INCORPORATING APPLICATION-TRANSPARENT NODE-CRASH TOLERANCE TO A SOFT REAL-TIME SELF-PLANNED AGENT FRAMEWORK

RAMPRASAD MADHAVAN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2005

Your file Votre référence
ISBN: 0-494-10292-6
Our file Notre référence
ISBN: 0-494-10292-6

Abstract

Incorporating Application-Transparent Node-Crash Tolerance to a Soft Real-time Self-Planned Agent Framework

Ramprasad Madhavan

Fault tolerance is essential to any soft real-time distributed system; besides correctness and timeliness. Traditionally system designers are required to consider both real-time and fault-tolerance requirements while building real-time applications. This is a complex task for a designer. In general distributed systems, fault tolerance has been researched well. However, significantly less work has been done in the field of fault tolerance in soft real-time systems. This thesis focuses on achieving application transparent fault-tolerance in a soft real-time system framework and addresses the issue of redundancy management in the presence of deadlines. Specifically, the thesis focuses on incorporating application-transparent node-crash tolerance in a soft real-time self-planned agent framework (SPAF). A SPAF application is decomposed into several missions and each mission is completed by successfully completing multi-agent tasks through a sequence of phases. Each task in a mission can have many solutions and the choice of the solution depends on the remaining time and available resources. Fault tolerance is achieved by using the conventional primary-backup approach in conjunction with the dynamic task planning feature of SPAF. A cold backup and hot backup are used to accomplish application and system recovery during a node crash respectively. The model and the design of the fault tolerance solution are presented in detail. The functionality and efficiency of the fault tolerance design is illustrated through the implementation and simulations using a custom built application respectively. The test results are very encouraging and the application performance is almost the same even after inclusion of the fault tolerance mechanisms.

Acknowledgments

First of all I would like to thank Dr.R.Jayakumar for his generous patience, support and guidance throughout my work and studies at Concordia. My sincere gratitude to Dr.H.F.Li whose guidance was priceless. His idea forms the basis of my thesis.

I owe many thanks to my friends (esp. inconcs, tamizha tamizha), roomies and colleagues from whom I have been able to learn and achieve much much more than I ever thought possible. Every moment in their company was fun.

I would like to thank my parents for their continuous love and support. Their belief in me is my motivation to strive for excellence in studies and life. Without them, I would'nt have been here writing this.

Contents

List of Figures

List of Tables

List of Acronyms

ACL - Agent Communication Language

AMS - Agent Management Service

FIPA - Foundation for Intelligent Physical Agents

MMS - Mission Management Service

SPAF - Soft Real-time Self Planned Agent System

TMR - Triple Modular Redundancy

Chapter 1

Introduction

1.1 Overview

Computers have been used in a variety of applications starting from desktop to industrial, medical, emergency and various others. They are being introduced in new hardware and software environments every day. Many of these systems not only require correct functional behavior but also require correct temporal behavior. Such systems are called *real-time systems* [SR91]. Examples of real-time systems include air traffic control systems, space stations, life support systems etc.

Real-time systems can be classified as *hard* and *soft* real-time systems. The tasks in hard real-time have stringent timing constraints and consequences of missing a deadline may be catastrophic. Critical systems such as aircraft control systems, space stations, radar missile tracking etc come under this category. On the other hand, the tasks in soft real-time systems have less stringent timing constraints, and in most cases, the quality of the service decreases as a function of the remaining time. Examples include: multimedia, security monitoring, stock monitoring etc. In this thesis, only the soft real-time systems are considered.

Most real-time systems are distributed in nature. The presence of multiple distributed processors increases the chances of a failure. Due to performance effects of violating the timing constraints in soft real-time systems, it is important that the task is completed even in the presence of faults.

Consider an intensive care unit system of hospitals. Such systems perform various monitoring tasks like processing ECG signals, brain waves, heart beats etc., and give an early warning during possible life threatening situations. A wide range of medical systems such as Neuronet[KBS91], Intelligent alarmer [YC98] etc. with similar data acquisition, processing and display requirements are also available. Systems like these must be able to perform inspite of unexpected faults.

In a similar way, the systems that perform automated monitoring of stock exchanges [SKAS] fall under this category. Faults in such system could result in loss of millions of dollars. Similar soft real-time systems requiring fault tolerance can be found in [LKR02] [TG02] [Ziq03]. Therefore, it is essential that fault tolerance should be an integral part of any soft real-time system [AOSM05][SD02].

1.2 Faults and Fault Tolerance

Every computer in operation today is certain to experience faults during its operation, whether it is a normal desktop system or a real-time system. Faults can arrive into a system from different sources like: operational environment, human users, hardware malfunction, software design errors, etc. A fault can be defined as a defect or a flaw that occurs within a system component.

Depending upon the type of the component, faults can be classified into two types: *hardware faults* and *software faults*. In this thesis, a soft real-time self-planned agent framework (SPAF) is considered. Hence, a hardware fault is said to have occurred when a hardware computing unit or node of a system fails. A software fault is said to have occurred when a software computing unit or a software agent fails.

Depending on the duration of failures the faults can be classified into three kinds: *permanent*, *transient* or *intermittent* faults. Permanent faults are caused by total failure of a computing unit, transient faults are caused by temporary malfunctions or unavailability of a computing unit and intermittent faults are caused by repeated occurrences of transient faults.

A system is *fault-tolerant* if it continues to perform its specified tasks in the presence of hardware or software faults [Joh89]. Generally, fault tolerance can be achieved by replication or redundancy by the addition of information, resource or time beyond what is needed for normal system operation [GS97] [Joh89]. Transient and intermittent faults can be tolerated by time redundancy or, in other words, the system takes additional time to re-try or re-execute a faulty task [RT93]. Permanent faults are tolerated by spatial redundancy or, in other words, replication of data and/or computations. In the case of software faults, basic unit of replication is the software computing unit (or agents) and for hardware faults, basic unit of replication is both the hardware and software computing units (nodes and agents). In this work, tolerating permanent node crashes is given prime importance.

One of the popular techniques for tolerating permanent hardware faults in multiprocessor real-time systems is the use of *triple modular redundancy(TMR)* [LV62] [Pra86]. In TMR, the hardware is triplicated and a majority voting is performed to determine the output of the system. The system generates a correct output even if one of the three modules is faulty. An extension of TMR is NMR [MK86] where N modules are used instead of three.

Another important and widely used technique is the *primary-backup approach* [BN93], which uses a primary and one or more backup processors. The primary processor performs all the computation and frequently updates its backup. When a failure is encountered a backup takes over the role of primary from the point of last update.

1.3 Application Transparent Fault Tolerance

The complexities of the general distributed systems and real-time systems are constantly increasing. Designing fault tolerant applications has become a complex task for the system designers. In distributed systems, research has been done in providing application transparent fault tolerance. A fault tolerance layer is generally built between the application and its runtime environment [T.B94], [BRNB]. The encapsulated layer either implements its own or its runtime environment's fault handling features to provide software and hardware fault tolerance.

In soft real-time systems numerous fault tolerance mechanisms have been proposed. The fault tolerance requirements are specific and designed along with the application. The result is that the fault tolerance constraints are imposed on the designer in addition to the real-time constraints. Significantly less work has been done in providing application transparent fault tolerance for soft real-time systems. The main reason being the problem of managing the fault tolerant tasks in the presence of deadlines independent of the application.

1.4 Contribution

The main contribution of this thesis lies in the design and implementation of an application-transparent node-crash tolerance solution for a soft real-time self-planned agent framework (SPAF) [Jin05]. A SPAF application is decomposed into several missions and each mission is completed by successfully completing multi-agent tasks through a sequence of phases. Each task in a mission has multiple solutions and the choice of a solution depends on the remaining time and resources. Also, SPAF provides every mission with a task planner to perform dynamic task planning during the mission execution. Thus, dynamic task planner inherently tolerates the timing faults. The fault tolerance solution presented here makes use of the conventional primary backup approach in conjunction with the dynamic task planning feature of SPAF and a simple handshake based fault detection mechanism for tolerating permanent single node crash. A node crash results in a permanent loss of information that can affect both the application and the system services. Cold backup and hot backup solution are used to accomplish application and system recovery during a node crash respectively.

A SPAF application has missions with a variety of deadline requirements. Missions with tight deadlines may not be able to afford the time consumed in the maintenance of the backup. In that case, a simple mission reset solution is provided. The appropriate fault tolerance solution is assigned during the mission admission process.

