to S_i (Step 1). Because ρ_i might have 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 entry log^i_{arrive} (Step 3). If ρ_i finds log^i_{arrive}, it retransmits msg^i_{arrive} (Step 4). If ρ_i doesn't find the log entry, it recovers α in S_i by using the piggyback checkpointed data (Step 5). Finally, the recovered actual agent at S_i sends message msg^i_{arrive}.

The system recovers the lost actual agent in S_i instead of S_{i-1} because when ρ_i detects that a recovery is necessary, the system can immediately recover that actual agent in S_i. If the system performs the recovery in S_{i-1}, ρ_i must send a message to S_{i-1} to inform ω_{i-1} that an agent recovery is necessary. 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 because ρ_i could be lost, the messages retransmitted from S_i could be lost, or another successive failure might strike S_i. Such a failure could terminate both ρ_i and the just-recovered actual agent. Therefore, ω_{i-1} should wait until message msg^i_{arrive} arrives.

It's possible for ρ_i to reach S_i while α is still on the way. However, the probability of this case occurring should be low. Because both α and ρ_i must travel from S_{i-1} to S_i in the same network, they suffer from more or less the same network latency. Although there might 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 differences in speed among these routes.

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

ω_{i-1} could fail to receive msg^i_{leave} for one of the following reasons:

1. The message is lost due to an unreliable network.
2. The message arrives after the timeout period of ω_{i-1}.
3. Actual agent α gets lost just after sending message msg^i_{arrive}.
4. Actual agent α gets lost just after logging entry log^i_{leave}.
5. Actual agent α gets lost after spawning witness agent ω_i.

If the failure occurs because of one of the first two reasons, the system can handle it in a way similar to that just discussed. Witness agent ω_{i-1} sends probe ρ_i to search for log^i_{leave} in the log file of S_i. The missing-detection problem could occur if the reason for the failures is Case 4 or 5. We'll discuss the solution to Case 4 later; Case 5 is the same as the case in which ω_i can't receive msg^{i+1}_{arrive}, which we discussed in the previous subsection.

For Case 3, probe ρ_i checks whether log^i_{leave} exists. The absence of log^i_{leave} implies that the actual agent was lost while performing its computation. (Case 5 of the previous section would fall into this category.) We expect that ω_{i-1} won't receive msg^i_{leave} after the loss of α. So, because α is lost, the system must undo its partially completed task. Therefore, it's necessary to roll back those operations using Markus Strasser and Kurt Pothernel's method for preserving the data consistency in S_i.[3] The system treats the entire computation process as a single transaction. Because the transaction doesn't fully complete, the system must abort all actions executed in this transaction. The log in S_i serves to recover the data inside S_i. Probe ρ_i doesn't perform the rollback recovery; instead, the system performs this rollback during the server's recovery. Therefore, if the probe can't find log entry $\log^i_{\text{leave}}$, it can immediately use the checkpointed data to recover α. After the recovery is complete, the recovered actual agent continues to perform its computation in S_i. This simplifies the agent failure detection mechanism's implementation.

The probe's execution steps when msg^i_{leave} is missing are similar to the steps in Figure 3. Again, the actual agent's recovery occurs in the server expected to host the actual agent—that is, in S_i.

Failures of witness agents and the recovery strategy

After the actual agent logs entry $\log^i_{\text{leave}}$ and before it moves to the next server, S_{i+1}, it spawns witness agent ω_i at server S_i. The reasons for engaging this witness-agent-spawning strategy instead of letting lagged witness agent ω_{i-1} move forward to server S_i are first, to reduce network communication, thus minimizing the chances of agent loss introduced by link failures, and second, to create a chain of witness agents.

As the actual agent proceeds along its itinerary, the system spawns witness agents along the way. The most recently created witness agent monitors the actual agent; the older witness agents monitor the witness agent that's just one server closer to the actual agent in its itinerary. That is, using "→" to represent the monitoring relation,

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

Now, we introduce a server called *home* (the agent owner's machine). The home server transmits agents as they start traveling on the network and receives agents when they finish. We let S_0 denote this home server. Therefore, ω_0 denotes the witness agent residing at the home server. This *witnessing dependency* can't be broken; otherwise, there'd eventually be no witness agent to monitor α.

