A Progressive Fault Detection and Service Recovery Mechanism in Mobile Agent Systems

WONG, Tsz-Yeung

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

June, 2002

Abstract

In this thesis, we present the approach of deploying cooperating agents to detect failures as well as recover services in a mobile agent system. Failures in the mobile agent system can be classified into three types, namely *server failure*, *agent failure*, and *link failure*. The server failure includes hardware and software failures in the server where agents reside. This can be handled by traditional fault tolerance mechanism in distributed systems. 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 agent,namely the *witness agent*, is the agent that monitors the actual 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. Finally, the link failure includes the failure of the linkage of communication network. This can induce a more severe scenario, such as the network partition. Since link failure is beyond the control of an agent system, the agent system cannot recover it. We suggest modification in our approach to ease the impact of the link failure. We conduct mathematical analysis and reliability evaluation for our approach, which shows that it is a promising technique in achieving mobile agent system reliability.

摘要

在本論文中，我們探討在流動軟體系統中故障偵測及故障修正的研究。在流動軟體系統中發生的錯誤是分爲三大類的：伺服器故障，流動軟體故障，以及網路連結故障。伺服器故障包括伺服器硬體與軟體的錯誤，這些錯誤可以應用分佈式系統的傳統的容錯技術來修正。我們應用流動軟體的合作技術來修正流動軟體故障。我們引入了兩種不同的流動軟體，一種是眞實軟體。眞實軟體的功能是爲它的主人進行運算，最後把運算的結果傳送回主人。另一種是見證軟體。它的功能是見證眞實軟體的運作。我們引入了一套以訊號傳遞爲媒介的協定，作爲兩種不同軟體的溝通渠道以及流動軟體故障修正的方法。這種流動軟體故障修正機關能夠正確處理伺服器故障，流動軟體故障及訊號傳遞故障。這個機關能偵測及修正流動軟體系統中的大部份故障。網路連結故障包括了網路間的連線故障。這些故障可以引發嚴重的後果，例如網路分割。我們提出了故障修正機關的修改以舒緩網路連結故障的影響。我們進行了故障修正機關的可靠性評估，並透過這個評估顯示出這故障修正機關是一個可靠的機關。

Acknowledgement

In completing the work reported in this thesis, I am most grateful to my thesis advisor, Dr. Michael Lyu, who has been giving continuous support and guidance to me throughout the past two years.

I am also obliged to my colleagues in the Department Computer Science and Engineering, especially Pun-Mo Ho, Cheuk-Man Lee, Tak-Fu Tung, Kai-Chun Chiu, and Lap-Chi Lau. They have given me invaluable advice and support in these two years of research life.

Contents

List of Figures

Chapter 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 Internet is provided by a middle-ware platform, a mobile agent execution environment. There are agent research projects done in recent years such as Mole [2]. Also, there are commercial products developed including Aglets [3], Concordia [4] and Tryllian [5]).

Since agents are objects that are traveling in a computer network, it is very complicated and difficult for us to estimate the running time of an agent. It is because the agent may suffer from congestion in the network, or it may be waiting and executing in a busy server. These kinds of uncertainties raise problems to the reliable agent system design. The agent owner cannot tell whether the agent is lost or the execution is delayed. This may lead to two undesirable situations:

- The agent owner believes that the agent has been lost, but in fact it is not. If the owner launches another agent, which may cause multiple executions of the same piece of agent code.
- The agent owner waits for the agent to finish its itinerary, but the agent is actually terminated due to server or agent failures.

Fault-tolerant mobile agent protocol aims to remove the uncertainties during the execution of agents. It should ensure that the agent can eventually reach its destination, or notifies the agent owner of a potential problem. There are restrictions that every fault-tolerant protocol design should follow in addition to the above goals.

Blocking-free. Assume that we have a perfect[1] failure detection mechanism. We can use simple *checkpointing* mechanism to safe-guard the agent execution. For instance, we can back up the whole agent to permanent storage in a node. Once a node crushes, the agent in that failed node is discarded. We can use the backup agent to continue the computation when the failed node is recovered. However, it is *prone to blocking*. The agent execution is blocked until a failure is eventually detected.

Exactly-once. For instance, an user launches an agent to settle a payment. However, he/she is not lucky enough that the agent is trapped inside a busy network, and, hence, the delay becomes huge. The user may assume that the agent may be terminated. Then, he/she launches another agent. Nevertheless, this extra agent may settle the same payment once more. This is an undesirable result. Therefore, we have to hold the *exactly-once* execution property since most of the agent operations are not *idempotent* (or *non-intrusive*).

1.1 Related Work

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* together with *failure masking* [6, 7]. The idea is to use replicated servers to mask failures. When one server is

[1]Perfect failure detection mechanism means that if there exists a failure in the system, we can eventually find it out. However, the time needed to find it out is not guaranteed.

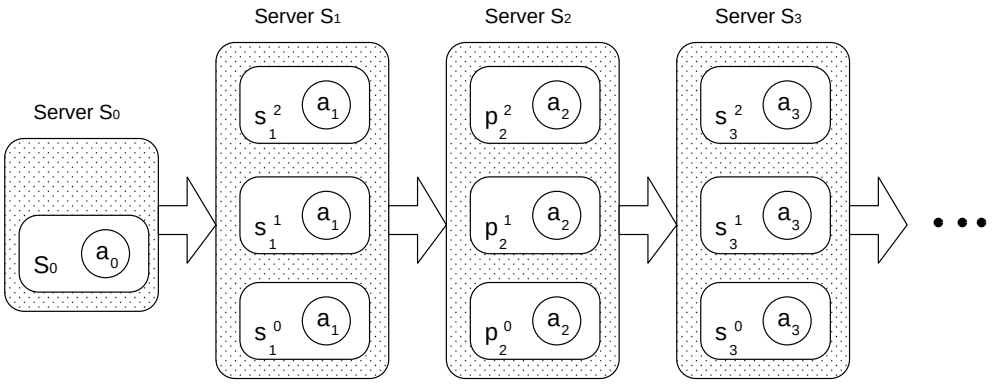

Figure 1.1: Replication deployed in agent system.

down, we can still use the results from other servers in order to continue the computation.

Figure 1.1 shows how the above mechanism works. The servers named S_1, S_2, and S_3 are deploying the replication technique. The server S_1, for example, represents one *logical server*. In reality, there are three servers running at the same time, namely, S_1^0, S_1^1, and S_1^2. On the other hand, an incoming agent arriving at S_1 will be cloned, and three instances of the same agent will be executing on three distinct physical servers simultaneously. After the computations of the three agent instances have finished, the results will be compared. The expected results coming from the three servers should be the same. If there are failures, the outcomes can be different, or one or more servers do not response within a certain time, then the majority and available results will be used. The advantage of this approach is that the computation will not be *blocked* when a failure happens. Failures can be masked when most of the servers are working. Hence, the computation can continue although failures happened.

However, this fault-tolerant 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 data consistency.

On the other hand, Stasser and Pothernel [8] have proposed a protocol in rollback of mobile agent execution. Their main work includes the introduction of compensation operation and the classification of agent data. The compensation of an operation aims at undoing the semantic effects of this operation. Obviously, not all kinds of operation can be compensated. The simplest case is when the operation is non-intrusive, i.e., it will not change the states of both the server and the agent. If the operation changes those states, it is desirable to have a separate compensation operation that can undo all the changes.

Rollback of the execution includes the rollback of data in the server as well as the rollback of private data inside the agent. The data objects in the private data space of the agent can be classified into two categories, namely the *strongly reversible objects* and the *weakly reversible objects*. *Strongly reversible objects* are data objects the can be compensated by means of an image, or the *checkpointed image*, of the objects. *Weakly reversible objects* are data objects that may be different from the original data after the compensation, i.e., cannot be compensated using a before-image. With the introduction of compensation operations and the classification of the agent data, we can establish an effective rollback mechanism.

1.2 Progressive Fault-Tolerant Mechanism

Our approach is rooted from the approach suggested in [9]. We distinguish two types of agents. The first 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*. The witness agent always

travels behind the actual agent. That means the witness agent will follow the itinerary of the actual agent. These two types of agents communicate by using a peer-to-peer message passing mechanism. In addition to the introduction of the witness agent and the message passing mechanism, we require to *log* the actions performed by the actual agents since after failures have happened, the server has to abort uncommitted actions when the system performs *rollback recovery*. Moreover, the approach requires to use *checkpointed data* [10] to recover the lost agents.

The key difference between the protocol suggested in [9] and our protocol is that the former depends on a *reliable broadcast*, while we allow the network to be unreliable. That is, we can remedy the failures in transmission of messages as well as the loss of the agent in the network. In [9], 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*.

Consequently, we propose a progressive failure detection and service recovery mechanism in four levels [11]. Different levels determine the availability and data consistency that can be achieved for the mobile agent systems:

Level 0: *No tolerance to faults in the mobile agent system*

> In this level, when the executing agent process dies, either due to the server failures or the faults inside the agent, it has to be manually restarted from an initial internal state. That means the execution has to be restarted by the agent's owner. On the other hand, the affected server may leave its data in an incorrect or inconsistent state due to system crashes. It may take a long time to restart properly by the manual initialization procedures.

Level 1: *Automatic server failure detection and recovery*

When a server failure happens, the failure will be detected by another program (or a daemon). The detection program restarts the server, and aborts any uncommitted transactions inside the server. This preserves the consistency of the data inside the affected server. However, the agent has to start running from the initial state. When the re-transmitted agent travels to the visited hosts, the data in these servers will be modified twice. This violates the exactly-once execution property of the mobile agent systems [12, 13].

Level 2: *Automatic agent failure detection and recovery*

When a server failure happens, the agents that reside in the failed server will be lost. The loss of agents can be detected in this level. The situation cannot be improved without the help of rollback recovery and checkpointing [10]. The agent performs checkpointing at each host, which checkpoints the internal state of the agent after the agent's execution is completed. When a failure is detected, the checkpointed data can be retrieved for the recovery of the lost agent. The recovery of the agent takes place at the server where the agent fails. Therefore, the exactly-once property is preserved. Moreover, as the internal states of the agent is checkpointed, we preserve the agent data consistency.

Level 3: *Link failure*

We can always model a network as an undirected multi-graph. The network is undirected since we assume that the network is always duplex, and it is a multi-graph because there are multiple links from one node to another. We assume that a network is not always a complete graph, i.e. for a graph G, there exists a vertex u such that the maximum of the shortest path from u to other vertices in G is larger than 1. We further assume that the multiple edges are combined into one edge. This means that the failure of an edge from u to v implies the failures of all the links

from u to v.

We start our discussion by assuming that an actual agent, α, is now in server u and it is ready to migrate to v. A link failure can happen in three different moments: (1) before α leaves u; (2) while α is traveling to v; (3) after α has reached v. The above three cases will have different consequences. This leads to modifications of the *level 2 fault-tolerant mechanism*, which we call the *level 3 fault-tolerant mechanism.*

In the above classification, the corresponding failure detection and recovery mechanisms can only handle the stopping failures caused by software faults in the mobile agents and the mobile agent platform. The hardware failures and the Byzantine failures [14] are out of the cope of this thesis.

1.3 Organization of This Thesis

This thesis is organized in the following way:

- Chapter 1 (this chapter) is an introduction of the thesis. It gives a brief description of mobile agent technology. It also states the problems of fault-prone mobile agent systems, and previous work done in this area. Moreover, it also outlines the contribution and the organization of this thesis.
- Chapter 2 gives an outline of the problems as well as solutions dealing with the servers failure in mobile agent systems. It states the importance of the server failure detection and recovery.
- Chapter 3 focuses on the details of the agent failure detection and recovery mechanisms. It describes the mobile agent system architecture that supports the proposed mechanism, and outlines the protocol of the

mechanism. A detailed discussion on different failure scenarios is provided, and how the mechanism works on these scenarios is described. It also includes a simplification of this mechanism.

- Chapter 4 gives a detailed analysis of the proposed mechanism in the previous chapter. It includes a detailed definition and description of the mechanism. The analysis includes a liveness proof of the mechanism, and the analysis of the simplified mechanism.
- Chapter 5 provides an extension of the mechanism. The extended mechanism discusses the link failures in the system. Since the link failure can hardly recover fully, we propose a solution that can remedy this kind of failures.
- We describe the evaluation of the mechanism in chapter 6. It includes the Concordia implementation and the simulated experiments of the proposed fault-tolerant mobile agent system.
- Finally, chapter 7 concludes this thesis and provides some directions of future research.

1.4 Contribution of The Thesis

This thesis makes the following contributions:

- It designs a progressive fault detection and recovery mobile agent system design by using cooperative agents.
- It provides the impossibility proofs on the liveness of the system and the analysis of the simplification of the proposed mechanism.
- It develops the reliability evaluation experiments by the agent implementation and the stochastic petri nets simulations.

Chapter 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 and recovery are vital issues in the design of a reliable mobile agent system. An agent requires a server to be hosted and to be provided 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 becomes 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. Hence, these problems address the importance of the server failure detection, and outline a series of tasks required to be accomplished during the recovery of servers.

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 while 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

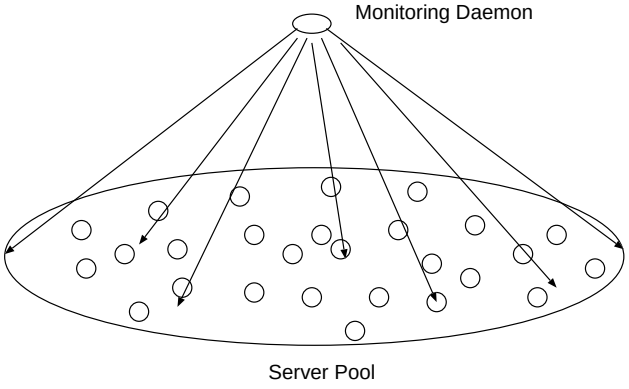

Figure 2.1: Server failure detection daemon.

Our mechanism to detect and recover a server failure is to launch a daemon in a machine as shown in Figure 2.1. 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 of failure*. Since we have only one server monitor in the system, the failure of the server monitor will cause the level 1 fault-tolerant mechanism to be failed. 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.

An alternative approach is suggested by Huang [11]. The main idea of this approach is to use another program to monitor the availability of the server program. The detection of the server availability is mostly done by the operating system by using *fork* and *signal*. The server program is the child program, and the parent program monitors it. When the abort or terminate signal is captured by the parent program, the parent program re-instantiates the server program. It is an easy but, yet, elegant approach. However, in terms of implementation, this approach is not interoperable since it is language dependent.

Chapter 3

Agent Failure Detection and Recovery

We discuss the agent failure detection and recovery mechanism, or the level 2 fault-tolerant mechanism, in this chapter. Our approach maintains the exactly-once property. However, it is block-prone. We introduce the system architecture in Section 3.1. In Section 3.2, we describe the protocol which involves the cooperations between two different kinds of agents. Different failure and recovery scenarios are discussed in Section 3.3. It also addresses the scenarios when the mechanism fails, i.e, the catastrophic failures, and we suggest solutions to remedy these situations. Finally, we have a simplification of the mechanism. The simplification of the mechanism reduces the complexity of message passing and the resources consumed by the mechanism.