1.5 Thesis Outline

The major research work of this thesis lies in the design of a comprehensive fault tolerance scheme for SPAF. The design is aimed to offer a minimum possible overhead in the absence of failures and a quick recovery in the presence of failures, independent of the application. Given below is a brief overview of the various chapters.

In chapter 2, the model of the soft real-time self-planned agent framework considered is presented. Relevant architectural details are provided. The failure models are also discussed.

In chapter 3, the high level analysis and model of the fault tolerance scheme is presented.

In chapter 4, the design of various fault tolerance components and design changes to the existing system for optimizing the overall system performance-overhead tradeoff, is discussed in detail.

In chapter 5, validation of the above design, verification of the efficiency of fault tolerance (recovery) mechanisms and measurement of the performance impact on the application with the inclusion of fault tolerance are performed through various tests. These tests are conducted using a custom-built application.

In chapter 6, the thesis is concluded and the future research work that could be embarked on the node-crash tolerant SPAF is described.

Chapter 2

Soft Real-time Self-planned Agent Framework

Real-time systems are growing more complex as a result of deployment in open and distributed environment such as internet. These systems require flexible, adaptive and intelligent system components for successful operation. The multi-agent paradigm is a branch of distributed systems that was introduced to simplify the design of complex systems operating in open and dynamic environment. A software agent [Woo02] is an autonomous entity that can interact with other agents to fulfill important goals. Real-time multi-agent systems with soft timing constraints is an emerging field of research and some systems supporting soft real-time constraints can be found in [HLVW05]. For our study we consider the soft real-time self-planned agent framework (SPAF) of [Jin05].

In this chapter, the model and the architecture of SPAF is described in detail. Later, the possible failure models associated with SPAF are classified.

2.1 Application Model

An application of SPAF is composed of several periodic and sporadic activities called *Missions*. The application operates in an environment where an external source may trigger launching of missions. The missions are totally independent of each other, in other words, there is no interaction or dependency among the missions during their execution.

A mission is accomplished by successfully completing a set of tasks. The tasks in a mission are completed by a dedicated set of agents in a sequence of *phases*. The model of the mission and a phase of a mission are described in the following subsections.

2.1.1 Mission

An application has several soft real-time missions; that is, the missions have a soft deadline and failing to complete a mission within the given time limit will not cause any serious consequence like an external system failure. Even though the missions are soft in nature, completion of missions is very important for the application. Each mission of an application is associated with an importance factor such that higher the importance factor more will be the priority given to that mission in accessing the available system resources. Also, each mission is associated with a quality factor that corresponds to the quality of the computation results.

2.1.2 Phase

A phase involves execution of a task. Since a multi-agent application is considered, every task is completed by a set of agents working together. A task can be completed in many ways with a solution decided at runtime based on the mission time and system resource availability. Depending on the solution, quality of the task varies.

A phase has a soft deadline just like a mission's deadline. In other words, failing to complete a phase in the expected time may not necessarily cause a mission to fail.

2.2 System Model

The system provides the application with two kinds of agents for every mission namely, the *Planner agent* and the *Application agent.* Planner agents are generic and perform task planning for their mission. Application agents on the other hand are application specific and are primarily for executing the tasks. The high level model of planner and application agents will be covered in the remaining of this section.

2.2.1 Planner Agent

A planner agent is provided for each mission to perform task planning on behalf of the application. The application provides the planner with the knowledge of all possible solutions in each phase of its mission. Thus, the main duty of the planner is to choose a solution to complete the task in that particular phase. In addition to planning, the planner also acts as a coordinator for the application agents. Thus the planner has the following knowledge:

1. All possible solutions for the current phase
2. Deadline of the mission and the remaining time
3. Current availability of the system resources
4. Dependency between the solutions of successive phases in terms of the survival of agents between the phases
5. The set of agents available for its mission

The above knowledge to the planner is provided either by the application or is collected at runtime by the planner itself. Described below are the two important interfaces through which the planner receives the knowledge.

Application-Planner Design Interface

The developer provides the following application related information to the planner at the design time:

1. Mission related information: Mission deadline, period, importance, phases.
2. Phase related information: Solution set (includes resource needs, quality, time, set of agents).

Application-Planner Runtime Interface

There exists a subset of application agents in each phase of a mission that survive through any two successive phases. The planner requires this information to perform task planning of a phase. Hence, the developer chooses those agents at the design time and provides input to the planner through the application agents at runtime.

2.2.2 Application Agents

Every application mission has a dedicated set of agents for executing its tasks. Application agents have a special capability set in terms of roles they could play. These roles are specified by the application developer at the design time. The agents are instructed to play certain roles by the planner agent depending on the solution chosen for that phase.

Application agents can survive over multiple consecutive phases in a mission. Based on the survivability of an agent, they are classified into two types:

1. *Torch carriers* which carry forward the mission state(data) from one phase to another in terms of the knowledge they acquired during that phase.

2. *Worker agents* which are specific to a given phase and die at the end of that phase completing the assigned work.

2.3 System Architecture

The SPAF provides the complete infrastructure to manage missions and the agents of an application. The system comprises of a basic multi-agent platform on which agents are deployed and their existence can be managed. A soft real-time framework manages the missions and provides task planning within each mission.

In this section, the architecture of the system is introduced with a description of the multi-agent platform and the soft real-time framework.

2.3.1 Multi-agent Platform

The multi-agent platform provides a distributed environment within which agents live and operate. The platform provides all the basic services required to manage and operate the agents.

The platform has the following logical entities:

1. Agent: An agent is the fundamental actor on the platform that combines one or more capability sets. An agent identifier labels an agent so that it can be identified unambiguously within the agent world. An agent may be registered to a transport address to which it can be communicated.

2. Agent management service (AMS): The AMS exerts the supervisory control over the Agent platform. It is responsible for the agent life cycle management and the agent name service. Further details of the AMS are elaborated later.

The platform has the following physical entities:

1. Agent container: The agent platform is distributed over a network of nodes in the form of multiple containers that can host agents. The system has one *main container* that contains all the service agents and several *application containers* that contains the agents capable of playing application specific roles. All the application containers are registered with the main container.

2. Real-time Agent: The platform has a basic entity called realtime agent that is extended from a realtime thread. All the agents in the system are extended from the realtime agent.

Agent Management Service

The AMS is the most important service of the multi-agent platform and is implemented as a real-time agent. It performs the following services:

1. Agent life cycle management service: The AMS instructs the underlying platform to perform agent life cycle operations such as Agent Create and Agent Destroy.

2. Agent name service: The AMS maintains an agent registry with the name and location of every agent present in the platform. Whenever an agent is created or killed then the AMS performs registration or deregistration operation on the agent registry. The main purpose of this registry is to facilitate agent communication in the platform.

Communication System

The communication system provided by the platform allows the agents to communicate among each other using the FIPA ACL messages [FIP00a]. The agents make use of the agent registry associated with the AMS to locate and talk to the remote agents. The agent platform makes use of sockets to transfer messages between agents residing in different hosts.

2.3.2 Self Planned Agent Framework

The framework provides all required mission management and execution services to the application. This section describes the two services in detail.

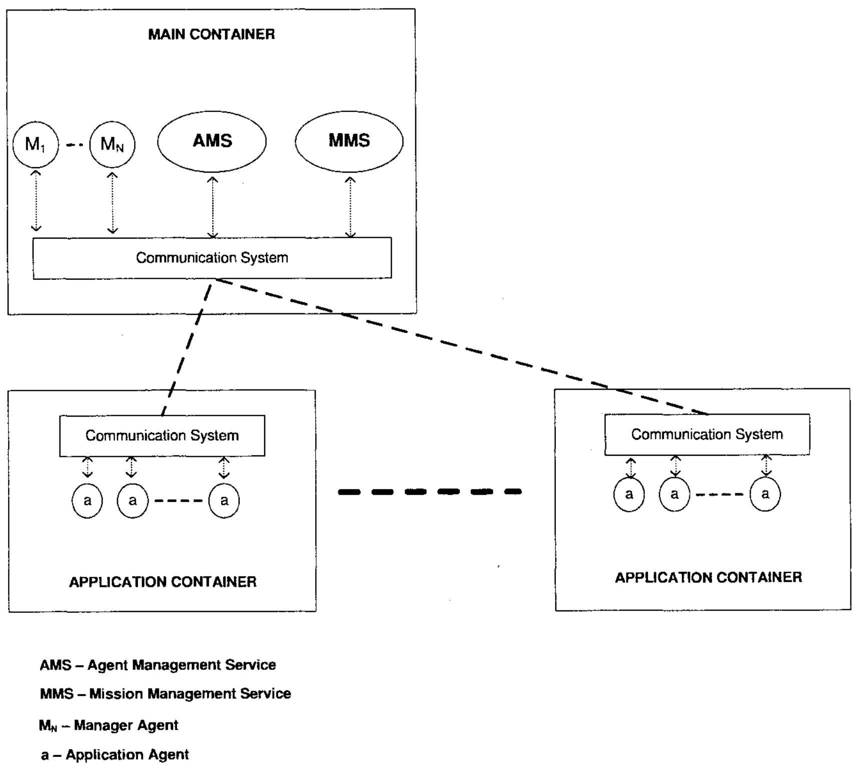


Figure 2.1: Soft real-time agent system

The *mission management service (MMS)* includes mission admission control. The mission admission control primarily checks for the feasibility of admitting new incoming application missions into the system. This service is provided through a service agent present in the main container.