To preserve the witnessing dependency, the witness agents not monitoring the actual agent periodically receive heartbeat messages from the next witness agents. That is, ω_i sends a periodic message, msg^i_{alive}, to ω_{i-1} to inform it that ω_i is alive; ω_{i-1} sends a periodic message, msg^{i-1}_{alive}, to ω_{i-2} to inform it that ω_{i-1} is alive; and so on. There are three possible reasons why ω_{i-1} can't receive msg^i_{alive} from ω_i:

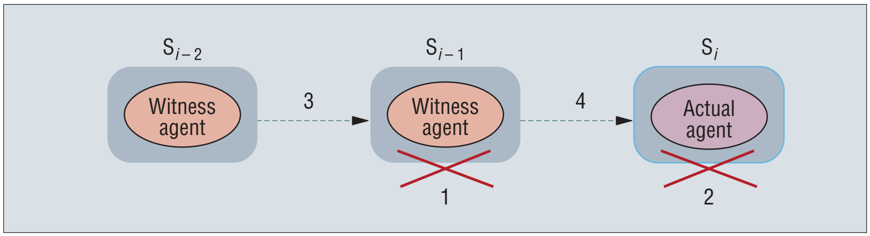

Figure 4. Witness agent failure scenario: (1) A failure strikes S_{i-1}, and the witnessing dependency is broken. (2) A failure strikes S_i, and the actual agent is terminated. (3) The witness agent at S_{i-2} recovers the witness agent at S_{i-1}. (4) The witness agent at S_{i-1} recovers the actual agent at S_i.

1. The network is congested or unreliable.
2. The system load of S_i is too high.
3. Witness agent ω_i was not created or is lost.

Regardless of the reason, ω_{i-1} can always assume that ω_i is lost. After timeout, ω_{i-1} sends ρ_i to S_i to replace the lost witness agent in S_i. There's no special data stored in the witness agent except the agent itinerary, so this type of probe need not carry the checkpointed data. Because there's a probability of false detection due to Cases 1 and 2, when ρ_i reaches S_i, it first checks whether the witness agent is still alive. If no witness agent exists, ρ_i initializes a new witness agent, which resends message msg^i_{alive} to ω_{i-1}; otherwise, ρ_i just disposes of itself.

Figure 4 illustrates a recovery procedure for a witness agent failure. If α is in S_i, then ω_{i-1} is monitoring α, and ω_{i-2} is monitoring ω_{i-1}. Assume the following failure sequence: First, S_{i-1} crashes, then S_i. Because S_{i-1} crashes, ω_{i-1} is lost, so there's no agent to monitor α. If the system doesn't recover ω_{i-1}, α is not recoverable after S_i crashes. This is obviously not desirable, so we need a mechanism to monitor and recover the failed witness agents. We achieve this by preserving the witnessing dependency: Witness agent ω_{i-2} can perform the recovery of ω_{i-1}, so that eventually ω_{i-1} can recover α. There are other, more complex scenarios, but as long as the witnessing dependency is preserved, agent failure detection and recovery are always possible.

Simplifying the witnessing dependency

To maintain the witnessing dependency, the actual agent creates witness agents along its itinerary, and the witness agents exchange heartbeat messages. These procedures consume considerable resources. If, however, no more than k servers can fail at the same time, we can simplify our mechanism by shortening the witnessing dependency. We accomplish this by keeping the witness length less than or equal to k. If α is at server S_i, the simplified dependency becomes

If $(i \le k)$

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

Else

$$\omega_{i-k} \rightarrow \omega_{i-k+1} \rightarrow \ldots \rightarrow \omega_{i-1} \rightarrow \alpha$$

Because no more than k servers can fail simultaneously, k witness agents are sufficient to guarantee the availability of α. When a failure occurs in S_i, ω_{i-1} can recover α after the server restarts. 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}, ω_{i-k} can't be recovered, so the witnessing dependency decreases by 1. However, when α travels to S_{i+1}, α creates a new witness agent ω_i, and a new dependency forms involving ω_{i-1}, ω_i, and α; thus, the witnessing dependency brings the number of witness agents back to k. Finally, when α successfully logs entry $\log^{i+1}_{\text{arrive}}$, the system can terminate ω_{i-k} by sending message msg^{i+1}_{kill} from S_{i+1} to S_{i-k}.