3.1 System Architecture

We introduce the system architecture of the mobile agents that are capable of supporting the level 2 fault-tolerant mechanism. In order to detect the failures of the actual agents as well as recover the failed *actual agent*, we design another type of agents, namely the *witness agent*, to witness and monitor whether the actual agent is alive or terminated. Due to the introduction of the witness

agent, we have to design a communication mechanism between both types of agents. In our design, they 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 certainly knows the addresses of the visited servers. 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*. There can be several reasons, e.g., the witness agent is on the way to the target server. 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.

On the other hand, every server has to log the actions performed by an agent. The logging actions are invoked by the actual agents. The information logged by the agent is vital for failure detection as well as recovery[1] . Also, the hosting servers have to log which objects have been updated. These logs are 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 is also lost due to the failure. Moreover, if we allow the agent to start computing from the starting point of the itinerary, the *exactly-once* property will be violated. Therefore, we have to *checkpoint* the data of the agent as well as *rollback* the computation when necessary. The servers are required to have a permanent storage to store the checkpointed data in the server. Moreover, the servers have to log messages in the permanent storage of the server in order to perform rollback of executions. The overall design of the server architecture

[1]The importance of logging is addressed in Section 3.3

is shown in Figure 3.1.

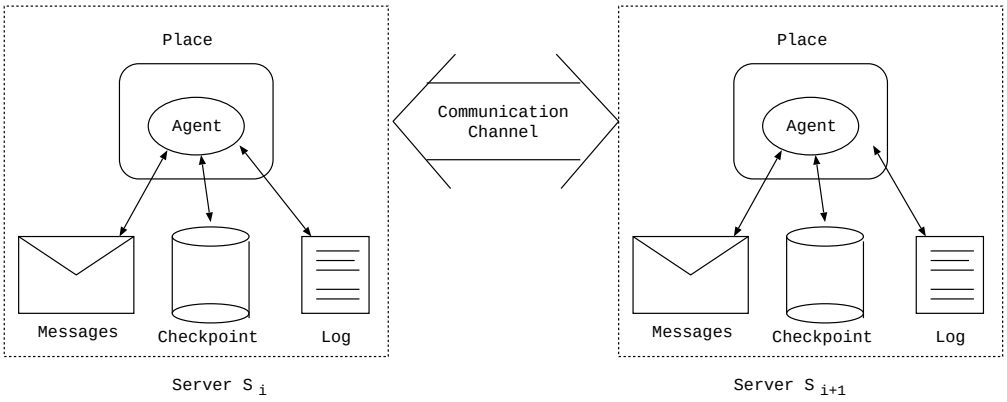

Figure 3.1: The server design.

3.2 Protocol Design

Our protocol depends on messages passing as well as messages 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 first discuss the behavior of the actual agent α. It 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, ω_{i-1} know that the actual agent, α, has successfully arrived at 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}.

Then, α performs the computations delegated by the owner on S_i. When it finishes the computations, it immediately checkpoints its internal data in the permanent storage of S_i. Then, it logs a message log^i_{leave} in S_i. The purpose of this message is 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}.

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}, just arrives at S_{i-1}. Before ω_{i-1} can advance further in the network, it waits for the messages sent from the actual agent, 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 means msg^i_{leave} is considered as unattended[2] , 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 namely 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 both msg^i_{arrive} and msg^i_{leave} contain information about the location of S_i where α has just visited.

Figure 3.2 shows the flow of the protocol. The actual agent, α, just 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

[2]Unattended messages means the target receiver is not in the server, e.g., the witness agent is on the way

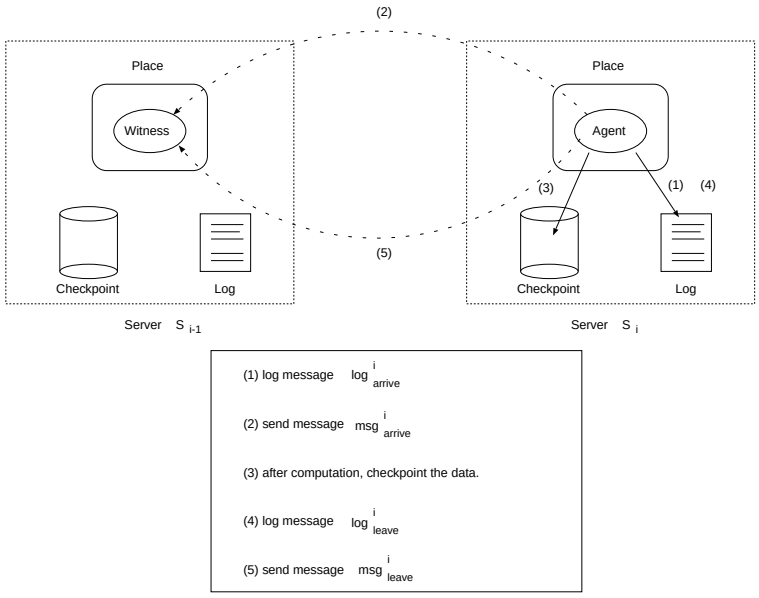

Figure 3.2: Steps in the witness protocol.

important step since this property guarantees that the checkpointed data will be available if the actual agent has 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

In the previous section, we have described the basic of the level 2 fault-tolerant mechanism while this section is extending the previous protocol. In this section, we discuss different scenarios with the presence of faults. We describe the actions of the witness agents in order to detect the loss of the actual and the witness agents and recover the lost agents. We also disclose the purpose of the direct and indirect messages as well as the log messages. Moreover, we

introduce a more kind of agents and a more type of messages.

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. This assumption can forbid the possibility 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 When ω_{i-1} fails to receive msg^i_{arrive}

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

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 terminated when it is ready to leave S_{i-1};
4. α is terminated when it has just arrived at S_i without logging; or
5. α is terminated when it has just arrived at S_i with logging.

If the failures are because of the first two reasons, i.e., the actual agent is not terminated, 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 of α inside S_i. The witness agent can send out a *probe*, ρ_i, to search for log^i_{arrive} in S_i. If the log message is found, ρ_i can re-transmit msg^i_{arrive} in order to recover the lost messages. The probe is another agent. Its responsibility is to search for target log messages in a specified server.

If ω_{i-1} fails to receive msg^i_{arrive} because of the loss of the actual agent, there are chances that the problem of *missing detection* arise. In the fifth case, since the log message log^i_{arrive} is present, the probe would wrongly determine that the actual agent is still alive. However, the actual agent is terminated, so the recovery of the actual agent would be missed. Fortunately, this case can be handled and 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 3.3: ω_{i-1} fails to receive msg^i_{arrive}.

Figure 3.3 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 permanent storage of 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 essential 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 newly 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 When ω_{i-1} fails to receive msg^i_{leave}

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

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 terminated when it has just sent the message msg^i_{arrive}; or
4. α is terminated 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. Recalling from the previous section, the probe mis-interprets the log message log^i_{arrive} in S_i. The probe would believe that the msg^i_{arrive} is lost in the network. However, the agent is actually lost. This case results in missing detection and the probe will re-transmit the expected message, msg^i_{arrive} 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 failures is due to the fourth case. That is, the actual agent is terminated but we have not detected it. These two cases can be settled as follow. 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 terminated and does not exist in S_{i+1}. Consequently, we can successfully detect the agent failure by the third case of the previous subsection.

If the failure happens because of 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 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.

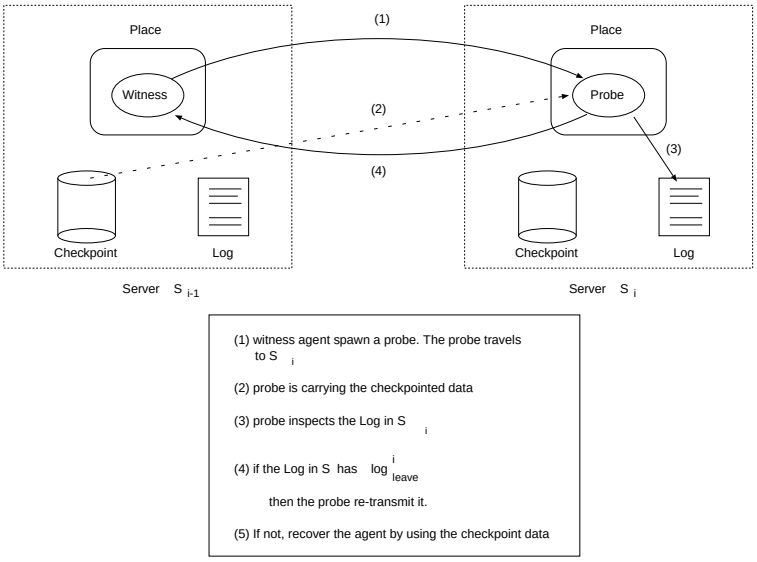

Figure 3.4: ω_{i-1} fails to receive msg^i_{leave}

The execution steps of the probe when log^i_{leave} is missing is very similar to the steps in Figure 3.3. It is shown in Figure 3.4 Note that for both failure scenarios, the recovery 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 the witness agents and recovery scenarios

Before the actual agent completes its itinerary, there are witness agents spawned along the itinerary of the actual agent. The youngest 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 :

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

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

We name the above dependency the *witnessing dependency*. For instance, if α is in S_i. ω_{i-1} is monitoring α, and ω_{i-2} is monitoring ω_{i-1}. This dependency cannot be broken. Assuming we have the following failure sequence: S_{i-1} crushes first 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 a disastrous scenario (Figure 3.5 illustrates this scenario.). Therefore, we need a mechanism to monitor and recover the lost 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}.

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, those witness agents that are not monitoring the actual agent receive periodic messages from the witness agent that they are monitoring. That mean ω_i sends periodic messages to ω_{i-1} in order to let ω_{i-1} knows that ω_i is alive. We label this

message as msg^i_{alive}. When ω_{i-1} cannot receive msg^i_{alive} from ω_i, the reasons can be classified as follow:

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 3.6). 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 should prohibit multiple instances of ω_i from executing.

Figure 3.6 summarizes the life cycle of a witness agent. When a witness agent ω_i is first created, it travels to its destination S_i [State (1)]. When it reaches S_i, it starts waiting for the message msg^{i+1}_{arrive} [State (2)]. If the message comes earlier than ω_i, ω_i can find it in the mailbox at S_i. After msg^{i+1}_{arrive} has been received, ω_i starts waiting for msg^{i+1}_{leave} [State (3)]. At last, α leaves S_{i+1}. Then, ω_i spawns ω_{i+1}. Its job is then switched from monitoring α to monitoring ω_{i+1}. In the meanwhile, it continuously sends msg^i_{alive} to ω_{i-1} periodically [State (4)]. However, not all witness agents are starting its life from *State 1*. Some witness agents start its life from *State 4* as they are responsible of recovering the lost witness agents.

When the actual agent has finished all the computations in its itinerary, all the witness agents should be terminated. The method of terminating the agents along the itinerary can be done by sending a sequence of terminating message along the itinerary of the actual agent. We name that message log_{term}. log_{term} will be kept in the permanent storage of the servers. When a witness

agent finds this log message in its hosting server, it will be terminated. A similar but detailed approach is described in [15], which deals with a different agent application for an orphan detection problem.

3.3.4 Catastrophic failures

The witness agent protocol cannot guarantee that all failures can be detected and recovered. First of all, the witnessing dependency cannot be always preserved. The weakness is at the *starting node* 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 if 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.

We provide a solution that can ease the catastrophic failures. 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 in order to handle this failure series. The effect of sending this *witness agent* is similar to the case when a *witness agent*, ω_i, fails to receive msg_{alive}^{i+1}. This method can recover ω_0 and the *witness dependency* effectively with an appropriate timeout period. However, the drawback is that the owner has to send out periodic agents to S_0.

3.4 Simplification

Note the witnessing dependency is useful only when several servers fail in a short period of time. Nevertheless, this dependency uses a lot of resources along the itinerary of the actual agent. If we can assume that *no two or more servers can fail at a short 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 terminating message from S_i to S_{i-1}. The key of this simplification is how long is the period between two failures. We would have a detailed analysis in section 4.4.

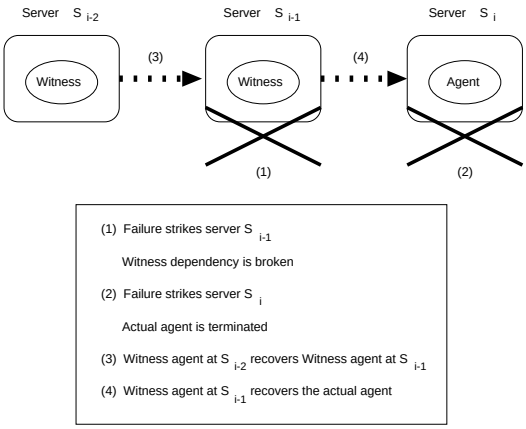

Figure 3.5: Witness agent failure scenario

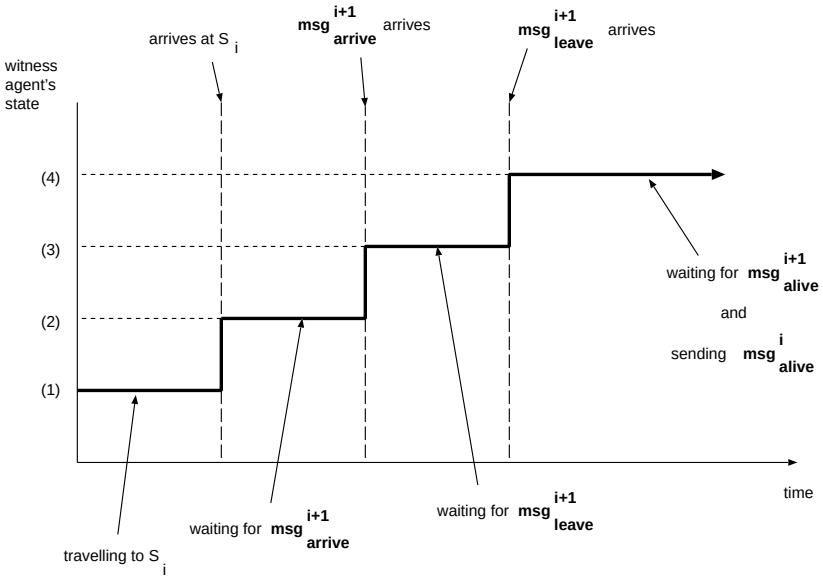

Figure 3.6: The life of a witness agent

Chapter 4

Fault-Tolerant Mechanism Analysis

In this chapter, we have a mathematical analysis over the whole level 2 fault-tolerant mechanism including the proof of the liveness of the mechanism and the possibility of the mechanism simplification. We define notations in the mechanism in our first section. We then define the fault-tolerant mechanism both formally and informally in order that we can have a correct and sound analysis over the whole mechanism or algorithm. Lastly, we will have liveness proofs of the algorithm as well as an analysis of the mechanism simplification.

4.1 Definitions and Notations

Every actual agent bears a unique identification number i. The witness agents and the probes that are monitoring the liveness of the actual agent i have the same agent ID number i. The system distinguishes these 3 kinds of agents by recognizing their types [1] .

An actual agent with agent ID i has an itinerary list $\mathscr{I}_{\mathbf{i}}$. $\mathscr{I}_i$ is a list of server names $S_0, \ \ldots \ S_{m-1}$, where m is the number of servers in the itinerary.

[1] In object-oriented programming language like JAVA, we use *classes* to distinguish them.