The mission execution management services are services that are provided by the system once a mission is admitted and initialized. This includes mission planning, agent co-ordination, mission exception handling etc. The mission execution management services are provided through a *manager agent* that is equipped with a *task planner*. Each mission has its own manager agent.

The following sub sections discuss in detail the above mentioned components.

MMS

The MMS agent is responsible for the registration and de-registration of the missions. An incoming mission is registered only if it passes the time consistency tests (execution time feasibility) and the resource availability.

Once a mission is registered, a manager agent is created for the mission and upon completion, all of its agents (manager and application agents) and associated system resources are released.

The MMS resides in the platform's main container along with the AMS. It interacts with the AMS to perform its own operations.

Manager Agent

Every mission is initialized along with a manager equipped with a task planner. Upon receiving a start signal or an occurrence of a timed event, the manager starts the task planning process by invoking its task planner. The task planner is responsible for the task planning and co-ordination of the agents of the mission from start to end.

The time-space diagram in Figure 2.2 shows a normal execution model of the mission. Task planner upon receiving a "start mission" signal from its manager starts the planning of its first phase. Once a solution is chosen, it recruits the required worker agents by interacting with the AMS. The planner assigns the task to the specified agents through the invocation parameters and then waits for agent reports. Once all agents have reported and task successfully completed, the planner preserves the torch carriers which carries the mission state to next phase and kills the rest.

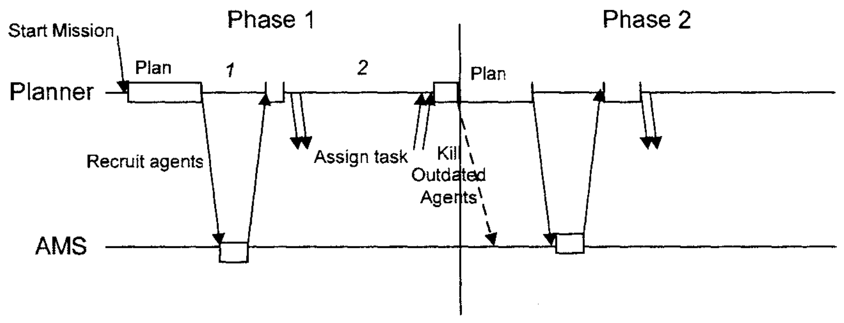


Figure 2.2: Planner Execution Model

2.3.3 General Application Agent

The system is based on multi-agent role based computation and provides the application with agents that are capable of executing roles. The roles may be assigned to the agents either at design time or at runtime. Agents may be required to carry and play many roles during their life time. Hence, a scheduler is built within to schedule the roles one at a time. The agents also have the knowledge of the dependency between the available roles if they are to play multiple consecutive roles in a mission. Once a role is completed it reports back to its manager (Task planner) with the dependency information. In order to record and share the knowledge attained while playing a role, every agent is provided with a local storage.

2.4 Failure Models

In this section, the possible failure models associated with SPAF are classified.

2.4.1 Application Fault

An application fault is said to have occurred when one or more application missions fail. Since the missions are independent of each other during the execution, failure of one mission does not affect the other. The various faults that can cause a mission failure are:

Timing faults

A timing fault is said to have occurred when one or more agents do not complete the assigned work within in a specified time. A timing fault may lead to a phase failure that in turn may result in a mission failure. A timing fault is most likely caused due to a delayed response from agent present in an excessive system load environment.

The task planner inherently tolerates the timing fault by flexible planning. During a phase failure, the task planner automatically switches to a planning mode wherein, the remaining mission is executed using cheaper solutions taking less time.

Application Agent Faults

When an agent is no longer accessible by the application mission then agent failure is said to have occurred. Agents can crash during an execution as a result of a design or runtime error. Agent crash can result in permanent loss of any mission information it carries. A single agent crash leads to stalling of a mission and eventually a failure.

A system can also suffer from temporary agent faults such as unavailability of certain agents for a finite amount of time. These occur primarily as a result of temporary node failure. Such faults can also lead to timing failures.

2.4.2 System Failures

A system failure is said to have occurred when system or platform services fail. Since an application is closely bound to the system, a failure in the system may lead to an entire application failure. The following subsections discuss the possible system failures:

Container Failure

The multi-agent platform is distributed over a network of nodes in the form of containers. Main container contain service agents while application containers contain application agents. A node can contain one or more containers. A node failure can lead to inaccessibility of the main container and/or application containers residing on that particular node. Failure of each kind of containers offers different levels of damage to the application.

1. Application container failure

   An application container failure results in unavailability of all its application agents running on it. Agent crash leads to loss of mission related information it carries and thus failing the corresponding mission. Since an application container contains agents that are operating in different missions, one container failure can cause multiple mission failures.

2. Main container failure

   When a main container fails, all system services such as AMS, MMS, planners etc become unavailable to the application. The application no longer has control over its application agents, ultimately leading to the failure of all its operating missions.

Failure of a container can also be temporary. When a node becomes unavailable for a finite amount of time then all the containers running on it become inaccessible for that period of time. This introduces an uncertain delay in the related missions. Once the node becomes available, the application returns to a normal operating mode. The effect of the temporary node failure on an application depends on its duration.

Service Agent Failure

The application constantly requires the services of the framework to operate and hence, there is a possibility that the entire application fails when there is a service agent failure. The services are implemented as agents and they fail as a result of design or runtime errors.

2.4.3 Failure Model Considered

From the above discussion, it is seen that a permanent node crash definitely offers the maximum damage to the application. As discussed in chapter 1, node crashes are common in distributed systems and systems with real time constraints cannot afford this failure.

In this thesis, the SPAF is made tolerant to single node crashes that result in the failure of the main container or application container or both. The following chapters describe in detail the fault tolerance model and the modified system architecture.

2.5 Summary

In this chapter, the model of a system considered for this study has been discussed. This is followed by a description of the system architecture providing the details of the important components at a level that concerns this study.

Later in this chapter, the possible failure models associated with the system were highlighted. Also, the significant failures that will be considered for fault tolerance in this study were discussed.

Chapter 3

Fault Tolerance Model

Node crash can result in the failure of main container or application containers. Since the two containers are of different nature, the failures must be dealt in a different manner. In this chapter, the fault tolerance model for application container and main container are presented.

3.1 Application Container

An application container contains application agents that are part of different missions of an application. An application container failure results in failure of all the agents running on it, which in turn, may result in failure of several application missions. This situation severely affects the application's performance.

3.1.1 Analysis

Node crash fault tolerance can be provided by replication of missions on two different physical locations, such that, upon failure of one, another can be used. Replication can either be active or passive.

In active replication, the entire mission is executed twice in parallel. During a node failure, the results of the non-affected missions are taken. This approach provides fault tolerance through fault masking and requires no fault detection mechanisms. Complex voting mechanisms need to be employed in the absence of failures so that only one of the two results is taken. Also, to employ this scheme one needs twice the system resources as that of no fault tolerance. This cost is not affordable in soft real-time systems.

In passive replication or primary-backup approach, the mission is executed only once and the important state changes during the execution are updated to the backup. Here the cost of fault tolerance is significantly lower than the previous approach and increases with the frequency of backup updates. The main setback of this approach is the need for failure detection and recovery mechanisms.

The primary-backup approach coupled with flexible mission planning appears to be a suitable fault tolerance scheme for SPAF application container failures.

3.1.2 Requirements

The fault tolerance mechanism is required to backup the important agent information in every application container from time to time. When an application container failure occurs, the affected missions, depending upon their condition are either terminated to a safe state or restarted using the backup information to meet soft real-time constraints.

3.1.3 Possible Solution Models

Based on the above requirements, the following fault tolerance solutions are possible.