Stochastic Petri net models

Using SPN models and simulations,[5] we evaluated how our agent FDR mechanism improved agent survivability. We denote a mobile-agent system with no fault tolerance as Level 0. For comparison, we introduce a server FDR mechanism. Before α leaves the current server, it checks whether its next destination server is alive. If yes, α moves to it; otherwise, α stays at the current server until the next server comes back to work. If a mobile-agent system engages this server FDR strategy, it's at Level 1. If it additionally embeds the agent FDR strategy, it's at Level 2. We define *agent survivability* as the ratio of successful actual agents (ones that complete their scheduled round-trip journeys) in

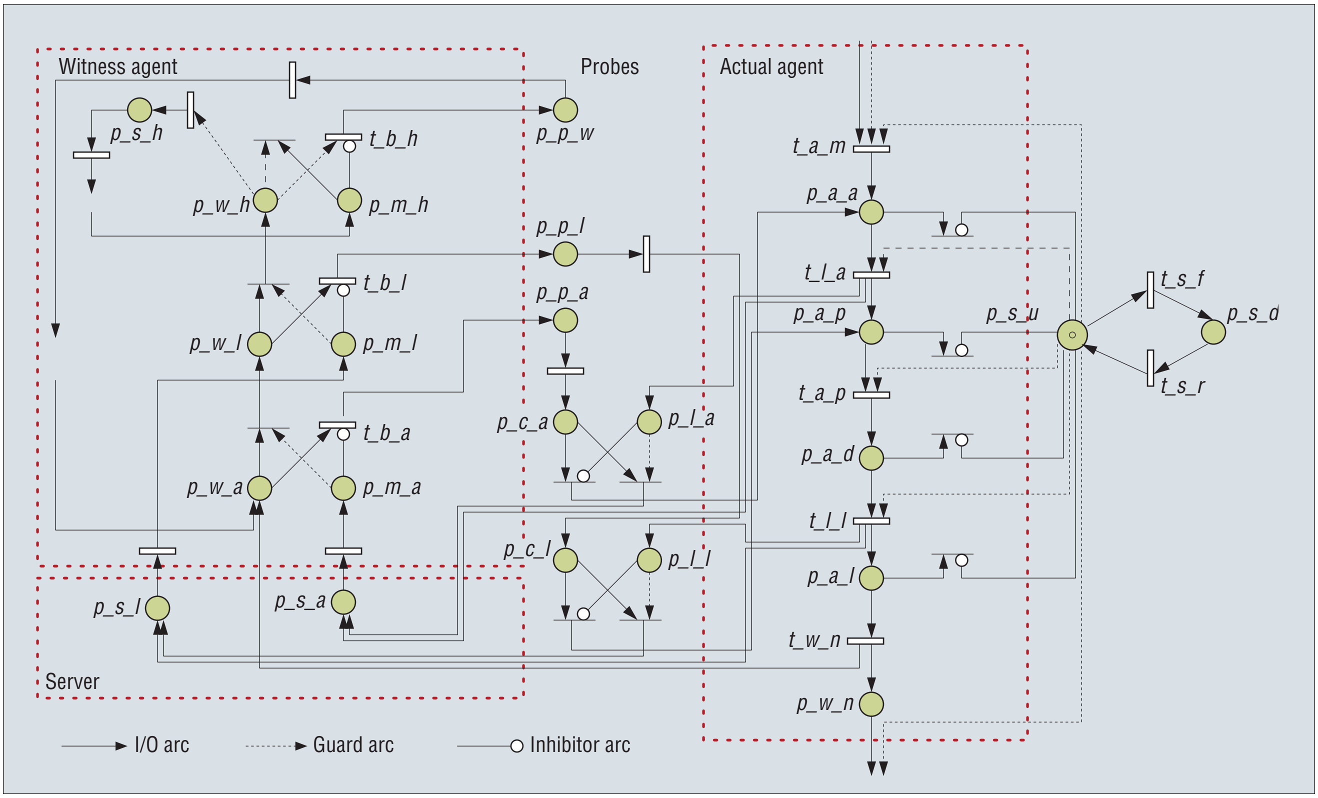

Figure 5. A stochastic Petri net model for Level 2.

a network of agent servers divided by the total number of actual agents launched.