There are totally **n** servers in the system. A server S_j contains a processing unit $\mathscr{P}_j$ and a stable storage $\mathscr{S}_j$. We define the server failure of our system. We assume that there only exists stopping failures. Other failures such as Byzantine failure [14] do not exist. "A server failure of S_j" defines :

> $\mathscr{P}_j$ fails to advance in the computation of agents, and the storage $\mathscr{S}_j$ fails to operate.

This implies that the storage $\mathscr{S}_j$ does not fail if the processor $\mathscr{P}_j$ is working. On the other hand, when $\mathscr{P}_j$ fails, $\mathscr{S}_j$ should also fail. We further define that when a server failure occurs in S_j, all the agents inside S_j will be terminated.

The time of the system is measured in *rounds*. Every event in the system should last for an integer multiple of rounds. We assume that the processors in different servers are having the same computing speed. We further assume that the amount of computation needed for every agent at every processor is the same. We define the time constants for different events in the system.

- We denote the time needed for an actual agent to complete computation in server S_j be e_j rounds.
- The time needed for an agent to travel from S_i to S_j be a_{ij} rounds.
- The time needed for a message to travel from S_i to S_j be m_{ij} rounds.
- The time needed for a probe to recover an actual agent in S_i be r_{ai}.
- The time needed for a server monitor (Level 1) to inspect and recover S_i be r_{si}.

where $e_j,\ a_{ij},\ m_{ij},\ r_{ai},\ r_{si} \in \mathbb{N}$, and $e_j,\ a_{ij},\ m_{ij},\ r_{ai},\ r_{si} > 0$.

We also define different variable time periods in the system.

- Denote the time for server monitor to recover a failed server be $T_{recover}$.

- Denote the timeout of waiting for msg^i_{arrive} be T_{arrive}.
- Denote the timeout of waiting for msg^i_{leave} be T_{leave}.
- Denote the timeout of waiting for msg^i_{alive} be T_{alive}.
- Denote the period of the heartbeat message, msg^i_{alive}, be $T_{heartbeat}$

where $T_{arrive}, T_{leave}, T_{alive}, T_{heartbeat} \in \mathbb{N}$, and,
$T_{arrive}, T_{leave}, T_{alive}, T_{heartbeat} \geq 0$.

4.2 Assumptions

- For simplicity, we assume that the topology of the network is a complete graph. This implies that every agent and message can travel to every server in the system.
- We further assume that the number of rounds needed for message travel be unique throughout all the servers, i.e., $m_{ij} = m^*, \forall\, i, j \in \{0, 1, \ldots, n-1\}$, and $m^* \in \mathbb{N}$.
- The same token applies on agent travel, i.e., $a_{ij} = a^*, \forall\, i, j \in \{0, 1, \ldots, n-1\}$, and $a^* \in \mathbb{N}$.
- The above also applies on server recovery time, i.e., $r_{si} = r^*_s, \forall\ i \in \{0, 1, \ldots, n-1\}$, and $r^*_s \in \mathbb{N}$.
- Moreover, $r_{ai} = r^*_a, \forall\ i \in \{0, 1, \ldots, n-1\}$, and $r^*_a \in \mathbb{N}$
- There is no harm to have the above assumptions since we can assume that m^*, a^*, r^*_a and r^*_s are upper bounds of the required time.

4.3 The Algorithm

In this section, we define the level 2 fault-tolerant mechanism in details. This includes the algorithms of the actual agent, the witness agent, as well as the probe. We first describe the algorithms informally to let the readers to have an brief understanding of the algorithms. Following the informal descriptions of the algorithms, we state the formal algorithms. For generality, we introduce the following notations:

- During the actual agent is traveling through its itinerary, we label the actual agent that is residing or traveling to S_i be α_i.
- We let the witness agent that is residing or traveling to S_i be ω_i.

4.3.1 Informal algorithm descriptions

Actual Agent:

When an actual agent α_i arrives at S_i, where $i \in \{0, \ldots, n-1\}$, it logs the message log^i_{arrive} on $\mathscr{S}_i$. On the next round, it sends out msg^i_{arrive} to ω_{i-1} in S_{i-1}. It starts executing the required computations from the next round. After the execution has completed, it logs log^i_{leave} on $\mathscr{S}_i$. It sends another message msg^i_{leave} to S_{i-1} before leaving S_i. Eventually, it migrates to S_{i+1} on the next round.

Failure Handling:

1. Before α_i can migrate to S_{i+1}, α_i has to check if S_{i+1} is available or not.
2. If yes, α_i migrates.

3. If not, α_i waits until S_{i+1} is available again.

Witness Agent:

When a witness agent, ω_i, arrives at S_i, where $i \in \{0, \ldots, n-2\}$, it waits for the message msg_{arrive}^{i+1} for T_{arrive} rounds. If msg_{arrive}^{i+1} does not arrive after T_{arrive} rounds, a probe, ρ_{i+1}, will be sent to S_{i+1}. ω_i starts waiting for another T_{arrive} rounds.

ω_i starts waiting for msg_{leave}^{i+1} after it has received msg_{arrive}^{i+1} from S_{i+1}, and it waits for T_{leave} rounds. If msg_{leave}^{i+1} does not arrive after T_{leave} rounds, a probe, ρ_{i+1}, will be sent to S_{i+1}. ω_i starts waiting for another T_{leave} rounds.

After receiving msg_{leave}^{i+1}, ω_i starts waiting for msg_{alive}^{i+1} from S_{i+1}, and it waits for T_{alive} rounds. If msg_{alive}^{i+1} does not arrive after T_{alive} rounds, ω_{i+1} will be spawned and travels to S_{i+1}. On arrival, S_{i+1} will start sending msg_{alive}^{i+1} by pre-setting its internal state.

On the other hand, on arriving at S_i, ω_i starts sending msg_{alive}^{i} to ω_{i-1} with period $T_{heartbeat}$.

Failure Handling:

1. Before the newly spawned ω_i can migrate to S_i, ω_i has to check if S_i is available.
2. If yes, ω_i migrates.
3. If not, ω_i waits until S_i is available again.

Probe:

For each probe ρ_i, where $i \in \{0, \ldots, n-1\}$, depending on its internal state, it searches for either log^i_{arrive} or log^i_{leave} after it has arrived at S_i. If the search is successful, it re-transmits msg^i_{arrive} or msg^i_{leave} accordingly. If the search fails, it recovers α_i by using the checkpointed data from S_{i-1}.

Failure Handling:

1. Before the newly spawned ρ_i can migrate to S_i, ρ_i has to check if S_i is available.
2. If yes, ρ_i migrates.
3. If not, ρ_i waits until S_i is available again.

4.3.2 Formal algorithm descriptions

The description of every kind of processes, or agents, is divided in three parts, namely *states*, *msgs*, and *trans* (style in Lynch's book [16]). The *states* segment represents the internal states, or variables, of the process. Every state has its own *domain* as well as initial value. The *msgs* segment specifies when and what messages that the process will send. Lastly, the *trans* segment describes under what conditions that the internal states of the process will change.

Also, we adopt the previously defined timeout bounds, i.e., T_{arrive}, T_{leave}, T_{alive}, and $T_{heartbeat}$, in the formal algorithm description. The following descriptions are written in C language-like format.

Actual Agent:

states$_i$:

execute_rounds $\in \mathbb{N}$, initially 0.

previous_state $\in$ {*executing, send_message, migrating*}, *initially send_message.*

current_state $\in$ {*executing, send_message, migrating*}, *initially migrating.*

msgs$_i$:

if previous_state is migrating and current_state is executing, then

send msg^i_{arrive} *to* S_{i-1}

else if previous_state is executing and current_state is send_message, then

send msg^i_{leave} *to* S_{i-1}

end if

trans$_i$:

if current_state is send_message, then

previous_state := current_state

current_state := migrating

else if current_state is migrating, then

log the message log^i_{arrive}

previous_state := current_state

current_state := executing

execute_rounds := 0

start execute jobs.

else

if execute_rounds = e_i, then

log the message log^i_{leave}

previous_state := current_state

current_state := send_message

else

execute_rounds := execute_rounds + 1

end if

end if

if current_state is migrating, then

if S_{i+1} is available, then

migrate to S_{i+1}

end if

end if

Witness Agent:

states$_i$:

wait_arrive_rounds, wait_leave_rounds, wait_alive_rounds, heartbeat_rounds $\in \mathbb{N}$, *initially all are 0*

current_state $\in \{$*wait_arrive, wait_leave, spawn_witness, wait_alive*$\}$, *initially is wait_arrive*

send_heartbeat $\in \{$*true, false*$\}$, *initially is true*

msgs$_i$:

if send_heartbeat = true, then

send msg^i_{alive} *to* S_{i-1}

send_heartbeat := false

end if

trans$_i$:

message := get message from channel

if current_state is wait_arrive, then

if message is not null, then

current_state := wait_leave

else

if wait_arrive_rounds = T_{arrive}*, then*

create probe ρ_{i+1}

$\rho_{i+1} \rightarrow$ *current_state := search_arrive*

send ρ_{i+1} *to* S_{i+1}

wait_arrive_rounds := 0

else

wait_arrive_rounds := wait_arrive_rounds + 1

end if

end if

else if current_state is wait_leave, then

if message is not null, then

current_state := wait_alive

else

if wait_leave_rounds = T_{leave}, *then*

create probe ρ_{i+1}

$\rho_{i+1} \rightarrow$ *current_state := search_leave*

send ρ_{i+1} *to* S_{i+1}

wait_leave_rounds := 0

else

wait_leave_rounds := wait_leave_rounds + 1

end if

end if

else if current_state is spawn_witness, then

spawn ω_{i+1}

send out ω_{i+1} *to* S_{i+1}

current_state := wait_alive

else if current_state is wait_alive, then

if message is not null, then

wait_alive_rounds := 0

else

if wait_alive_rounds = T_{alive}, *then*

spawn ω_{i+1}

$\omega_{i+1} \rightarrow$ *current_state := wait_alive*

send out ω_{i+1} *to* S_{i+1}

wait_alive_rounds := 0

else

wait_alive_rounds := wait_alive_rounds + 1

end if

end if

end if

if heartbeat_rounds $= T_{heartbeat}$*, then*

send_heartbeat := true

heartbeat_rounds := 0

else

heartbeat_rounds := heartbeat_rounds + 1

end if

Probe:

states$_i$:

current_state $\in$ {*search_arrive, search_leave, send_arrive, send_leave, terminate*}*, initial value pre-set by* ω_{i-1}

msgs$_i$:

if current_state is send_arrive, then

send msg^i_{arrive} *to* S_{i-1}

else if current_state is send_leave, then

send msg^i_{leave} *to* S_{i-1}

end if

trans$_i$:

if current_state is search_arrive, then

if log^i_{arrive} *is not found, then*

current_state := terminate

recover α_i

$\alpha_i \rightarrow$ *previous_state := migrating*

$\alpha_i \rightarrow$ *current_state := executing*

else

current_state := send_arrive

end if

else if current_state is search_leave, then

if log^i_{leave} *is not found, then*

current_state := terminate

recover α_i

$\alpha_i \rightarrow$ *previous_state := executing*

$\alpha_i \rightarrow$ *current_state := executing*

else

current_state := send_leave

end if

else if current_state is send_arrive or current_state is send_leave, then

current_state := terminate

else

process termination

end if

4.4 Liveness Proof

In this section, we present the liveness proof of the proposed mechanism. We sketch the outline of the proof first. Inside the Section 4.1, we have defined several *time constants* as well as three distinct *variable time measurements*, namely T_{arrive}, T_{leave}, and T_{alive}. Our goal is to prove that the system will not be blocked forever under certain conditions. If the system is blocked forever, at least one of the above variable time measurements will reach *infinity*. Hence, the first few steps of our proof are aimed to derive the lower and upper bounds of those variable time measurements. Given that the itinerary of the agent is not infinite long, if the upper bounds of all variable time measurements are not approaching infinity, then the system should not be blocked forever.

Lemma 4.1 $r_s^* \leq T_{recover} \leq nr_s^*$

Proof.

This lemma is the analysis of the level 1 fault-tolerant mechanism. In the worst case, all servers are stopped, and the monitor starts inspecting and recovering the servers from S_{i+1}. Hence, the upper bound is $n * r_s^*$. The lower bound is trivial, i.e., r_s^*.

$$\therefore r_s^* \leq T_{recover} \leq nr_s^*$$

∎

Lemma 4.2 We define the lower bounds for various timeouts:

1. $T_{arrive} \geq 0$
2. $T_{leave} \geq e^*$
3. $T_{alive} \geq a^* + m^*$

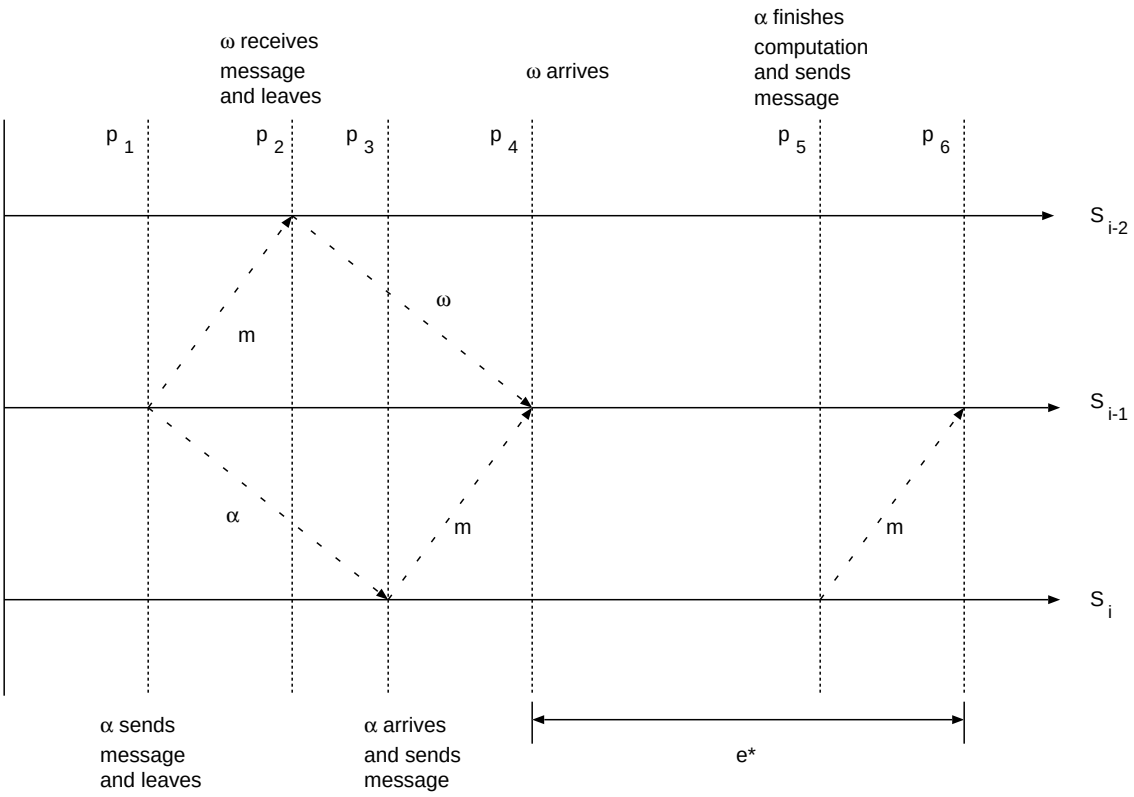

Figure 4.1: Minimum Time of T_{arrive} and T_{leave}

Proof.