Cold Backup Approach

Application container fault tolerance can be achieved by having *cold backups* of the active torch carrier agents in a container residing on a different node within the multi-agent platform. Since the mission state is carried from one phase to another by the torch carrier agents, their states are checkpointed in the form of newly created agents called clones at the end of each phase. These clones have the same knowledge and capability like the torch carrier agents at the beginning of the phase but as the phase progresses, the clones become cold backup. A worker agent is specific to a given phase and does not carry forward any mission information. Hence, its presence in the agent pool is sufficient for it having cold backups. Upon a failure, any other other agent in the system can take over its place.

When a failure of an application container is detected, all the affected missions are restarted from the current phase using the clones of the torch carrier agents and a new set of worker agents. If a mission is unable to find a solution due to limited time, it terminates its current execution and waits for the next start event.

The main advantage of this approach is that an affected mission is expected to recover from the current phase quickly with the readily available cold backup agents. This significantly increases the chances of a mission meeting its deadline even in the presence of failures.

The clear disadvantage of this method is the overhead in creating new clones at the end of phases of a mission. This overhead issue could lead to severe performance drawbacks as some missions may miss deadlines as a result of extra time consumed in performing the cloning. The overhead in creating the clones at runtime can be reduced tremendously if the torch carrier agents of the mission are determined before hand and the clones created at the initialization time but this is not feasible.

One possible solution for this problem is to make all the application agents in the system equally capable and to be created during the application initialization time. This way the time consumed to create a new clone with special capability and initial state during the runtime is almost completely eliminated. Now creating a clone means mere assigning of states to the agents (clones) already existing in a different node.

Checkpoint and Recovery

This model is similar to the previous model. Fault tolerance can be accomplished by backing up the mission state at the end of each phase as standard checkpoints. The checkpoints are stored in repository whose location is different from that of the mission agents, say Main container. When a failure is encountered, the affected missions are restarted by creating a new set of agents with the state corresponding to the checkpoints and capabilities similar to the original ones.

To store and maintain the checkpoints, one needs a repository and a repository manager that helps storing and retrieving the checkpoints.

This approach eliminates the overhead of managing the clones at runtime but introduces additional time in creating and initializing new set of agents with special state during the recovery process. This delay may reduce the chances of a possible mission completion in some cases. Additionally, the traffic through the checkpoint repository manager also increases as the number of missions in the application increases. This can be a performance bottleneck. Also the recovery manager must be fault free; otherwise an application container failure can result in failure of numerous application missions.

Mission Reset

This is the simplest of the three mechanisms wherein affected missions are simply reset to the initial state when a failure is detected. The mission restarts its execution using freshly initialized agents and attempts to meet the deadline by using the cheaper solutions in all phases of the mission. This approach offers no runtime overhead but also offers the least chance of mission completion during a failure.

3.1.4 Choice of Fault Tolerance Solution

Since the recovery time is the most important factor for a mission, the solution that provides the quickest recovery must be chosen. The cold backup approach appears to be the best choice among the three approaches.

An application can have a variety of missions based on the tightness of their deadlines. It is possible that certain missions may not be able to afford the time taken for clone management at runtime. For such missions, alternative solutions need to be chosen. The simple mission reset appears to be suitable for missions that have very tight deadlines.

Hence the system must assign a fault tolerance solution according to the nature of the incoming mission. The basic condition that decides the choice of fault tolerance for a mission at mission admission time is:

Slack time = Deadline - Minimum execution time

Overhead of cold backup = Total time taken to backup the mission state as clones

If slack time *is greater than* overhead of cold backup then choose cold backup approach.

If slack time *is less or equal to* overhead of cold backup then choose mission reset.

The *Deadline* is the actual deadline of a mission and the *Minimum execution time* is the minimum time required to complete a mission. The time taken to backup the mission state is calculated using the number of updates in a mission and an approximate time taken to transmit the checkpoint.

3.2 Main Container

Main container contains all the service agents that are required to manage the application missions and agents. A node crash can lead to a main container failure which causes failure of all the services that the system offers. This ultimately results in a complete application failure.

3.2.1 Analysis

Node crash tolerance can be achieved by replication of all the system services. As discussed in the previous section, even though active replication provides an on-the-fly fault tolerance, it is an expensive solution when one considers the cost of replicated processing and complex voting mechanisms. Again one finds the passive replication or primary-backup approach attractive in terms of requirements and cost. Here the primary processor processes all the service requests and sends a simple update message to its backup. When a fault is detected, the backup replaces primary and starts serving the future requests.

3.2.2 Requirements

The fault tolerance scheme is required to backup the state of the main container at a different location so that the backup information can be used to resume new services again. The main requirement of the fault tolerant solution is to recover all the failed services and the application missions correctly and quickly. Since one deals with a distributed system, it has to be ensured that the state of the new service must be consistent with the operational state of the rest of the system. Due to real time constraints, the recovery process must aim to minimize the recovery time.

3.2.3 Hot Backup Model

Main container fault tolerance can be accomplished using a primary and a hot backup main container. The backup container resides on a different host and with dormant clones of all the service agents present in the primary. The primary main container processes all the requests from the application and duly updates all the changes in its states on to the backup. Failing to update the important changes may cause an erroneous situation during the failure recovery. Hence an absolute consistency or a hot backup is required at all times.

The updating process is carried out using a primary-backup commit protocol which ensures that the local operations and state changes happen atomically in both containers. The commit protocol is described in detail in Chapter 4.

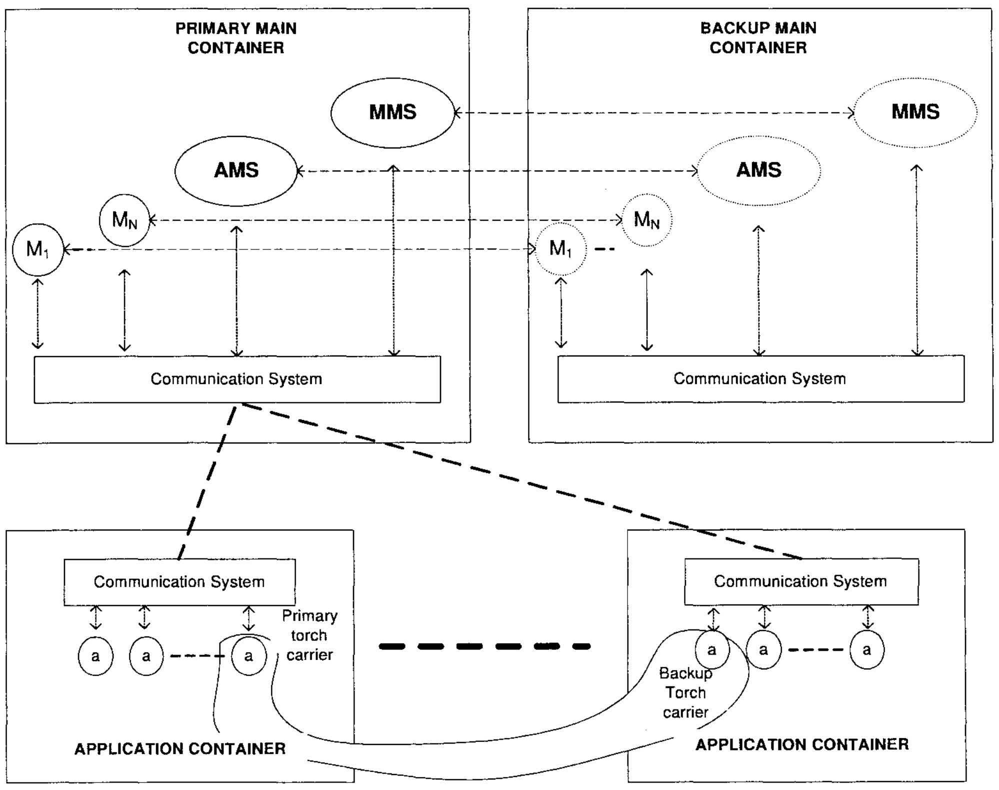
At each commit event, the primary service completes the required operation and updates its backup of the changes. The update can be a mere state update or a set of actions to be performed by the clone service. Events such as registration/deregistration of missions involve actions to be performed within the backup main container. Once the backup completes the update process and responds with the acknowledgement, the primary resumes its operation. If the protocol fails then a failure is detected.

The update process is equivalent to taking checkpoint of all the services in the primary main container and placing it in a different location. A main container checkpoint can be thought as a set of most recent states of the individual services after a successful update process. When a primary main container failure is detected, the backup starts from the recent main container checkpoint and assumes the role of the primary. In chapter 4, the recovery protocols are elaborated in detail and exception cases are pointed out.