An example model

Figure 5 shows the SPN that models the mobile-agent system at Level 2. The SPNs for Levels 0 and 1 are subsets of the SPN for Level 2, so they need not appear here. The right-hand box manifests the actual agent's state transitions at a server. Transitions *t_a_m*, *t_l_a*, *t_a_p*, *t_l_l*, and *t_w_n* are *timed transitions*; they model the time to travel 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 unboxed places and transitions on the right side of the figure are for the server itself: *t_s_f* models the time to a failure, and *t_s_r* is the time to perform a recovery. (A *place* is an SPN term similar to a *state* in a state diagram.) Here, we assume instant failure detection.

We could also model a more realistic, round-robin failure detection approach.[1] When a *token* is at place *p_s_u*, the server is available. However, if no token is at the place, the server fails, and all agents in that server are lost. We use inhibitor and guard arcs to model these phenomena. The guard arc from *p_s_u* to *t_a_m* prevents actual agents from moving to the server when it fails. The upper-left box shows the witness agent's state transitions. Three places, *p_w_a*, *p_w_l*, and *p_w_h*, represent the witness agent's different states waiting for the arrival message, the leave message, and heartbeat messages, respectively. The middle, unboxed area represents the state transition of probes. The system dispatches three types of probes—those for retrieving the arrival message, those for retrieving the leave message, and those for recovering a witness agent. The two places in the lower-left box represent the server's state after sending the arrival and leave messages, respectively, to a witness agent; the actual agent and the probe share these two places. After a server recovers from a failure, this SPN model initializes places *p_l_a*, *p_l_l*, *p_m_a*, and *p_m_l* with a token if their corresponding logs and messages are present.

Figure 5 shows different agents' behaviors in one server. However, we can link several servers to form a chain, representing an actual agent's itinerary and the witnessing dependency.

Experimental results

We conducted our experiments using simulations developed with C-Sim.[6] Some of the parameters were as follows:

- Network transmission rate: 100 for agents, 200 for messages
- Server repair rate, *t_s_r*: 0.1
- All message log rates: 100
- Arrival, leave, and heartbeat message bound times: 1, 100, and 20, respectively
- Heartbeat interval: 5

We conducted the experiments using different itineraries with various numbers of servers. These experiments illustrate how well agent survivability improved.

Figures 6 through 8 show the results of using the C-Sim implementation with different server failure and job completion rates. For each parameter pair, we conducted six simulations: one for Level 0 (a mobile-agent system with no fault tolerance); one for Level 1 (the same system but with a server FDR strategy); and four for Level 2 (using both server and agent FDR strategies), each with a different number (k) of witness agents. Figure 6 shows that agent survivability decreases

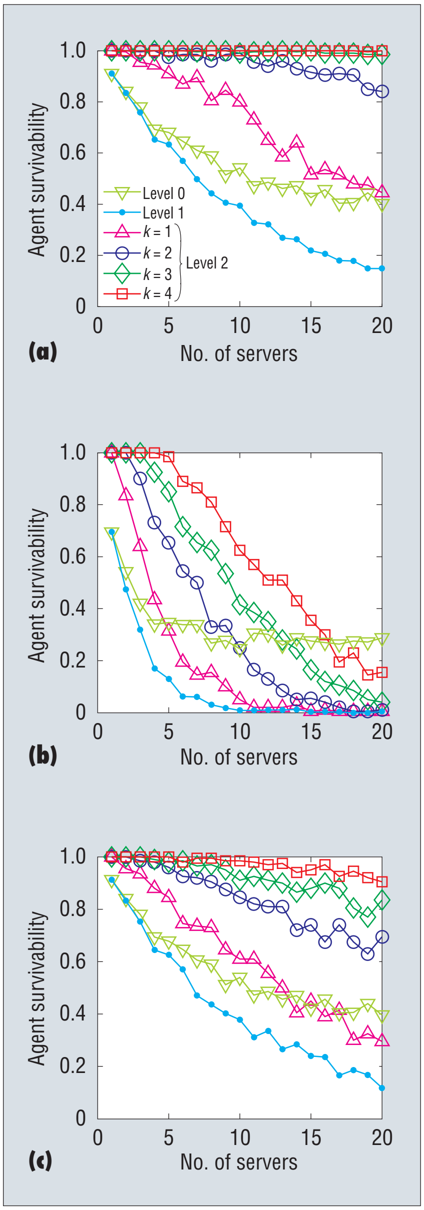

Figure 6. Agent survivability when (a) the server failure rate is 0.001 and the job completion rate is 0.01, (b) the server failure rate is 0.005 and the job completion rate is 0.01, and (c) the server failure rate is 0.005 and the job completion rate is 0.05.