1. Lower bound of T_{arrive}.

   Figure 4.1 shows the *time-space* diagram (by Lamport [17]) of the system. T_{arrive} is counting at the moment that ω_{i-1} arrives at S_{i-1}. When α_{i-1} sends msg^{i-1}_{leave}, on the next round, it migrates to S_i. When α_i arrives at S_i, it sends msg^{i}_{arrive} on the next round. Hence, it takes $a^* + e^* + 2$. On the other hand, ω_{i-1} also takes $a^* + e^* + 2$ rounds to travel from S_{i-2} to S_{i-1} (p_4 in Figure 4.1).

   $$\therefore\ T_{arrive} \geq 0$$

2. Lower bound of T_{leave}.

   The time between an actual agent executing its job and sending the leave message is the time e^* (between p_4 and p_6 in Figure 4.1).

   $$\therefore\ T_{leave} \geq e^*$$

3. Lower bound of T_{alive}.

   The time between the witness agent being created, spawning another witness agent, and receiving msg^i_{alive} is $a^* + m^*$.

   $$\therefore\ T_{alive} \geq a^* + m^*$$

■

Lemma 4.2 is an important lemma. It states the number of rounds that the witness agents have to wait without the presence of faults. Hence, inside the implementation of the witness agent, we can set the timeout periods to be those lower bounds. It is because we can assume that faults are rare events.

Definition 4.1 Let τ be the inter-arrival time of failures of S_i, $\forall i \in (0, 1, \ldots, n-1)$, and $\tau \in \mathbb{N}$.

Definition 4.2 Let $\mathbb{S}_f$ be a 1 x m vector, where

$$\mathbb{S}_f = \{(f_0, f_1, \ldots, f_{m-1}) : f_m \in \{0, 1, \ldots, n-1\}\ and\ m \in \mathbb{Z}^+ \cup \{0\}\},$$
and $||\mathbb{S}_f|| = m$

$\mathbb{S}_f$ defines a failure sequence with inter-arrival time τ. It implies, without loss of generality, S_{f_i} fails first, then, after τ rounds, $S_{f_{i+1}}$ fails.

Lemma 4.3 $a^* + e^* \leq \tau \leq \infty$ if the system is not blocked forever.

Proof.

It is trivial that the upper bound of τ is ∞, i.e., no failure. To prove the lower bound of τ, we require to calculate **(I)** the minimum rounds for α_i migrating to S_{i+1}, and **(II)** the minimum rounds for ω_i to spawn ω_{i+1}. First of all, we assume that there exists a failure sequence $\mathbb{S}'_f = (i, i, \dots)$ where $||\mathbb{S}'_f|| = \infty$, i.e., all failures happen only in S_i.

(I) $\because\ \mathbb{S}'_f = (i, i, \dots)\ \therefore$ there must be a moment of time that α_{i-1} is waiting for the recovery of S_i, and S_i is just recovered.

During α_{i-1} is migrating to S_i, there should be no failure happens otherwise the actual agent will be lost. The migrating of α_i takes a^* rounds. Also, the execution takes e^* rounds.

$$\therefore\ \tau \geq a^* + e^*$$

(II) It takes a^* rounds for ω_i can successfully migrate from S_{i-1} to S_i.

It takes another $min(T_{arrive}) + min(T_{leave})$ rounds for ω_{i+1} can successfully migrate from S_i to S_{i+1}.

$$\because\ T_{arrive} \geq 0 \ \text{ and } \ T_{leave} \geq e^*$$
$$\therefore\ \tau \geq a^* + e^*$$

From (I) and (II), we conclude that:

$$\tau \geq a^* + e^*$$

∎

Corollary 4.1 It is impossible for α to complete its itinerary if $\tau < a^* + e^*$.

Proof.

This corollary follows from Lemma 4.3. ∎

Assertion 4.1 $r_a^* < e^*$

Assertion 4.1 guarantees that the time of the agent recovery should be shorter than the agent execution time The time needed to have an agent recovered is the migrating time for the probe plus the agent recovery time, i.e., $a^* + r_a^*$. We do not desire to have an incomplete recovery. Hence, $a^* + r_a^* < \tau$ must hold. If $r_a^* = e^*$, there is a chance of an incomplete recovery by Lemma 4.3. Therefore, it would be nice to have $r_a^* < e^*$. It is also a reasonable assertion because the agent recovery should not be as time consuming as the agent execution.

Lemma 4.4

$$0 \leq \ T_{arrive} \ \leq nr_s^* + a^* + r_a^*$$

Proof.

We use induction to proof the upper bound of T_{arrive}.
Let $T^{max}_{arrive(k)}$ be the upper bound of T_{arrive} where, $||\mathbb{S}_f|| = k$.

- If $k = 0$, from Lemma 4.2, it is trivial that $T^{max}_{arrive(0)} = 0$.
- If $k = 1$,

  Suppose that a failure strikes S_i at the round that ω_{i-1} arrives at S_{i-1}.

  At the same round, the timer of T_{arrive} will start counting.

  $\because T_{arrive} \geq 0$, we choose the time that ω_{i-1} should wait be 0.

  $\because$ A failure happens at S_i, T_{arrive}, will be reached momentarily.

  However, ω_{i-1} is not required to wait for $T_{recover}$ rounds. Instead, it should be $T_{recover} - m^*$ rounds. Hence, $T^{max}_{arrive(1)}$ will the sum of $T_{recover} - m^*$, the agent traveling time, the agent recovery time, and the message

traveling time.

$$\begin{aligned} \therefore T^{max}_{arrive(1)} &= (T_{recover} - m^*) + a^* + r^*_a + m^* \\ &= T_{recover} + a^* + r^*_a \end{aligned}$$

- If $k = k'$,

  (I) Let the failure sequence be $\mathbb{S}'_f = (i, i, \ldots, i)$ where $||\mathbb{S}'_f|| = k'$.

  After the first failure is recovered, the time when the next failure happen is τ. However, the time when the ρ_i reaches S_i and recovers α_i is $a^* + r^*_a$.

  $\therefore$ If the second and further failures can affect the upper bound of $T^{max}_{arrive(k')} \Rightarrow \tau \leq a^* + r^*_a$.

  $\because a^* + e^* \leq \tau \ \Rightarrow e^* \leq r^*_a \ \Rightarrow$ Contradiction with Assertion 4.1.

  $\therefore$ We can conclude that only the first failure can affect $T^{max}_{arrive(k')}$.

$$\begin{aligned} \therefore 0 \leq T^{max}_{arrive(k')} &\leq T_{recover} + a^* + r^*_a \\ &\leq nr^* + a^* + r^*_a \end{aligned}$$

  (II) Let the failure sequence be $\mathbb{S}''_f$ where $\mathbb{S}''_f \neq \mathbb{S}'_f$.

  Other failures not happening in S_i cannot affect T_{arrive} of ω_{i-1}. It is because if the failure happens on S_{i-1}, i.e., ω_{i-1} will be terminated, T_{arrive} counting of ω_{i-1} will be discarded. The only way that can extend T_{arrive} is the failures that terminate α_i.

  $\therefore$ Only consecutive failures happening on S_i can affect T_{arrive} of ω_{i-1}.

■

Lemma 4.5

$$e^* \leq T_{leave} < k(nr_s^* + a^* + r_a^*) + (k-1)e^* + 2m^*$$

where k is the number of failures, $k \in \mathbb{N}$, and $a^* + e^* \leq \tau < a^* + e^* + r_a^*$

Proof.

We also use induction to proof the upper bound of T_{leave}.
Let $T^{max}_{leave(k)}$ be the upper bound of T_{leave} where, $||\mathbb{S}_f|| = k$.

- If $k = 0$, from Lemma 4.2, it is trivial that $T^{max}_{leave(0)} = e^*$.
- If $k = 1$,

  Suppose that a failure strikes S_i after α_i sends out msg^i_{arrive} and before α_i sends out msg^i_{leave}. We have 2 cases here, either the computation has finished or it has not finished. Since we only have 1 failure, we can treat these 2 cases as 1. Since we are estimating the upper bound, we assume that the failure happens when the computation is about to be finished, i.e., at least 1 round is remaining.

  $\because$ At the moment that S_i crushes, ω_{i-1} is waiting for msg^i_{leave} for $e^* - m^* - 1$ rounds.

  $\therefore$ S_i is recovered after $T_{recover} - (e^* - m^*) - 1$ from the view point of ω_{i-1}.

$$\begin{aligned} \therefore T^{max}_{leave(1)} &= T_{recover} - (e^* - m^*) - 1 + a^* + r_a^* + e^* + m^* \\ &= T_{recover} + a^* + r_a^* + 2m^* - 1 \\ &< nr_s^* + a^* + r_a^* + 2m^* \end{aligned}$$

- If $k = k'$,

(I) Let the failure sequence be $\mathbb{S}'_f = (i, i, \ldots, i)$ where $||\mathbb{S}'_f|| = k'$.

After the first failure is recovered, the time when the next failure happen is τ. The time when ρ_i reaches S_i, recovers α_i, and the recovered α_i sends msg^i_{leave} is $a^* + r^*_a + e^*$.

$\therefore$ If the second and further failures can affect $T^{max}_{arrive(k')}$,

$\Rightarrow \ \tau \ < \ a^* + r^*_a + e^*$ must hold.

$\because a^* + e^* \leq \tau < a^* + r^*_a + e^* \ \ \therefore r^*_a > 0$, which is always true.

$\therefore$ We can conclude that the failure sequence $\mathbb{S}'_f$, if $a^* + e^* \ \leq \ \tau \ < \ a^* + r^*_a + e^*$, the failure sequence $\mathbb{S}'_f$ will always prohibit the computation from advancing.

$$\begin{aligned}
\therefore T^{max}_{leave(k')} &= T_{recover} - (e^* - m^*) - 1 + (a^* + r^*_a + e^* - 1) \\
&\quad + (T_{recover} + a^* + r^*_a + e^* - 1) + \cdots + m^* \\
&= k'(T_{recover} + a^* + r^*_a + e^* - 1) + 2m^* - e^* \\
&< k'(nr^*_s + a^* + r^*_a) + (k' - 1)e^* + 2m^*
\end{aligned}$$

(II) Using similar argument in Lemma 4.4, if we have a different failure sequence, the estimation of the upper bound of T_{leave} is still the same.

■

Lemma 4.6

$$max \begin{pmatrix} a^* + m^* \\ T_{heartbeat} \end{pmatrix} \leq \ T_{alive} \ < min \begin{pmatrix} nr^*_s + 2a^* + 2m^* \\ nr^*_s + a^* + e^* \end{pmatrix}$$

Proof.

Let $T^{max}_{alive(k)}$ be the upper bound of T_{alive} where, $||\mathbb{S}_f|| = k$.

- If $k = 0$, from Lemma 4.2, $T^{max}_{alive(0)} = a^* + m^*$.

  However, msg^i_{alive} is a series of periodic messages. The time between two successive messages is $T_{heartbeat}$. Hence,

  $$T^{max}_{alive(0)} = min(a^* + m^*,\ T_{heartbeat})$$

- If $k = 1$,

  In order to calculate the $T^{max}_{alive(1)}$, we have to choose the moment of failure which prohibit ω_i from receiving msg^{i+1}_{alive} for the longest duration.

  $\therefore$ The right moment should be just before the message is sent.

  $$\begin{aligned} \therefore T^{max}_{alive(1)} &= a^* + m^* - 1 + T_{recover} + a^* + m^* \\ &= T_{recover} + 2a^* + 2m^* - 1 \\ &< nr^*_s + 2a^* + 2m^* \end{aligned} \tag{4.1}$$

- If $k = k'$,

  (I) Let the failure sequence be $\mathbb{S}'_f = (i+1, i+1, \ldots, i+1)$ where $||\mathbb{S}'_f|| = k'$.

  If the second failure can affect $T^{max}_{alive(k')}$, then $\tau < a^*$

  $\because a^* + e^* \leq \tau < a^* \quad \therefore e^* < 0$. Hence, the result is a contradiction.

  $\therefore$ The second failure will not affect the upper bound of the timeout.

  However, msg^{i+1}_{alive} is periodic, further failures might affect $T^{max}_{alive(k')}$.

  $\therefore$ How the failures affect the timeout bound depends on $T_{heartbeat}$.

  If the next failure can affect $T^{max}_{alive(k')}$, then $\tau < a^* + T_{heartbeat}$

  $\therefore T_{heartbeat} > e^*$ in order to have a longer $T^{max}_{alive(k')}$. (*)

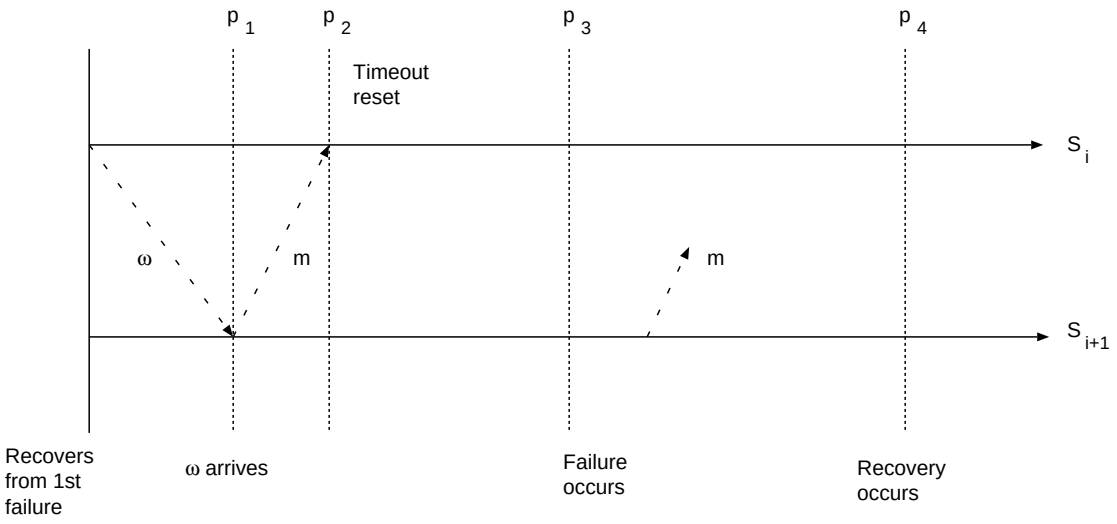

Figure 4.2: $T_{heartbeat} > e^*$

- If $T_{heartbeat} > e^*$,

Figure 4.2 shows corresponding *time-space* diagram. In this case, no msg_{alive}^{i+1} will be sent since another failure happens before msg_{alive}^{i+1} is sent.

$\therefore T_{alive(k')}^{max}$ is the time between p_2 and p_4 plus $a^* + m^*$.

$$\begin{aligned} \therefore T_{alive(k')}^{max} &= T_{recover} + (e^* - m^*) + a^* + m^* \\ &= T_{recover} + a^* + e^* \\ &\leq nr_s^* + a^* + e^* \end{aligned} \tag{4.2}$$

- If $T_{heartbeat} \leq e^*$,

Figure 4.3 shows the *time-space* diagram of this scenario. At p_2, $T_{alive(1)}^{max}$ is determined. At p_3, a failure happens. But, msg_{alive}^{i+1} can be sent before the failure happens. Hence, the timeout of ω_i will be reset at p_4.