The application environment is completely unaware of the presence of a backup main container. Therefore, a system interface is provided that masks the fault tolerance details from the application and transparently makes use of the backup upon primary's failure.

3.3 Summary

In the first section, the possible fault tolerance solutions for application container crash are analyzed and the solutions that matched the requirements are chosen. A change to the existing system design has been described to eliminate the execution time overhead offered by the fault tolerance solution. In the second section, the model and analysis of the main container fault tolerance solution has been presented.



AMS – Agent Management Service

MMS – Mission Management Service

M_N – Manager Agent

a – Application Agent

Figure 3.1: Node crash tolerant soft real-time agent system

Chapter 4

Fault Tolerant Self Planned Agent Framework Design

In this chapter, the models are elaborated and details are provided through the design of new system components and changes in the existing system design. The chapter elaborates the application container and main container fault tolerance design.

4.1 Application Container Fault Tolerance

The two important components of the fault tolerance mechanism are fault detection and fault recovery. In the first subsection, the fault detection mechanism is described and later, the design of the planner and application agent are presented.

4.1.1 Fault Detection Service

Application container crashes are detected by the *fault detection service* in the system. This service uses the conventional handshake mechanism to detect the node crash. Fault detection is accomplished by having a master that initiates periodic handshakes with the slave. The master resides in the main container and slave resides in the application container respectively. There are as many master slave pairs as the number of application containers in the system.

The master agent initiates the handshake process by an "Are you alive?" signal with a timeout and the slave responds with an "I am alive" signal immediately. The timeout value is determined during the application initialization time and may vary from one container

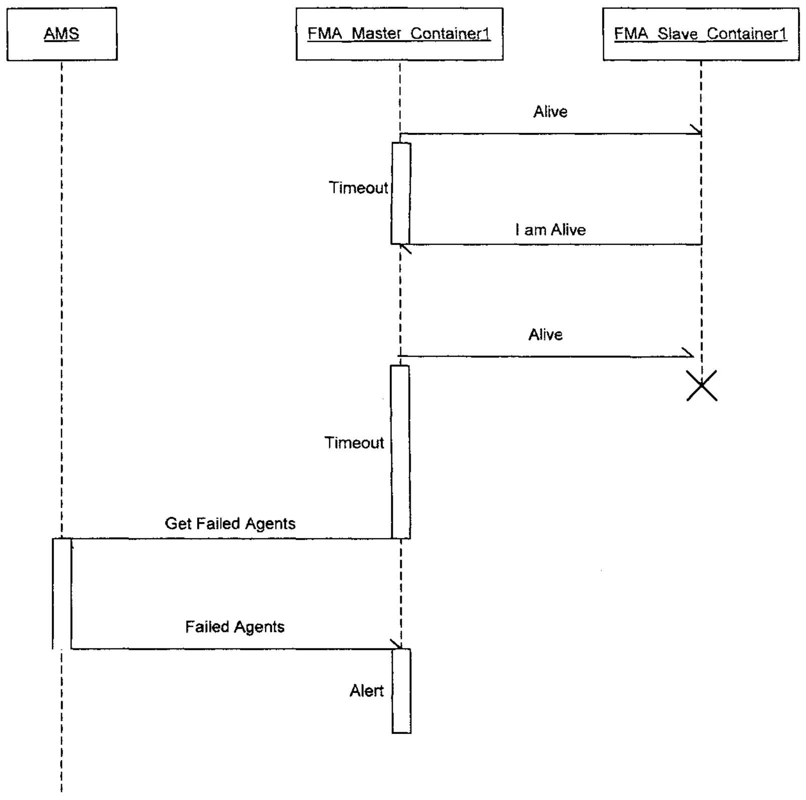


Figure 4.1: Fault Monitoring Service Protocol

to the other. The handshake process is successful if the slave responds within the given time. If time expires then the master announces the failure of the application container and alerts the managers of the affected missions. The master agent determines the affected missions by interacting with the AMS and includes the list of failed agents along with the alert. Figure 4.1 shows the handshake protocol between the master and the slave assigned to monitor an application container.

4.1.2 Planner

Every application mission is associated with a task planner that has the complete knowledge about its mission such as the phases of a mission, possible solutions of a phase, the name and location of its agents, roles played by its agents etc. Hence, the application container fault tolerance can be incorporated in the task planner of a mission manager. The task planner has two different instances: one capable of performing cold back-up and the other performing mission reset. During the mission admission, an appropriate planner is chosen for a given mission.

The following subsections describe in detail the design and operation of the two solutions.

4.1.2.1 Cold Backup

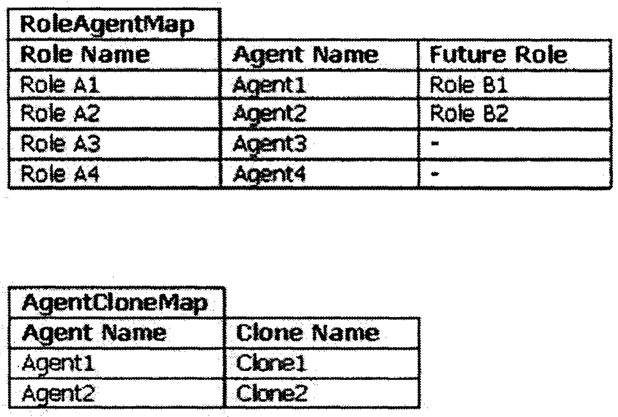
The design of the generic task planner is modified to perform the cold backup or clone management and the recovery management at execution time. The clone management includes two basic operations: creation of new clones and deletion of existing outdated clones at the end of a phase. The recovery management involves taking appropriate recovery actions during an application container failure depending upon the condition of its mission.

The planner performs the clone management operations with the knowledge of the name and location of the potential (next phase) torch carriers specified by the developer. At the end of a phase, the planner interacts with the AMS to recruit an agent in a different host to act as a clone for the potential torch carriers. Once all the clones have been created successfully, the older clones are deleted.

The planner maintains data structures to store and manage the dynamic knowledge of its agents that are currently serving the mission. The two important data structures are as shown in Figure 4.2.

The RoleAgentMap data structure contains the roles to be played, the agent that will play and the future roles playable (if any). The *role name* and *future role* is supplied by the developer and the *agent name* is dynamically obtained by the planner. Once a solution is chosen and agents are recruited, fresh information is stored in the data structure for that phase. The AgentCloneMap data structure stores the identification of current phase's torch carriers and its clones. At the end of each phase, the agents in the RoleAgentMap table that have future roles to play, will be assigned a clone.

Planner Execution Model



RoleAgentMap		
Role Name	Agent Name	Future Role
Role A1	Agent1	Role B1
Role A2	Agent2	Role B2
Role A3	Agent3	-
Role A4	Agent4	-

AgentCloneMap	
Agent Name	Clone Name
Agent1	Clone1
Agent2	Clone2

Figure 4.2: Data tables used by planner

This subsection describes the planner execution model in the presence and absence of failures.

- Normal Mode of Operation

  Once a mission is started, the task planner performs the task planning and chooses a solution for that phase. It interacts with the AMS to recruit the agents required for completing the task. The planner then assigns the role and role-related parameters to its agents and waits until the end of task. The task is completed once all the agents report back. The planner determines the potential torch carrier agents and clones them. Before starting the next phase, all the outdated clones of torch carriers and other regular worker agents are deleted. Figure 4.3 shows a sample mission scenario in the absence of failures.

- Recovery Mode of Operation

  A node failure can occur at any point of time during a phase and the planner must be able to handle accordingly. There can be three cases depending upon the time of failure:

  1. Failure before the task execution

     The planner receives a failure alert during the phase planning and agent recruitment process (Period 1 in Figure 4.3). If the failure involves existing torch

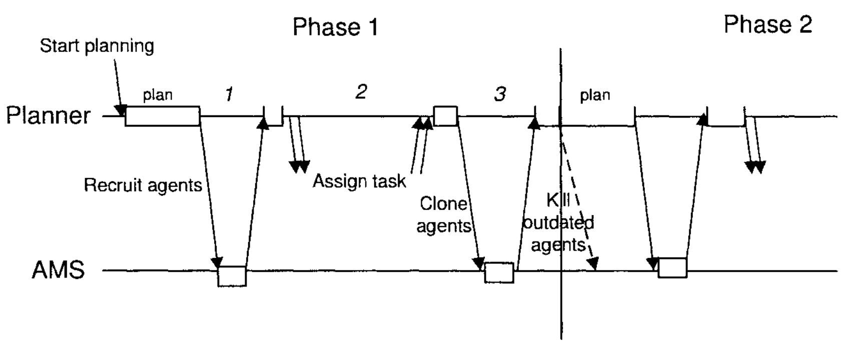


Figure 4.3: Planner execution model : Absence of failures

carrier agents then the current process is restarted with the clones of the failed ones. If clones are not available then the mission is restarted from the beginning since a part of the mission information is lost permanently.