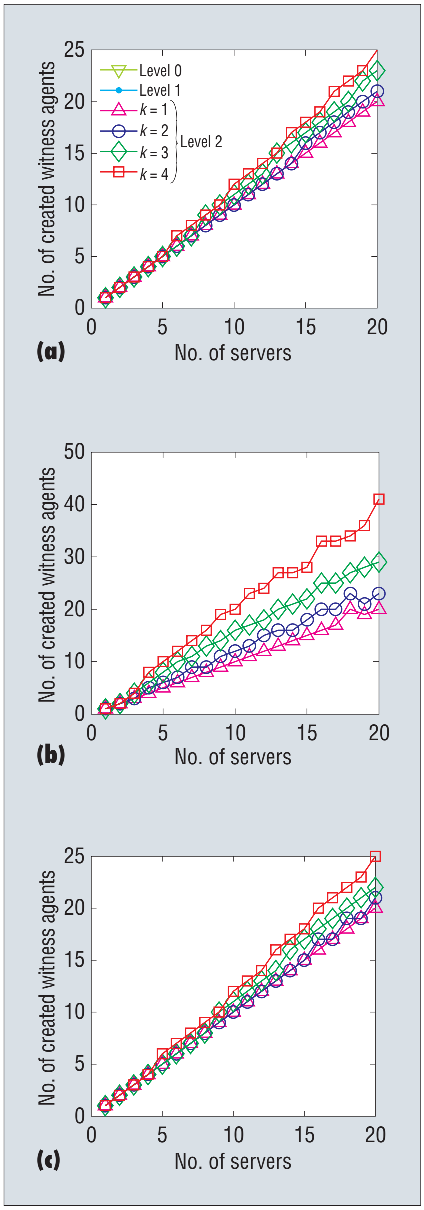

Figure 7. Number of created witness agents when (a) the server failure rate is 0.001 and the job completion rate is 0.01, (b) the server failure rate is 0.005 and the job completion rate is 0.01, and (c) the server failure rate is 0.005 and the job completion rate is 0.05.

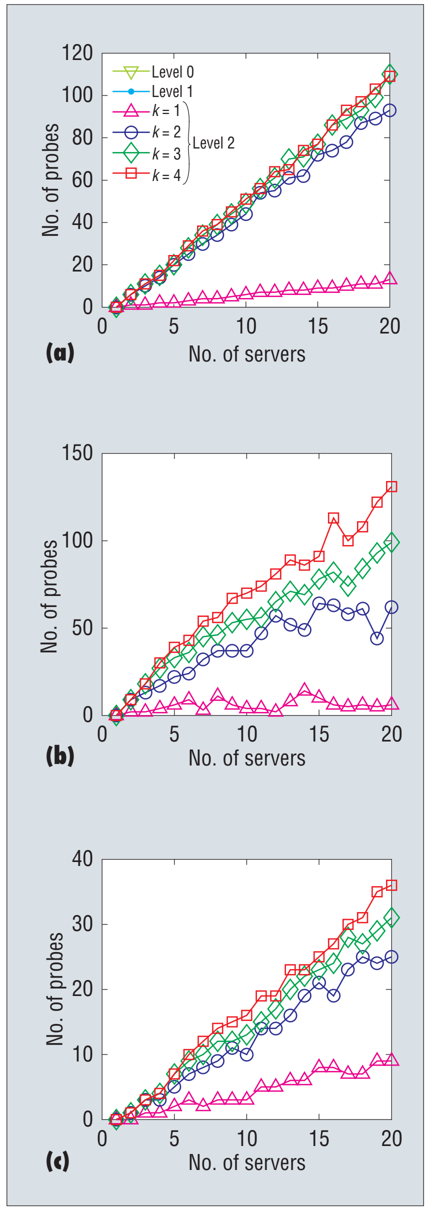

Figure 8. Number of probes when (a) the server failure rate is 0.001 and the job completion rate is 0.01, (b) the server failure rate is 0.005 and the job completion rate is 0.01, and (c) the server failure rate is 0.005 and the job completion rate is 0.05.

progressively as the number of servers increases. This is reasonable: As the chance of having to wait for a failed server to recover increases, the probability of an agent failing while it's waiting also increases. The agent FDR mechanism in Level 2 achieves relatively higher survivability than the other two levels. The improvement becomes more significant as the number of servers and the number of witness agents increase. So, to achieve a high percentage of successful agent round trips with more servers, we should increase the number of witness agents correspondingly.