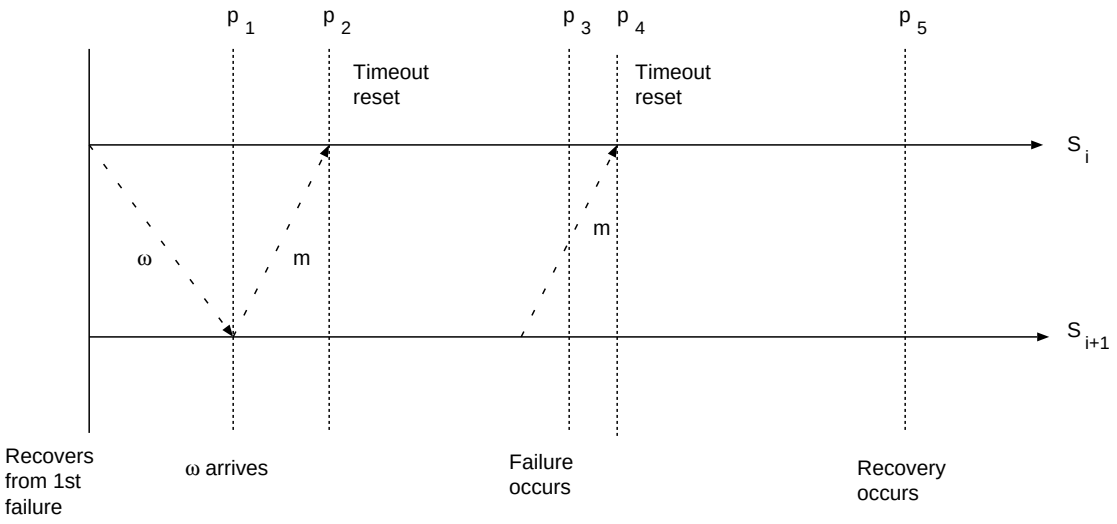

Figure 4.3: $T_{heartbeat} \leq e^*$

$\therefore T^{max}_{alive(k')}$ is the time between p_4 and p_5 plus $a^* + m^*$.

$$\begin{aligned} \therefore T^{max}_{alive(k')} &= T_{recover} - (e^* - T_{heartbeat}) + a^* + m^* \\ &< nr^*_s + a^* + m^* \end{aligned} \tag{4.3}$$

From (4.1), (4.2), and (4.3),

$$T_{alive} < min \begin{pmatrix} nr^*_s + 2a^* + 2m^* \\ nr^*_s + a^* + e^* \end{pmatrix}$$

We assert that the logic argument (*) is correct. Therefore, we have

$$\begin{aligned} nr^*_s + a^* + e^* &< nr^*_s + a^* + m^* \\ \Rightarrow \; e^* &< m^* \end{aligned}$$

(II) Other failures not happening in S_{i+1} can only affect T_{alive} time of witness agents other than ω_i. It can only also affect T_{arrive} and T_{leave} if the failures have terminated α.

$\therefore$ Only consecutive failures happening on S_{i+1} can affect T_{alive} of ω_i, and it is handled in previous cases.

■

Corollary 4.2

$$0 < T_{heartbeat} \leq e^*$$

Proof.

This result follows from the proof of Lemma 4.6.

$\because$ Choosing $T_{heartbeat} \leq e^*$ can mask one failure, and have a shorter T_{alive}.

■

Corollary 4.3

$$max(a^* + m^*,\ e^*) \leq\ T_{alive}\ < nr_s^* + 2a^* + 2m^*$$

Proof.

This result follows from Corollary 4.2 and Lemma 4.6.

■

After defining and proving several assertions, definitions, and lemmas, we have enough knowledge to prove the liveness of the system.

Theorem 4.1 The system is blocked iff $\mathbb{S}_f = (i, i, \dots)$ and $a^* + e^* \leq \tau < a^* + e^* + r_a^*$, where $||\mathbb{S}_f|| = \infty$, and $i \in (0, 1, \dots, n-1)$.

Proof.

"$\Rightarrow$" We are making use of Lemma 4.5 and its proof.

$\because$ The system is blocked $\therefore$ One of the timeouts must $\rightarrow \infty$.

From Lemma 4.1, 4.4, 4.5 and 4.6, only the upper bound of T_{leave} is proportionally increasing with the number of failures.

From the proof of Lemma 4.5, all the consecutive failures must be happening on the same server with maximum inter-arrival time $a^* + e^* + r_a^*$.

Moreover, as $T_{leave} \to \infty$, $k \to \infty$.

$$\therefore \mathbb{S}_f = (i, i, \dots) \text{ and } a^* + e^* \leq \tau < a^* + e^* + r_a^*$$

where $||\mathbb{S}_f|| = \infty$, and $i \in (0, 1, \dots, n-1)$.

"$\Leftarrow$" We are making use of Lemma 4.5 again.

From Lemma 4.5, $k \to \infty \Rightarrow T_{leave} \to \infty$.

$\because T_{leave} \to \infty \Rightarrow \alpha_i$ never finishes computation in S_i as infinite failures are happening on S_i.

$\therefore$ The system is blocked.

∎

Theorem 4.1 states that the system can still be blocked conditioning on the inter-arrival time of failures of a server. We can estimate the probability that the conditions will happen as follow.

Definition 4.3 Let $N_i(t)$ be a counting process such that, at time t, there are $N_i(t)$ failures happened in S_i. Let $T_{k(i)}$ denote the elapsed time between the $(k-1)^{st}$ and the k^{th} failure at S_i. We let the failure inter-arrival time distribution be an exponential distribution. Hence,

$$P\{T_{k(i)} > t \mid T_{k-1(i)} = s\} = \begin{cases} 1 - e^{-\lambda_i t} & if\ t > a^* + e^* \\ 0 & otherwise \end{cases}$$

where λ_i is the mean.

Definition 4.3 states that the server failure inter-arrival distribution is a *conditional exponential distribution* (see Figure 4.4). If the time is less than $a^* + e^*$, the probability is zero. Otherwise, the probability distribution is

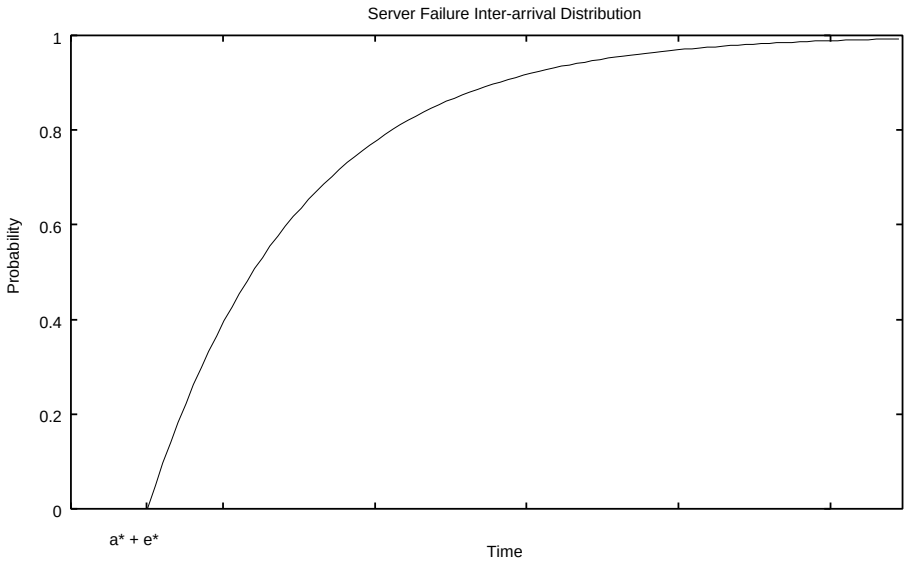

Figure 4.4: Server Failure Inter-arrival Distribution.

exponential. This follows from Lemma 4.3 since Lemma 4.3 states that $\tau > a^* + e^*$ in order that the system will not be block forever. Hence,

$$\begin{aligned} P\{a^* + e^* \leq \tau < a^* + e^* + r_a^*\} &= P\{T_{k(i)} > a^* + e^* + r_a^* \mid T_{k-1(i)} = s\} \\ &\quad - P\{T_{k(i)} > a^* + e^* \mid T_{k-1(i)} = s\} \\ &= e^{-\lambda_i(a^*+e^*)}(1 - e^{-\lambda_i r_a^*}) \end{aligned}$$

4.5 Simplification Analysis

In this section, we analyze the conditions for the successful deployment of the simplification of the level 2 fault detection and recovery mechanism. In Section 3.4, we have discussed logically that if the inter-arrival time between two failures is long enough, two witness agents are sufficient to monitor the actual agent. We want to analyze the lower bound of the failure inter-arrival time. This failure inter-arrival time is not τ in Definition 4.1. τ is the failure inter-arrival time of one server. We are now interested in the failure inter-arrival time throughout the system. Figure 4.5 shows what the system failure

arrival is. It is the sum of the arrivals of each server in the system. The total arrivals (bottom axis in Figure 4.5) shows the same pattern as the failure sequence $\mathbb{S}_f$ defined in Definition 4.2, i.e., $\mathbb{S}_f = (i-1,\ i,\ i-1,\ i+1,\ i-1,\ i+1,\ i,\ i-1)$.

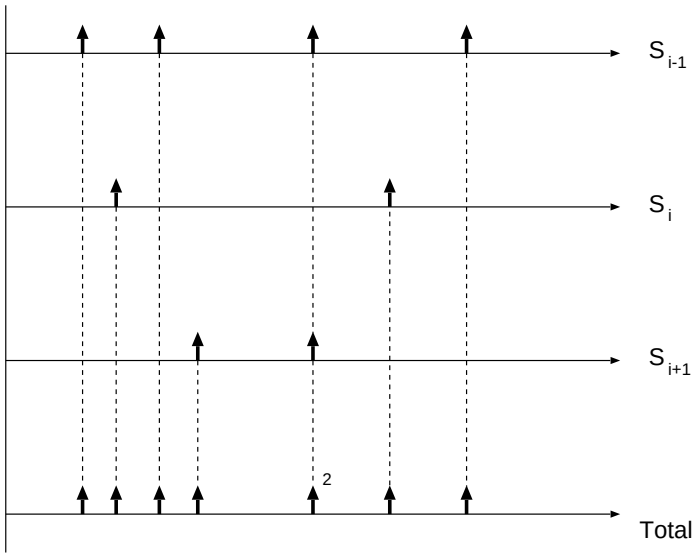

Figure 4.5: The system failure arrivals.

Definition 4.4 Define $\mathscr{T}$ be the inter-arrival time of the failures throughout the system, $\mathscr{T} \in \mathbb{N}$

Definition 4.4 defines the failure inter-arrival time of the system. The system failure arrival composes of the failure arrivals of every server in the system. Hence, $\mathscr{T} \geq 0$ should hold because there are chances that 2 servers failures at the same time. Also, it is obvious that $\mathscr{T} < \infty$ since there can be no failures in the system. We analyze the lower bound of $\mathscr{T}$ in the following lemma.

Lemma 4.7 If two witness agents are sufficient to maintain the liveness of the system, then

$$\mathscr{T} > max \begin{pmatrix} a^* + e^*, \\ 2a^* + m^*, \\ T_{recover} + a^* + m^* \end{pmatrix}$$

Proof.

This Lemma is the analysis of the simplification of Level 2 fault-tolerant mechanism (see Section 3.4).

Since 2 witness agents are sufficient, the required witness agents should be, without loss of generality, ω_{i-2} and ω_{i-1} with α_i in S_i (see Figure 4.6).

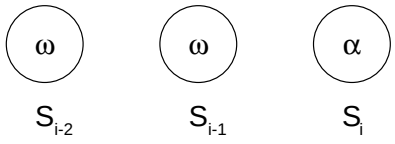

Figure 4.6: System configuration with 2 witness agents only.

We first analyze the case that ω_{i-1} is terminated (part (a)). Since the middle witness agent is lost, the race between the recovery of ω_{i-1} and the termination of ω_{i-2} arises. If the termination is faster, then, without any witness agents, the system is in a dangerous state. We will calculate when the failure should happen in order that two witness agents are sufficient.

Then, we analyze the case that the last agent is terminated (part(b)). Since the remaining witness agent may be terminated soon, we will analyze when the next failure should come.

Finally, we analyze the case that α_i is terminated (part (c)). It becomes the races between the recovery of α_i and the termination of ω_{i-1}. If ω_{i-1} is terminated, ω_{i-2} becomes the only surviving witness agent and it takes the responsibility of recovery ω_{i-1}.

(a) Let $\mathbb{S}_f^{(1)} = (i-1,\ i-2,\ i,\ i-1, \ldots)$, where $||\mathbb{S}_f^{(1)}|| = \infty$.

The failure sequence $\mathbb{S}_f^{(1)}$ will first disable ω_{i-1}, then ω_{i-2}. If $\mathscr{T}$ is small enough, α_i will also be terminated.

According to Figure 4.7, at p_1, α_i finishes its computation, and then it sends leave message to ω_{i-1}. At p_2, a failure strikes S_{i-1}. It depends on whether both the terminating message and spawned ω_i have been transmitted or not.

(I) If both the terminating message and spawned ω_i are transmitted successfully, then there will be only ω_i and α_{i+1} left in the system because the terminating message from ω_{i-1} will terminate ω_{i-2}.

According to $\mathbb{S}_f^{(1)}$, the next failure will happen in S_i at time p_7. Hence, $\mathscr{T} > a^* + e^*$ must hold in order that ω_i can have enough time to spawn and send ω_{i+1} to S_{i+1}. Otherwise, there will be no witness agents left in the system. Therefore, we have

$$\mathscr{T} > a^* + e^* \tag{4.4}$$

(II) If both the terminating message and spawned ω_i are terminated by failure, ω_{i-2} will still survive. We assume that msg_{alive}^{i-1} is sent at time one round before the failure happened on S_{i-1} (one round before p_2 in Figure 4.8). Hence, ω_{i-2} has to wait for $max(T_{recover} -$

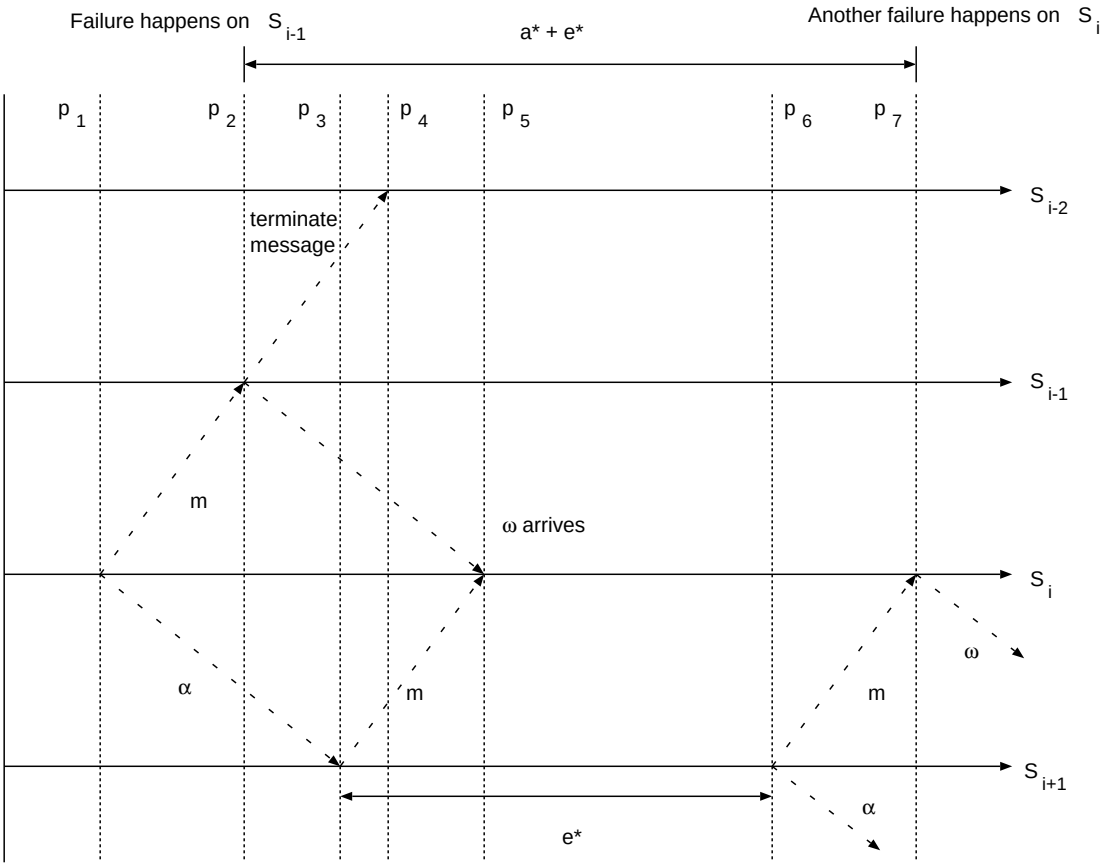

Figure 4.7: ω_i and terminating message are sent before failure happens.