2. Failure during task execution

   The planner receives a failure alert during the execution (Period *2* in Figure 4.3). The planner takes the necessary recovery actions depending upon the agents that failed. If one or more torch carrier and/or worker agent failed then the current task goes to an erroneous state. Hence, the phase is terminated and restarted with the existing clones and freshly recruited agents. The recovery involves recycling of agents that were part of the failed task. If some or none of the clones are available then certain information are lost permanently and mission cannot continue further. Therefore, it has to be restarted from the beginning. Figure 4.4 illustrates a similar recovery scenario.

3. Failure after task execution

   The planner receives failure while performing agent clone management (Period *3* of Figure 4.3). Once again the planner takes necessary recovery actions depending upon the type of agents that failed. If one or more torch carrier agents failed then knowledge regarding the completed task of the phase is permanently lost. Therefore, the entire phase is again re-executed with the existing clones.

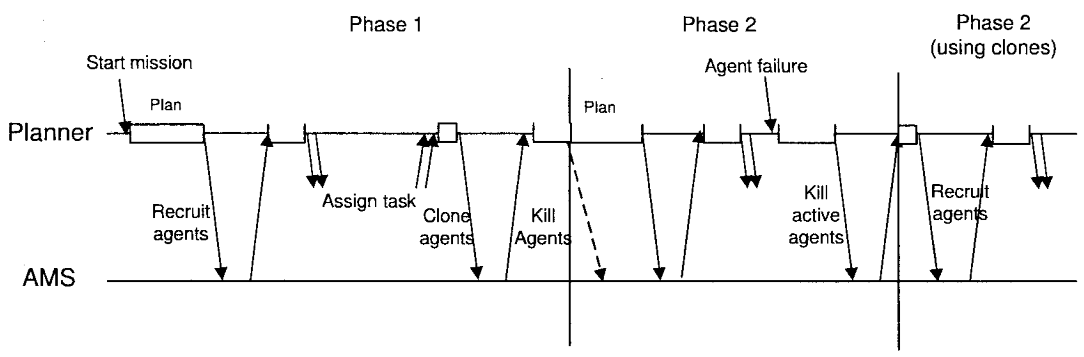


Figure 4.4: Planner execution model : Failure during task execution

If some or none of the clones are available then the entire mission is restarted.

4.1.2.2 Mission Reset

Mission that cannot afford the agent cloning use a simpler mission reset strategy. For instance, the task planner upon receiving a failure alert resets the mission state and starts again from the first phase with a new set of agents.

- Normal mode of operation

  This planner has the same knowledge as the default SPAF planner but with an additional responsibility of restarting the mission afresh upon a failure. Figure 4.5 shows the execution model of a planner in the absence of failures.

- Recovery mode of operation

  When a planner receives a failure alert from the failure detection service, it simply recycles all the remaining agents and restarts the mission afresh with a new set of agents. Figure 4.6 shows a sample mission recovery scenario.

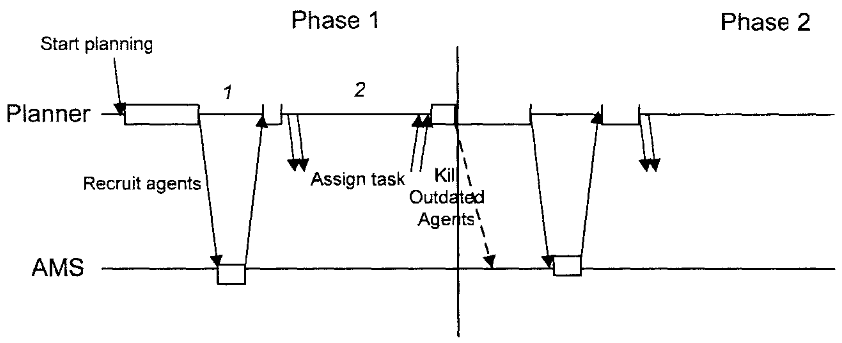


Figure 4.5: Planner execution model: Absence of failures

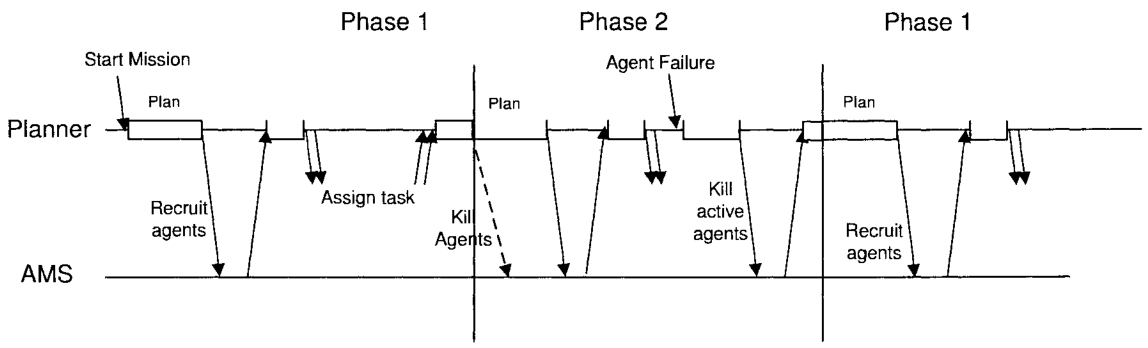


Figure 4.6: Planner execution model: Sample recovery scenario

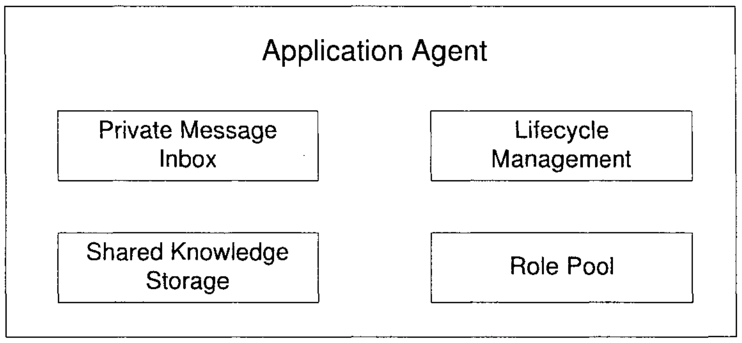


Figure 4.7: Agent Architecture

4.1.3 Application Agents

In order to make the above strategies effective, the SPAF application agent design is modified to include the following changes

1. All application agents are equally capable and can act as a torch carrier or a worker or a clone in any mission.
2. During the agent invocation it is sufficient to provide them with the role name, data parameters and other task related role parameters.
3. Since cloning is performed by agents, necessary infrastructure to retrieve and assign knowledge is available.

An application initialization involves creation of a pool of equally capable application agents in various application containers. Once the missions are initialized, the planners recruit the required number of agents from the agent pool by requesting the AMS. The planners performing agent cloning as part of mission requests the AMS to clone the specified agents. The AMS selects and instructs an agent residing in the different node to act as a clone of the specified agent. The cloning protocol is shown Figure 4.8

Since all the agents are equally capable, the cloning process is a mere transfer of state from one agent to the other. The AMS initiates this process by instructing an agent to act as a clone. The agent requests the state of the specified torch carrier agent, receives and assigns the state to itself. The clone agent is registered once the AMS receives its response.