We found one unexpected result. After engaging the server FDR approach (Level 1), the percentage of completed agents was less than it was in Level 1, without fault tolerance mechanisms (see Figure 6). The problem is that in both these levels, when an actual agent fails, there's no way to recover it. So, if an agent finishes its journey more quickly, the chance of it failing decreases. After engaging the server FDR strategy, the actual agent spends more time in the system because it waits at its current server when its next server is unavailable. Consequently, the chances of it failing increase. This is true even if we use the agent FDR mechanism and a few witness agents. Comparing Figures 6a and 6b, we see that the higher the failure rate, the higher the agent loss probability. However, if an agent completes its dedicated work at each server more quickly (Figure 6c), survivability increases. This implies that in unreliable systems, actual agents should complete their tasks as quickly as possible.

We achieve Level 2 by engaging witness agents and probes. Figures 7 and 8 show the cost of these additional resources. As Figure 7 shows, the number of created witness agents with Level 2 increases linearly with the number of servers. The higher the number (k) of required witness agents, the more witness agents the system creates. Figure 8 shows the number of probes generated during agent execution. The higher percentage of completed agents comes at the cost of more witness agents and probes spawned. This means that as the itinerary becomes longer, more witness agents and probes are necessary, so system complexity increases. The simulation results in Figures 6, 7, and 8 also indicate that there's a trade-off between survivability and overhead cost.

Simulation results show that our proposed agent FDR approach improves agent survivability in failure-prone mobile-agent systems. Thus, it can help create a more reliable agent deployment environment. However, this improvement comes at the expense of time and space resources. Therefore, achieving the expected agent survivability affordably is a trade-off that we need to investigate in the future. ■

References

1. M.R. Lyu and T.Y. Wong, "A Progressive Fault Tolerant Mechanism in Mobile Agent Systems," *Proc. 7th World Multiconf. Systemics, Cybernetics and Informatics*, vol. IX, Int'l Inst. Informatics and Systemics, 2003, pp. 299–306.
2. D. Johansen et al., "NAP: Practical Fault-Tolerance for Itinerant Computations," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems*, IEEE CS Press, 1999, pp. 180–189.
3. M. Strasser and K. Pothernel, "System Mechanisms for Partial Rollback of Mobile Agent Execution," *Proc. 20th IEEE Int'l Conf. Distributed Computing Systems*, IEEE CS Press, 2000, pp. 20–28.
4. M. Dalmeijer et al., "A Reliable Mobile Agents Architecture," *Proc. 1st Int'l Symp. Object-Oriented Real-Time Distributed Computing*, IEEE CS Press, 1998, pp. 64–72.
5. 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, 1996.
6. R. Jokl and S. Racek, *C-Sim Version 5.1*, tech. report DCSE/TR-2003-17, Dept. of Computer Science and Eng., Univ. of West Bohemia in Pilsen, 2003.

The Authors

Michael R. Lyu is a professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, wireless communication networks, Web technologies, digital libraries, and e-commerce systems. He received his PhD in computer science from the University of California, Los Angeles. He is a fellow of the IEEE. Contact him at Dept. of Computer Science and Engineering, Chinese Univ. of Hong Kong, Shatin N.T., Hong Kong SAR; lyu@cse.cuhk.edu.hk.

Xinyu Chen is a PhD candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include fault-tolerant distributed systems and mathematical modeling. He received his ME in signal and information processing from Peking University, Beijing. Contact him at Dept. of Computer Science and Engineering, Chinese Univ. of Hong Kong, Shatin N.T., Hong Kong SAR; xychen@cse.cuhk.edu.hk.

Tsz Yeung Wong is a PhD candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include distributed algorithms, graph algorithms, networking, and computer and network security. He received his MPhil in computer science from the Chinese University of Hong Kong. Contact him at Dept. of Computer Science and Engineering, Chinese Univ. of Hong Kong, Shatin N.T., Hong Kong SAR; tywong@cse.cuhk. edu.hk.