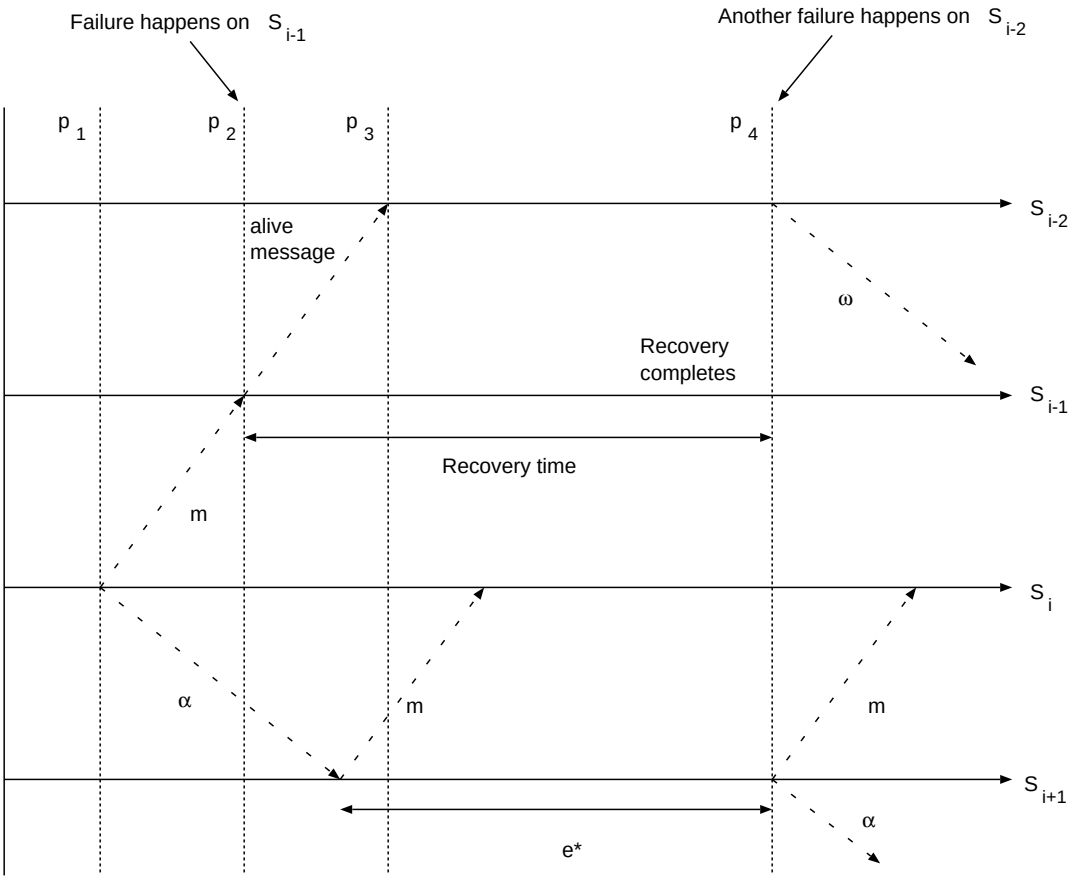

Figure 4.8: Failure happens before ω_i and terminating message are sent.

$m^*,\ min(T_{alive})$) rounds before ω_{i-2} recovers ω_{i-1}.

We let $\mathbb{S}_f^{(2)} = (i-1,\ i-2,\ i-1, \dots)$, where $||\mathbb{S}_f^{(2)}|| = \infty$. Together with Corollary 4.3,

$$\mathscr{T} \ > \ max(T_{recover} - m^*,\ a^* + m^*,\ e^*) \tag{4.5}$$

must hold in order that ω_i can survive.

Furthermore, the next failure may terminate ω_{i-1} again. Hence,

$$\therefore \mathscr{T} > a^* + min(T_{alive}) \Rightarrow \mathscr{T} > a^* + max(a^* + m^*,\ e^*) \tag{4.6}$$

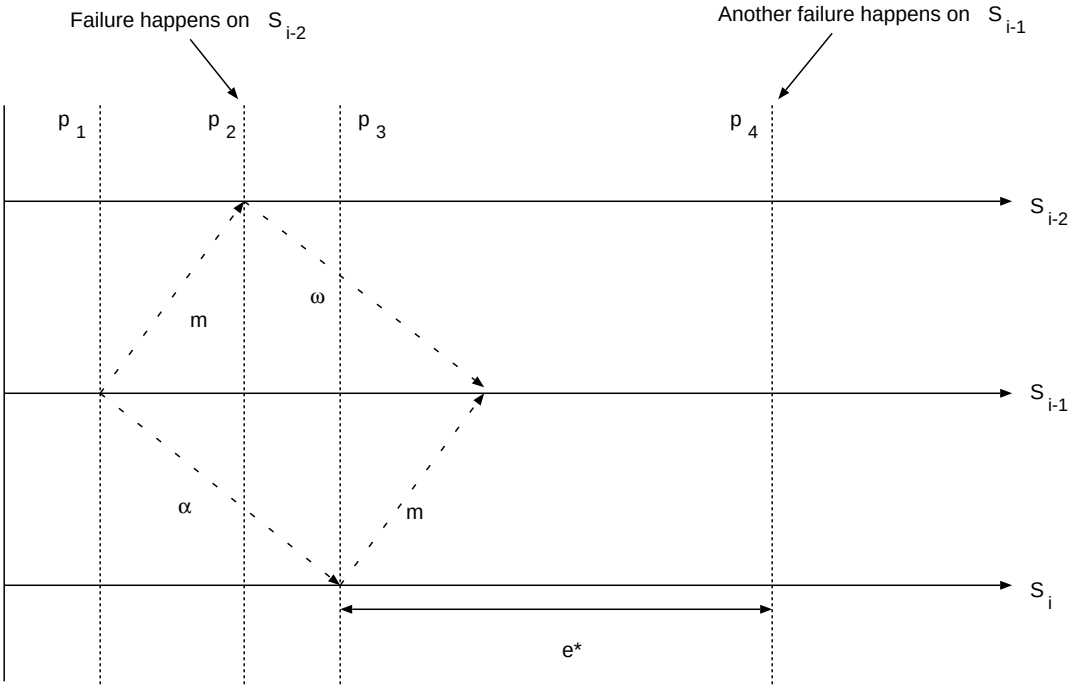

Figure 4.9: Failure happens when only the closet witness agent remains.

(b) Let $\mathbb{S}_f^{(3)} = (i-2, i-1, i, \dots)$, where $||\mathbb{S}_f^{(3)}|| = \infty$.

From Figure 4.9, the required time between the first two failures should be the difference between p_2 and p_4. Hence,

$$\mathscr{T} > e^* + a^* - m^* \tag{4.7}$$

(c) Let $\mathbb{S}_f^{(3)} = (i, i-1, i-2, \dots)$, where $||\mathbb{S}_f^{(3)}|| = \infty$.

Failure on S_i may terminate α_i. The next failure arrival time should be after ω_{i-1} sending out ρ_i. Otherwise, the recovery would be missed.

$\therefore$ In this case, the first failure can happen before **(i)** msg^i_{arrive}, or **(ii)** msg^i_{leave} is sent.

$\Rightarrow$ **(i)** $\mathscr{T} > T_{arrive} - a^* - m^* - r_a^*$, or **(ii)** $\mathscr{T} > T_{leave} - a^* - m^* - r_a^*$.

$\Rightarrow \mathscr{T} > T_{recover} - m^*$, or $\mathscr{T} > T_{recover} + m^*$

$\because$ We have to choose a larger time to guarantee that the recovery can proceed.

$$\therefore \mathscr{T} > T_{recover} + m^* \tag{4.8}$$

On the other hand, the second failure disables ω_{i-1}. ω_{i-2} will be responsible to recover ω_{i-1}.

In this scenario, ω_{i-2} has to recover ω_{i-1} in order to recover α_i eventually.

The lower bound of $\mathscr{T}$ will be: $\mathscr{T} > T_{alive} - a^* - m^*$

$\Rightarrow \mathscr{T} > T_{recover} + a^* + m^*$

$\therefore \mathscr{T} > T_{recover} + m^*$ must hold in order that a witness agent can recover an actual agent.

On the other hand, $\mathscr{T} > T_{recover} + a^* + m^*$ must hold in order that a witness agent can recover another witness agent.

$$\therefore \mathscr{T} > T_{recover} + a^* + m^* \tag{4.9}$$

$\therefore$ We conclude the maximum value of the minimum bound by equations (4.4) to (4.9)

$$
\begin{aligned}
\mathscr{T} \;&>\; max \left(\begin{array}{l} a^* + e^*, \\ max(T_{recover} - m^*,\ a^* + m^*,\ e^*), \\ a^* + e^* - m^*, \\ a^* + max(a^* + m^*,\ e^*), \\ T_{recover} + m^*, \\ T_{recover} + a^* + m^* \end{array} \right) \\
&\Rightarrow\; max \left(\begin{array}{l} a^* + e^*, \\ T_{recover} - m^*, \\ a^* + m^*, \\ e^*, \\ 2a^* + m^*, \\ T_{recover} + m^*, \\ T_{recover} + a^* + m^* \end{array} \right) \\
&\Rightarrow\; max \left(\begin{array}{l} a^* + e^*, \\ 2a^* + m^*, \\ T_{recover} + a^* + m^* \end{array} \right)
\end{aligned}
$$

■

Chapter 5

Link Failure Analysis

In this chapter, we discuss the issues of link failure. In the first section, we define what link failure is. Moreover, we address the problems raised from link failures. We propose partial solutions to remedy the problems of link failure in the next section. It is an extension of the level 2 fault-tolerant mechanism. We discuss how the proposed solutions can cooperate with the level 2 fault-tolerant mechanism. We name the modified mechanism the *level 3 fault-tolerant mechanism.*

5.1 Problems of Link Failure

When a link failure happens, say the link between the servers S_i and S_j is broken, there will not be messages nor agents that can travel from S_i to S_j, and vice versa. We cannot nor recover a link failure, but we can detect it. In order to tackle this problem, first, we have to assume that the link failure will be recovered eventually. In other words, the link failure lasts for an arbitrary length of time, but not forever. Otherwise, the agent will never reach the target server nor return to the destination (or the *home* server).

In our model, although there can be many routes going from one server to another, we abstract the routes into a single link. A link failure represents the un-availability of a link between two servers, say S_u and S_v (we name such an

edge (S_u, S_v)). That implies all the routes between S_u and S_v are disabled. Therefore, if an agent at server S_u wants to travel to S_v, it will stop advancing to S_v and waits at S_u until the link is enabled again. Fortunately, a link failure does not mean that S_v is not reachable. There can be other *paths* from S_u to S_v.

Network partitioning is a disastrous consequence of link failures. Inside a network graph, there are edges called cut edges. The failures of those edges will separate the graph into disconnected partitions. This implies there are chances that the agent will be trapped inside one of these partitions. If all the unvisited servers, the destination and the agent are on the same partition, the agent can still complete its itinerary. However, if the destination or some unvisited servers are in different partitions, it is impossible for the agent to reach the remaining servers on its itinerary until the failure of cut edges is recovered.

5.2 Solution

In this section, we discuss some partial solutions to ease the problems of link failure. In a mobile agent system, every agent has its itinerary which is pre-assigned in the *home server*. Suppose the agent is in S_u and its next server is S_v. When the edge (S_u, S_v) fails, this leads three scenarios. The three scenarios depend on the position of the actual agent when the failure happens. The three different scenarios result in different consequences based on the *level 2 fault-tolerant mechanism*.

1. Link failure occurs before the agent starts traveling to S_v;

   Consequence: the agent cannot proceed so it it waits in S_u until the edge (S_u, S_v) recovers.

2. Link failure occurs while the agent is on the way to S_v;

Consequence: the agent is lost (or only parts of the agent can arrive at S_v. Instead of treating the partial agent as a valid one, we treat the agent is lost in the network). A proper recovery of the agent should take place.

3. Link failure occurs after the agent has arrived at v.

   Consequence: we assume that we are imposing the level 2 fault-tolerant mechanism. The messages sending between S_u and S_v will not be able to reach their destinations deal to the link failure. Hence, the level 2 fault-tolerant mechanism will fail. However, the actual agent is still available. There may be chances that the actual agent can successfully reach the destination without the witness agents.

We discuss the mechanisms for tackling these three scenarios in the following subsections.

Scenario 1 - before the agent travels to S_v

In this case, the agent stops advancing and is caught in S_u. Instead of waiting for the recovery, it can travel to another unvisited server, say $S_{v'}$ in its itinerary list. The decision on whether traveling to $S_{v'}$ or waiting for the link recovery in S_u is based on the number of trials in detecting the availability of the target S_v. If the number of trials is beyond a pre-defined threshold, the agent gives up traveling to S_v and will instead travel to $S_{v'}$. The determination of the threshold is application dependent.

If the edge (S_u, S_v) is not a cut-edge, the actual agent can eventually travel to S_u without the recovery of (S_u, S_v) by the above mechanism. However, the actual agent may need to know the topology or routing information of the network in order to make an appropriate choice of $S_{v'}$. If the information is available, the process of choosing v' can be more efficient, and an alternative route can be determined for the actual agent migrates to v eventually. Figure

5.1 illustrates this approach. Unfortunately, the routing information of the whole network is usually not easy to be retrieved. More importantly, the routing information may change after the agent has gathered it. Nevertheless, if there is no unvisited servers in the same partition, the agent can only wait for the recovery of the cut edge.