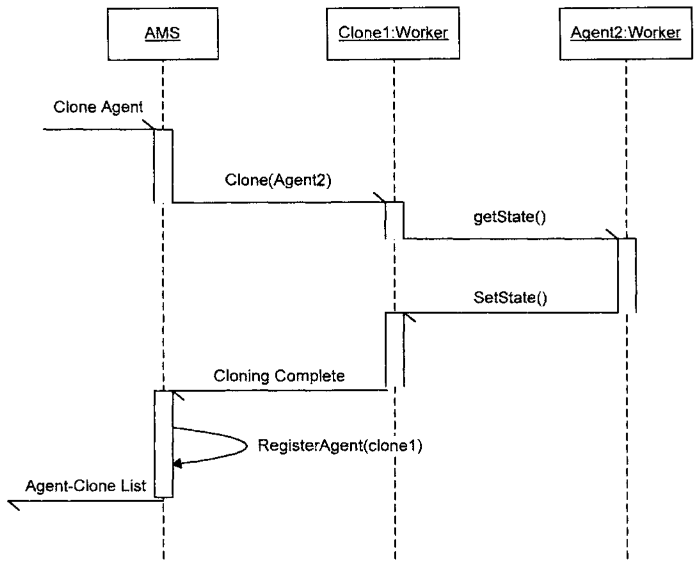


Figure 4.8: Agent Cloning protocol

4.2 Main Container Fault Tolerance

To incorporate our main container fault tolerance model, the system must be initialized with two main containers: primary and backup main container with replicated service agents. The system now has two services available at all times. The primary performs all the operations and updates the backup at events that change its state like agent registration and deregistration, mission registration and deregistration, planner events etc.

There are three main service agents that reside in the main container: AMS, MMS and manager agents. The service agent's design can be modified to accommodate the fault tolerance operations, such as, updating the backups, timeout and failure recovery etc. The backup container additionally has a fault detection service that exclusively monitors the liveliness of the primary container.

The above mentioned changes are elaborated in this section:

4.2.1 Fault Detection Service

This service is similar to that of the application container fault detection service. It detects failure of a main container using the periodic master-slave handshake mechanism. When a master detects the failure it takes appropriate actions depending on the type of container it is monitoring. If the primary main container fails then the master residing in the backup re-structures the agent platform and initializes all backup services from a stable state. If the backup main container fails then the master residing in the primary alerts all the services of the backup failure and stops further updates.

4.2.2 AMS

The AMS is primarily responsible for the agent management. All the state changes resulting from the AMS operations must be updated in the backup. Since a failure during an inconsistent primary-backup state may lead to an incorrect recovery, atomic updates must be guaranteed.

This subsection lists the events when the two AMSs must be consistent, describes in detail the primary-backup AMS commit protocol and is followed by the recovery protocols.

4.2.2.1 AMS Events

The list of events that affect the system state are:

1. Manager agent registration: When the AMS receives a request for creation of a new manager agent, it creates and initializes a new manager agent in its main container equipped with a mission planner. Upon initialization, the AMS makes a new entry to its agent registry.

2. Manager agent deregistration: When the AMS receives a request for de registration of an existing manager agent, it kills the manager agent and removes the corresponding entry from its agent registry.

3. Agent registration: On receiving application agent request from a manager, the AMS selects the required number of agent and add them to its agent registry. Now the AMS sends the list of agents to the requesting manager agent that can serve it.

4. Clone agent: When the AMS receives a clone request, it selects an agent and instructs it to acquire knowledge from the specified torch carrier. Once this operation is complete the selected agents is added to the agent registry.

5. Agent deregistration: When the AMS receives a request to de-register a set of agents/clones then it soft reset (clear knowledge) the agents and removes it from the agent registry. Once removed the agents wait for a next registration call.

During the events 1 and 2, the backup AMS must perform all the operations as its primary because changes involve local state. For example, manager agent registration requires a manager agent to be created in the main container.

During events 3,4 and 5, it would be sufficient to update the state of the agent registry as the changes involve the application containers. For example, agent de-registration involves reset of the agent state and must be performed only once.

4.2.2.2 Commit protocol

When the above mentioned events occur, the AMS completes the necessary operations and updates the backup AMS with an update message that contains the set of actions to be performed by the backup. The backup receives the update message, updates its own state by performing specified local operations such as, create/delete managers, add/remove entries

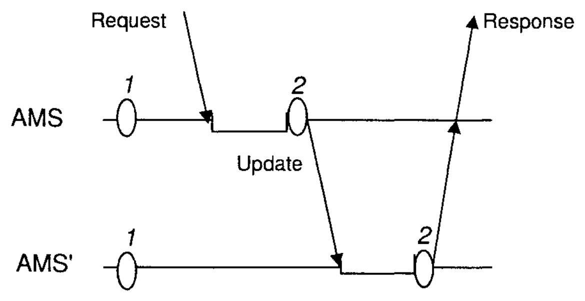


Figure 4.9: AMS primary-backup commit protocol

from registry etc. and responds with an acknowledgement to the primary. The primary responds to the client (MMS or Manager agents) only upon receiving the acknowledgement.The primary-backup AMS commit protocol is shown in the Figure 4.9.

4.2.2.3 Recovery Protocols

Listed below are the possible failure scenarios and corresponding recovery operations:

1. The primary fails during an operation:

   This scenario is shown in Figure 4.10. When the primary container crashes, the fault detection service detects the failure and alerts the backup AMS. The backup AMS now activates itself from the current stable state and waits for the future requests. If the primary fails when an operation involving AMS is underway then the application will re-send its request to the backup main container.

2. The primary fails after sending an update message:

   When a service request arrives at the primary AMS, it completes operation and sends an update to its backup. The primary responds to the client only upon receiving an acknowledgement from the backup. There is a rare possibility that the primary container fails while waiting for the acknowledgement from the backup, in other

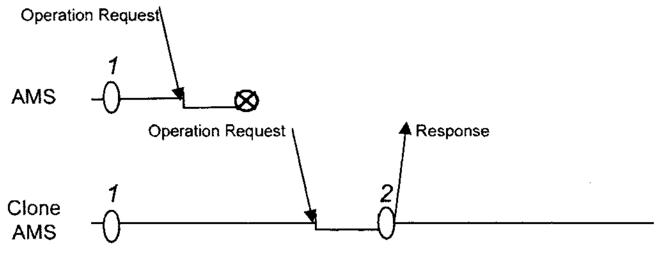


Figure 4.10: Failure Scenario 1: AMS recovery protocol

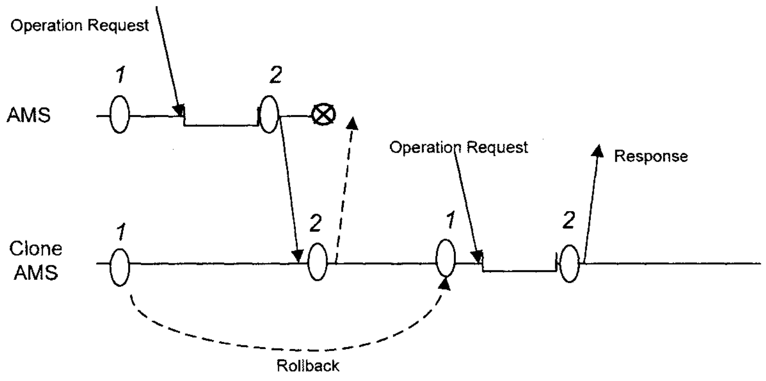


Figure 4.11: Failure Scenario 2: AMS recovery protocol

words, while the backup is performing an update operation. In such situation, even though the operation is complete the client will never receive confirmation because the client is not knowledgeable about the backup. The client will eventually re-send the operation to be performed. In such cases, the backup AMS must rollback to its previous checkpoint. Thus, the entire operation can be re-performed by the backup without any system conflicts. This situation is shown in the Figure 4.11.

4.2.3 MMS

The MMS is responsible for mission admission control and management. When an external request arrives it may have to interact with the AMS to complete the operation. This subsection lists the events when the two MMSs must be consistent, describes in detail the primary-backup AMS commit protocol and is followed by the recovery protocols.

4.2.3.1 MMS Events

1. Create mission: When a request to create mission arrives at the MMS, it first performs the time consistency checks. If the test is successful then the mission is admitted into the system and an entry into the mission registry is made.

2. Initialize mission instance: Upon a request to initialize an instance of an existing mission, the MMS checks for a mission name consistency.If the test is successful then MMS interacts with the AMS to create a new manager for the specified mission. Once the manager is initialized the MMS records the mission instance ID to its registry and sends an acknowledgement to the application.

3. Delete mission instance: When a request to kill an existing mission arrives from the application, the MMS interacts with the AMS and removes the Manager from the system. On completion the MMS removes the entry of the mission instance from its registry and acknowledges the application upon successful completion.

4. Delete Mission: When a request to delete a mission arrives at MMS then it checks if any instance of the mission is currently running in the system. If no instances are running then the application removes the mission entry from its registry.