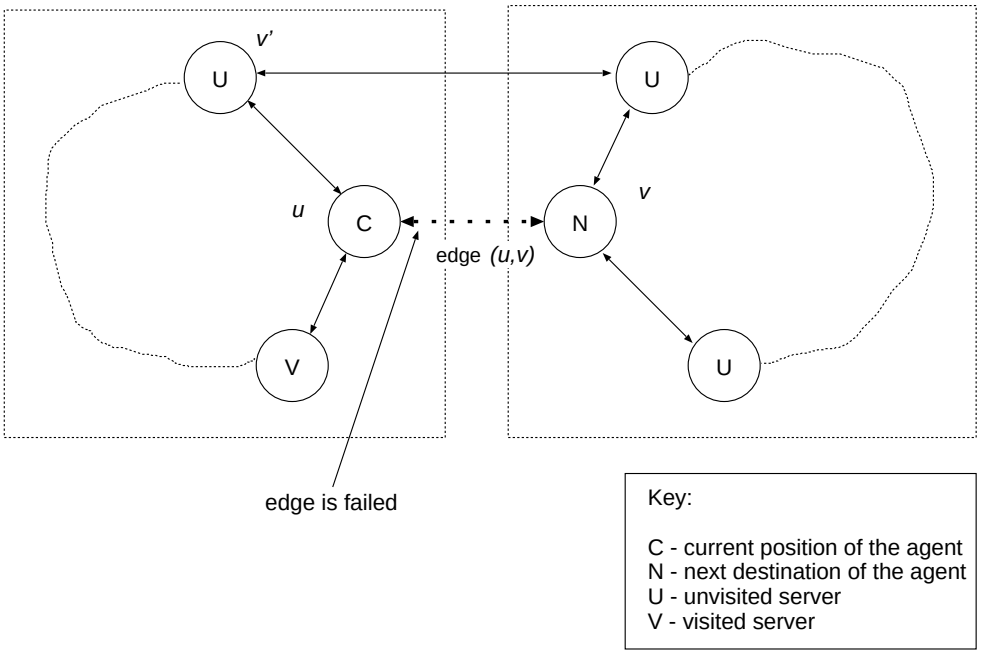

Figure 5.1: Choosing a suitable $S_{v'}$ is important.

Scenario 2 - when the agent is traveling to S_v

When the link failure happens as the agent is traveling, the agent is assumed to be lost. Since the actual agent fails to migrate to S_v, there will be no msg^v_{arrive} sending towards the witness agent in S_u, i.e. ω_u. Eventually, after the link is recovered, the actual agent will be recovered in S_v. In this scenario, one possible design is to allow the witness agent to recover the actual agent in another server, say $S_{v'}$. Such an option can increase the efficiency of the protocol. However, as the witness agent cannot guarantee whether the actual

agent has survived in the link failure or not, the witness agent cannot and should not make the decision to recover the execution of the actual agent at another server. It must wait for the link recovery.

Scenario 3 - after the agent has traveled to S_v

In this scenario, the level 2 fault-tolerant mechanism still works, but it may become less efficient. When the actual agent is in S_v, two messages, which are msg^v_{arrive} and msg^v_{leave} will be sent towards u. However, since the link is broken, the messages cannot reach S_u. Instead of waiting for the successful message transmissions, the actual agent keeps on advancing. When the actual agent resides in a server, it leaves *indirect messages* there for the witness agents (because there is no witness agents receiving those messages). The actual agent stops traveling until it either reaches the destination or is terminated by a server failure.

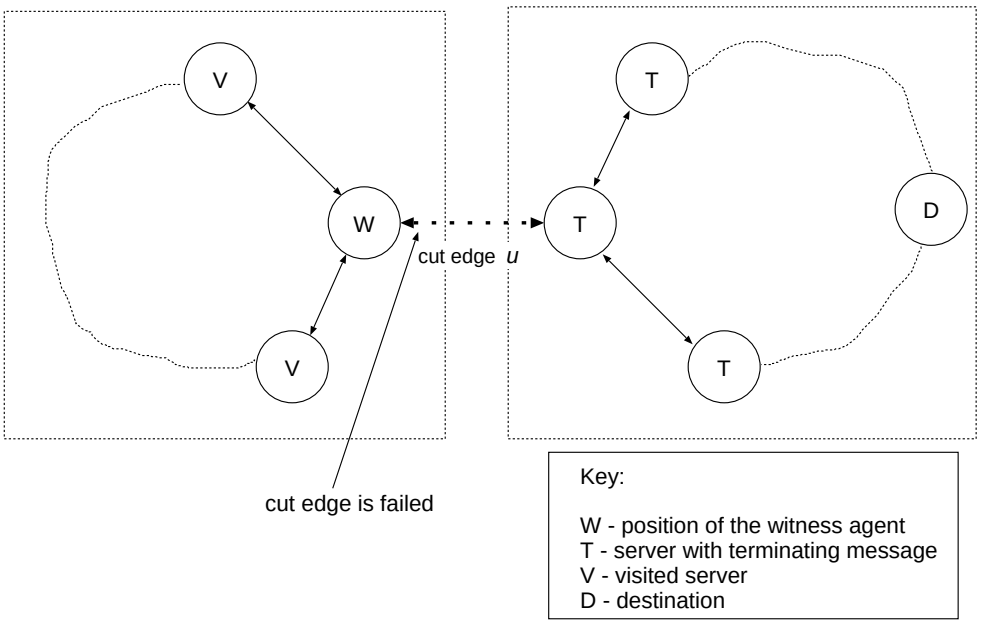

Figure 5.2: Terminating message waits for link recovery

On the other hand, ω_u keeps on trying to send probes to S_v. When it succeeds as the link is recovered, the probe will re-transmit the expected messages by using the log messages in S_v. Then, the process goes on until:

1. the probe finds that the actual agent is lost at one of the servers.

2. the witness agent reaches the destination.

In the first case, since the actual agent has left indirect messages along its itinerary, the witness agents can use these messages to catch up until it reaches the server where the actual agent is terminated. Eventually, the probe starts the recovery process. In the second case, it may be inefficient if the witness agents are not terminated until they reach the last server of the itinerary. It would be more efficient if we send terminating signal through the itinerary of the actual agent when the actual agent reaches the destination. The terminating signal is just another log message, denoted as it log_{term}. When a witness agent finds the log_{term} message inside a server, it will be terminated. Although the link failure will also block the terminating message, when the link is recovered, the witness agent will be terminated within one hop since the next server already records the log_{term} message. Figure 5.2 illustrates the above scenario.

Chapter 6

Reliability Evaluation

The reliability evaluation of our protocol is conducted by Stochastic Petri Net simulation [18, 19] using SPNP [20] as well as agent code implementation by using Concordia [4]. Reliability in this thesis is measured by the success rate of actual agents in completing their scheduled round-trip travels.

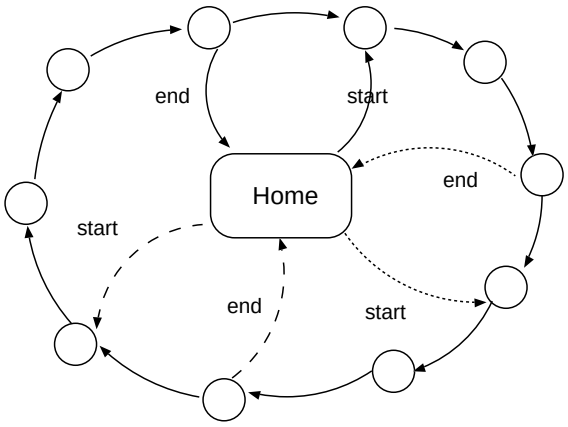

Figure 6.1: The Round-Trip-Travel Experiment

Our experiment aims at counting the number of successful 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 server 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. We have another daemon that monitors all the servers. We name it the *server monitor*. When it discovers that an agent server is dead, it restarts the agent server process within a specified time.

6.1 Server Failure Detection Analysis

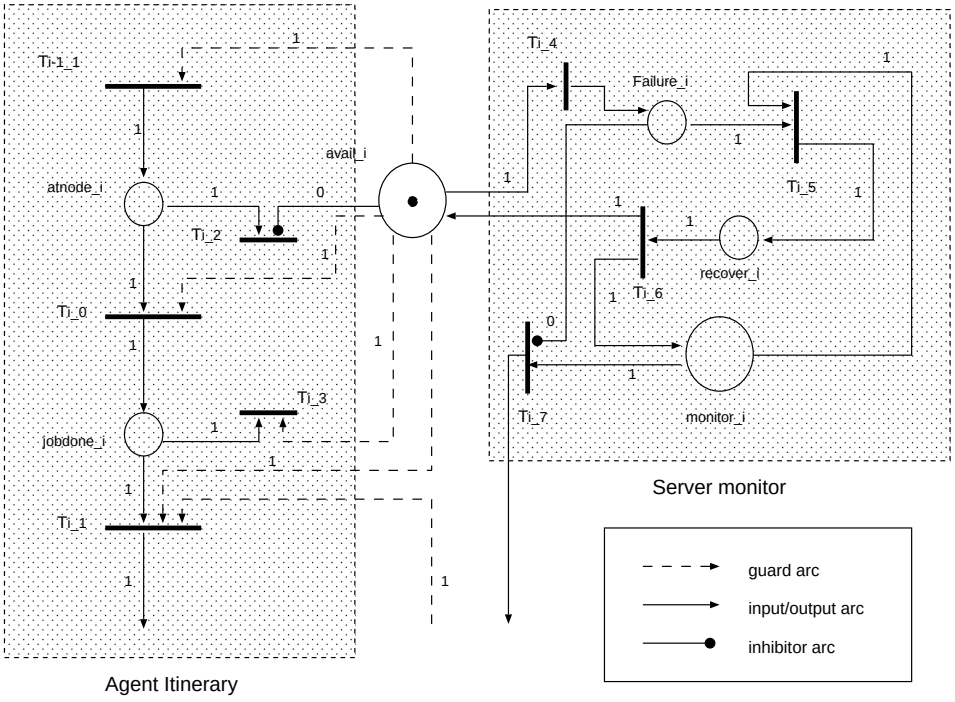

Figure 6.2: A server model with server failure detection

Figure 6.2 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 the time required to perform a recovery. The non-shaded *place* in the middle states the *availability of the server*. When there is a *token* 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 6.2 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.

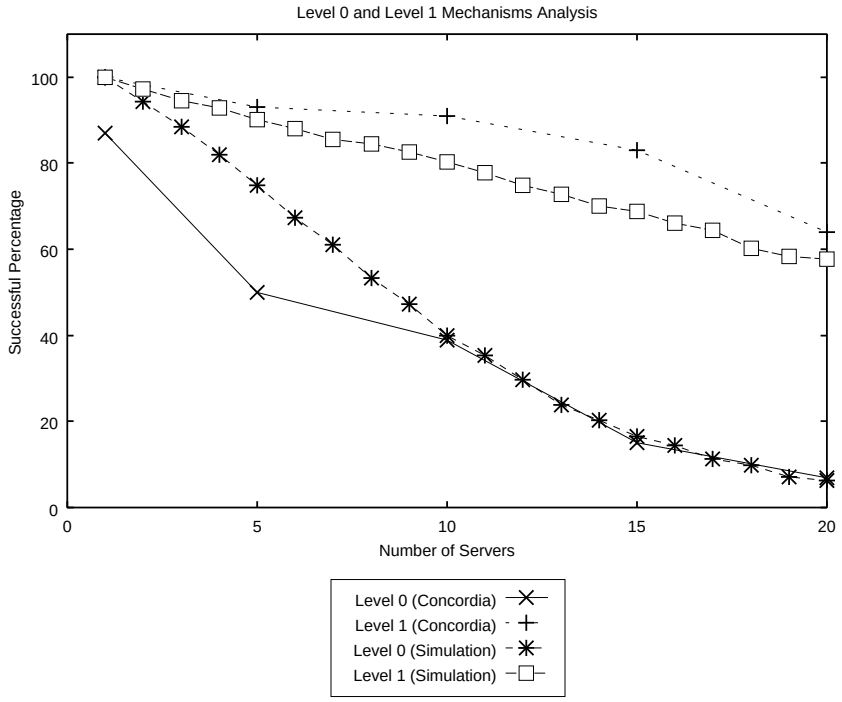

Figure 6.3: Evaluation result of server failure detection (Level 1 over Level 0)

The results of using both the Concordia implementation and the SPNP

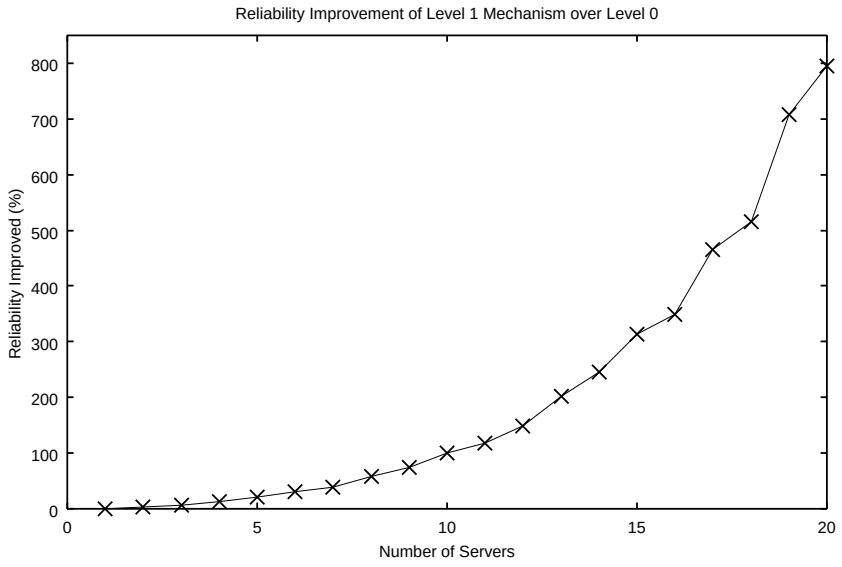

Figure 6.4: Reliability improvement with server failure detection

simulation are shown in Figure 6.3. The experiment compares two levels of fault-tolerance. One type represents the level 0 fault-tolerant mechanism implementation while another type represents level 1 implementation. 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 level 1 implementation drops much slower than the system with level 0 implementation. With the measurement of 20 servers in the agent itinerary, the successful rate of the agents with level 1 implementation falls between 55 and 60 percents. The successful percentage of the level 0 implementation, on the other hand, falls below 10 percent for both simulation experiment and Concordia implementation. Figure 6.4 shows the overall improvement of the level 1 implementation versus the level 0 implementation. The increasing slope implies that the advantage of *level 1* implementation becomes more significant as the number of servers increases.

The result measured by using simulation shows a monotonic increasing relation between the successful rate and the number of servers. As the number of servers increases, the number of successful round-trip-travels decreases progressively. It is a reasonable observation 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.

6.2 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 level 2 mechanism.

Figure 6.5 shows the Stochastic Petri Net that models both the server failure detection and recovery as well as the agent failure detection and recovery mechanisms. The two shaded modules on the right are similar to the structure of the server failure detection and recovery model (Figure 6.2). The modules on the left represent the additional structures that are required for the agent failure detection. We can observe from the model that the number of components required for the agent failure detection is much more than that for the server failure detection alone. This implies that the agent failure detection is more expensive and complex.

Our experiment is carried out by simulation with up to 20 servers, which is shown in Figure 6.6. The result indicates that the successful percentage of a round-trip travel in the level 2 fault-tolerant mechanism is further improved with respect to that with only the level 1 fault-tolerant implementation. The level 2 fault-tolerant mechanism can always recover failed agents, i.e., we have

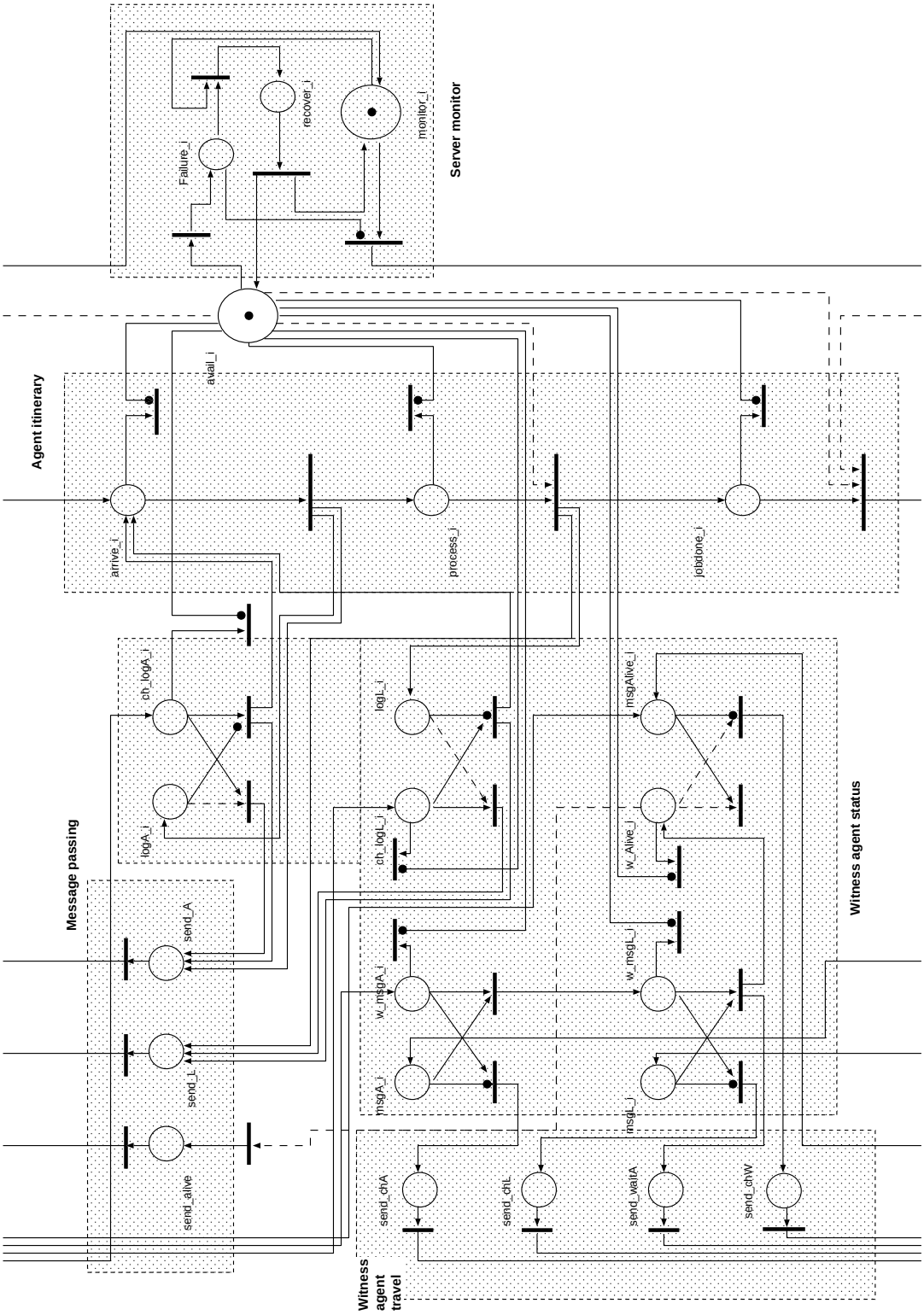

Figure 6.5: A server model with agent failure detection

a 100 percent recovery. Figure 6.7 depicts the reliability improvement of the level 2 fault-tolerant mechanism over the level 1 fault-tolerant mechanism. The result shows that the reliability is further enhanced. It reaches about 80 percent with an itinerary of 20 servers. However, one side effect is that whenever we have recovered an agent, the new agent may encounter another failure. This generates extra agents. Figure 6.8 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. This shows that more resources will be consumed and consequently the complexity of the system is increased.

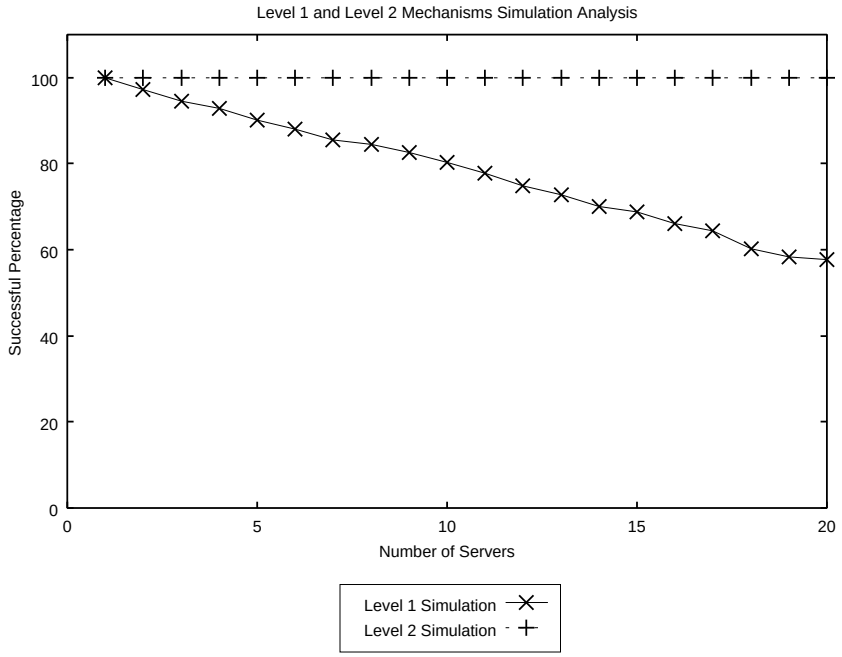

Figure 6.6: Level 1 and Level 2 simulation result.

Note that level 3 fault-tolerant involves link failures for more complicated scenarios, which is not included in our experiment for this thesis. This requires efforts in future research.

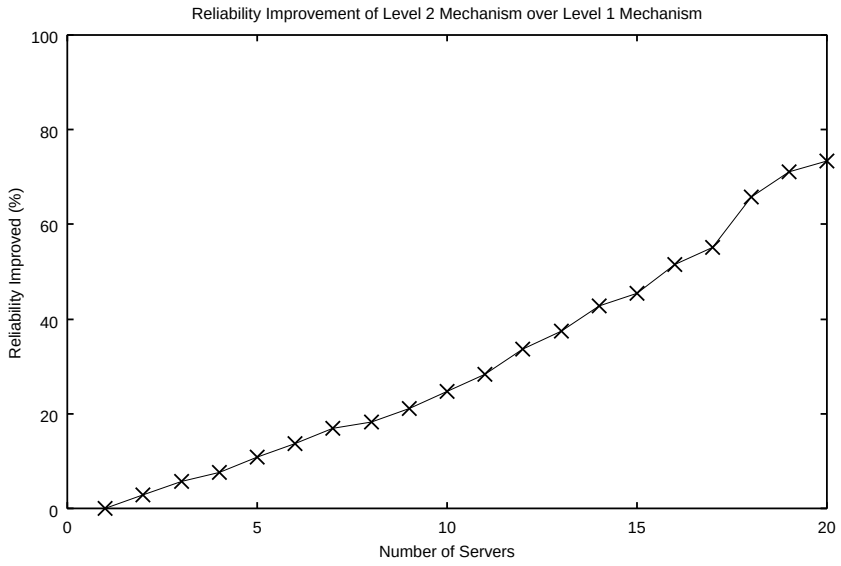

Figure 6.7: Reliability improvement with agent failure detection and recovery

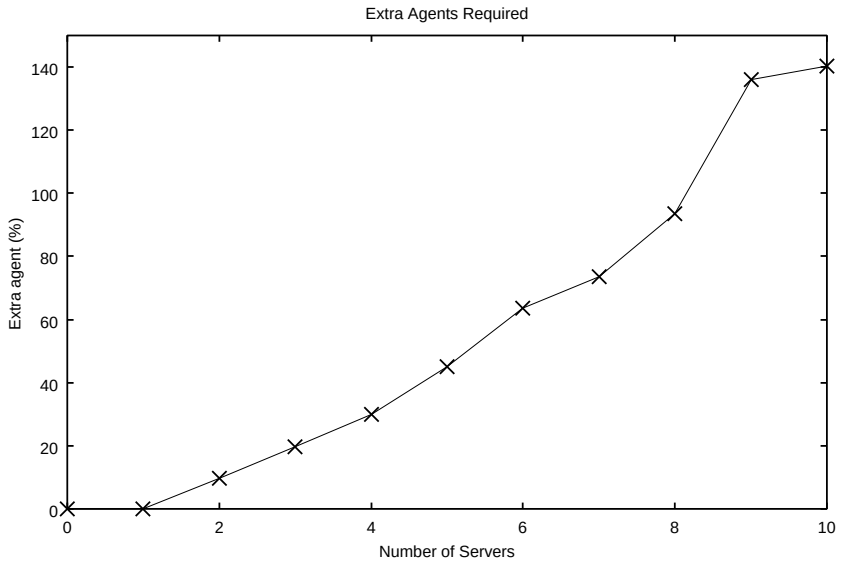

Figure 6.8: Extra agent per successful round-trip travel.

Conclusion and Future Work

In this thesis, we categorize the fault-tolerance of mobile agent systems into four levels. We also analyze different failure scenarios that may happen in the mobile agent systems. Moreover, we design a progressive fault-tolerant scheme that can detect the server, the agent, and the link failures. We further discribe 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 provide mathematical analysis of the mechanism. The analysis has shown a impossibility result of the liveness of the system. It shows that the liveness of the mechanism conditioning on the server failure arrival rate. The analysis also provides proves on the possibility of simplification of the mechanism. We conduct reliability evaluation of the proposed mechanism for server failures and agent failures. The result shows that, under the condition for up to 25 servers, with the server failure detection only (level 1), we achieve a significant improvement of the successful rate of the agent round-trip travels by two hundred percents. In addition to the server failure detection, we further improve the reliability by using the agent failure detection (level 2) by two hundred and seventy-five percent 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 agent resources and reliability of the proposed scheme are provided in this thesis.

In the future, we can model and perform more complex experiments on the level 3 fault-tolerant mechanism. Also, we can perform a more detailed

analysis of the mechanism such as the probability distribution of the system failure inter-arrival time. Note the fault detection and recovery mechanism can only tackle the stopping failure. We can further extend the mechanism to handle the Byzantine failure.

Bibliography

[1] A. H. Chan, T. Wong, C. K. Wong, and M. R. Lyu, "Design, implementation and experimentation on mobile agent security for electronic commerce applications," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1871–1878, 2000.

[2] 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*, vol. 1, no. 3, pp. 123–127, 1998.

[3] 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*, vol. 1, no. 3, pp. 111–121, 1998.

[4] D.Wong, N.Paciorek, T.Walsh, J.DiCelie, M.Young, and B.Peet, "Concordia: an infrastructure for collaborating mobile agents," in *Proceedings of 1st International Workshop, MA'97*, pp. 86–97, Lecture Notes in Computer Science 1219, 1997.

[5] Tryllian, "http://www.tryllian.net/."

[6] S. Pleish, , and A. Schiper, "Modeling fault-tolerant mobile agent execution as a sequence of agreement problems," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed System*, pp. 11–20, 2000.

[7] S. Pleisch and A. Schiper, "Fatomas - a fault tolerant mobile agent system based on the agent-dependent approach," in *The International Conference on Dependable Systems and Networks*, pp. 215–224, 2001.

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

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

[10] V. F. Nicola, *Checkpointing and the Modeling of Program Execution Time*, pp. 167–188. M. Lyu (ed.). John Wiley & Sons, 1994.

[11] Y. Huang and C. Kintala, *Software Fault Tolerance in the Application Layer*, pp. 139–165. M. Lyu (ed.). John Wiley & Sons, 1994.

[12] K. Rothermel and M. Stasser, "A fault-tolerant protocol for providing the exactly-once property of mobile agents," in *Proceedings of 17th IEEE Symposium on Reliable Distributed Systems*, pp. 100–108, 1998.

[13] M. Stasser and K. Rothermel, "Reliability concepts for mobile agents," *International Journal of Cooperative Information Systems (IJCIS)*, no. 4, pp. 355–382, 1998.

[14] M.Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of ACM*, vol. 27, pp. 228–234, April 1980.

[15] J. Baumann and K. Rothermel, "The shadow approach: An orphan detection protocol for mobile agents. technical report tr 1997/09," tech. rep., Faculty of Computer Science, University of Stuttgart, 1997.

[16] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[17] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[18] L. Tomek and K. S. Trivedi, *Analyses Using Stochastic Reward Nets*, pp. 231–248. M. Lyu (ed.). John Wiley & Sons, 1994.

[19] D. Xu and Y. Deng, "Modeling mobile agent systems with high level petri nets," in *IEEE Systems, Man, and Cybernetics,*, pp. 3177–3182, 2000.

[20] C. Hirel, B. Tuffin, and K. S. Trivedi, "Spnp: Stochastic petri nets, version 6.0.," in *11th International Conference of Computer performance evaluation: Modeling tools and techniques*, Lecture Notes in Computer Science 1786, Springer Verlag, 2000.

Appendix A

Glossary

S_i	Server i.
n	Total number of servers in the itinerary of the actual agent.
α	The actual agent.
ω_i	The witness agent in Server i.
ρ_i	The probe migrates to Server i.
log^i_{arrive}	The log message logged by the actual agent at Server i when the actual agent arrives at Server i.
msg^i_{arrive}	The message sending from the actual agent at Server i to the witness agent in Server $i-1$ when the actual agent arrives at Server i.
log^i_{leave}	The log message logged by the actual agent at Server i when the actual agent is ready to leave Server i.
msg^i_{leave}	The message sending from the actual agent at Server i to the witness agent in Server $i-1$ when the actual agent is ready to leave Server i.
log_{term}	The terminating message sending from the actual agent when it arrives at the last server of its itinerary.
e_i	The number of rounds needed for an actual agent to complete computation in server S_j.

e^*	The upper bound of $e_i, \forall i \in \{0, 1, \ldots, n-1\}$.
a_{ij}	The number of rounds needed for an agent to travel from S_i to S_j.
a^*	The upper bound of $a_{ij}, \forall i, j \in \{0, 1, \ldots, n-1\}$.
r_{ai}	The number of rounds needed for a probe to recover an actual agent in S_i.
r_a^*	The upper bound of $r_{ai}, \forall i \in \{0, 1, \ldots, n-1\}$.
r_{si}	The number of rounds needed for a server monitor (in Level 1 fault-tolerant mechanism) to inspect and recover S_i.
r_s^*	The upper bound of $r_{si}, \forall i \in \{0, 1, \ldots, n-1\}$.
$T_{recover}$	The number of rounds for server monitor to recover a failed server.
T_{arrive}	The timeout for ω_{i-1} waiting for msg^i_{arrive}.
T_{leave}	The timeout for ω_{i-1} waiting for msg^i_{leave}.
T_{alive}	The timeout for ω_{i-1} waiting for msg^i_{alive}.
$T_{heartbeat}$	The period of the heartbeat message msg^i_{alive}