During the events 2 and 3 the MMS must instruct its backup to perform certain actions like create or delete manager along with the state update. The events 1 and 4 requires mere

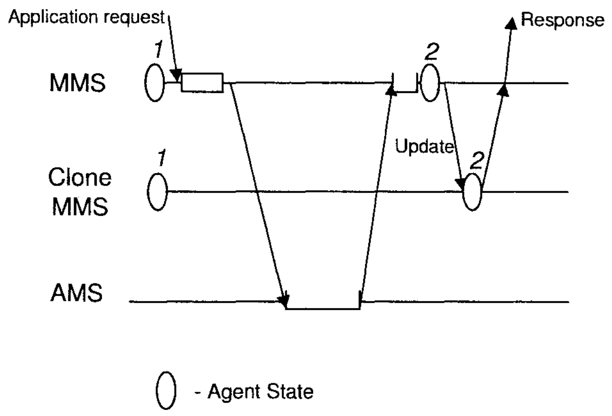


Figure 4.12: MMS primary-backup protocol

updates of MMS registry snapshot.

4.2.3.2 Primary-backup commit protocol

The primary-backup MMS commit protocol is similar to that of the AMS. When an external request arrives, the primary MMS processes it and updates the backup MMS with necessary changes to be made. The backup performs the specified actions locally and responds with an acknowledgement to the primary. The primary responds to the client only upon receiving the acknowledgement from the backup.

4.2.3.3 Recovery Protocols

The failure scenarios and corresponding recovery strategies are described below:

1. The primary fails during an operation:

   When the primary MMS is processing an external request and the main container crashes, the fault detection service detects the failure and activates its backup MMS for processing future requests. The backup can expect to receive the same request as previous operation was not completed. Figure 4.13 and 4.14 shows two different failure scenarios.

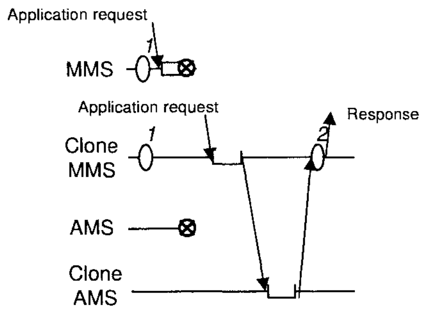


Figure 4.13: Failure Scenario 1: MMS recovery protocol

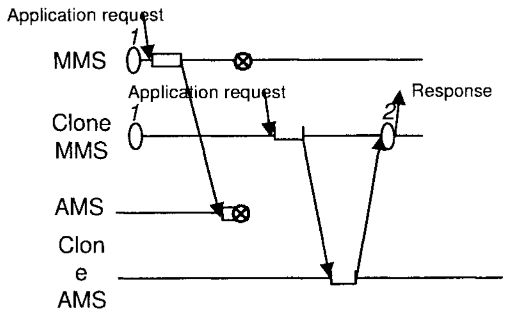


Figure 4.14: Failure Scenario 2: MMS recovery protocol

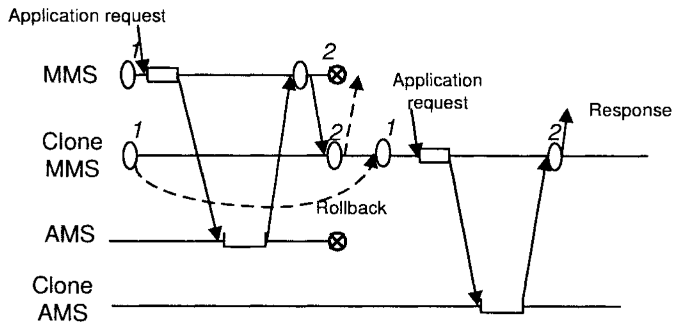


Figure 4.15: Failure Scenario 3: MMS recovery protocol

2. The primary fails after sending an update:

   The primary MMS fails just after sending the backup a state update as shown in Figure 4.15. When the failure detector detects this failure, the MMS rolls back to a previous stable state where an operation was successfully completed. This roll back may involve undo of operations mission registration and mission creation during the last transaction. Only upon successful rollback of all the services does the main container begins to process regular requests.

4.2.4 Manager Agent

Task planner of the manager agent is an important component of the mission. This subsection lists the events when the two managers must be consistent, describes in detail the primary-backup manager commit protocol and is followed by the recovery protocols.

4.2.4.1 Planner Events

A planner has the following significant events during its life time:

1. Mission execution start: At every mission start event the manager agent invokes its planner.

2. Agent recruitment: The planner performs task planning and recruits a set of agents from the AMS to execute the chosen solution. Once agents are recruited it signals

the agents to start the execution.

3. Task completion: Upon receiving the responses from all the agents the planner declares successful execution of the task.

4. Agent cloning and killing outdated agents: Once the task of a phase is completed, the planner performs agent cloning and kills the outdated agents and clones.

5. Phase restart: If a planner capable of performing agent cloning encounters an application container failure then it restarts the phase with the available clone agents. All the remaining agents of the mission are killed.

6. Mission reset : A mission restart involves killing all the agents of the mission and restarting it in the cheap planning mode. After restarting the planner recruits a new set of agents to execute the mission.

7. Mission termination: When a mission execution completes or stops the manager updates its backup and waits for the next start event to occur.

4.2.4.2 Commit Protocol

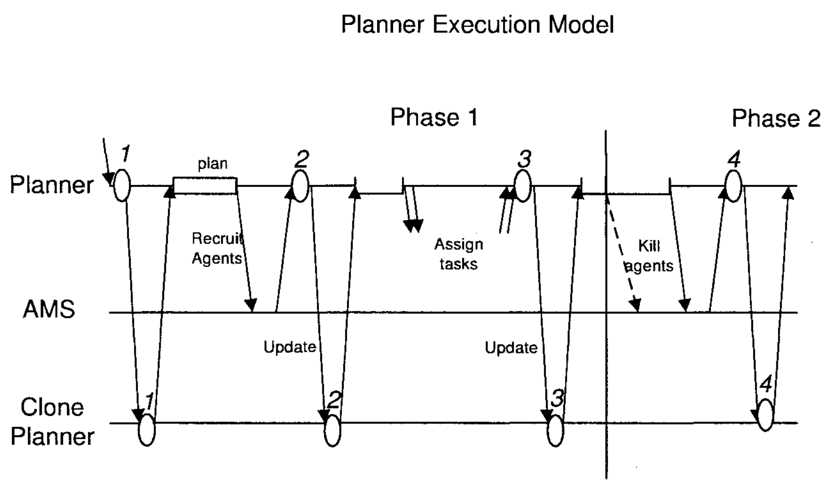
When the above events occur, the planner executes the operations locally and sends the state update to its backup. The backup planner upon receiving the update locally performs only a local state update. The update contains the recent snapshot of the data structures RoleAgentMap and AgentCloneMap. Once a response form the backup is received the planner proceeds forward. Figure4.16 shows a sample scenario.

4.2.4.3 Recovery Protocol

The possible failure scenarios and the corresponding recovery strategies are described below:

1. The main container failure encountered before the task execution. This can lead to one of the following cases:

    (a) The task planner crashes; all the torch carriers and associated clone agents are alive: As shown in Figure 4.17, task planner restarts planning with the most recent state and executes the tasks with available torch carrier agents and newly recruited agents.



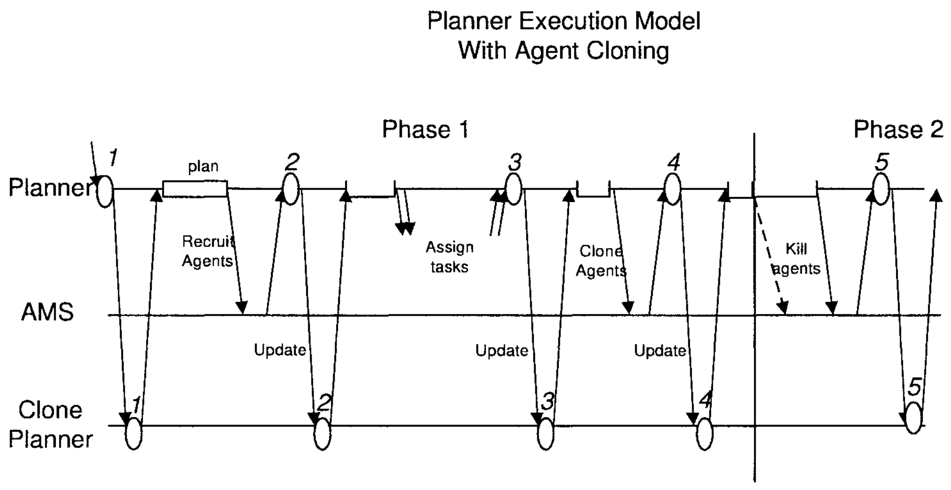


Figure 4.16: Planner primary-backup commit protocols

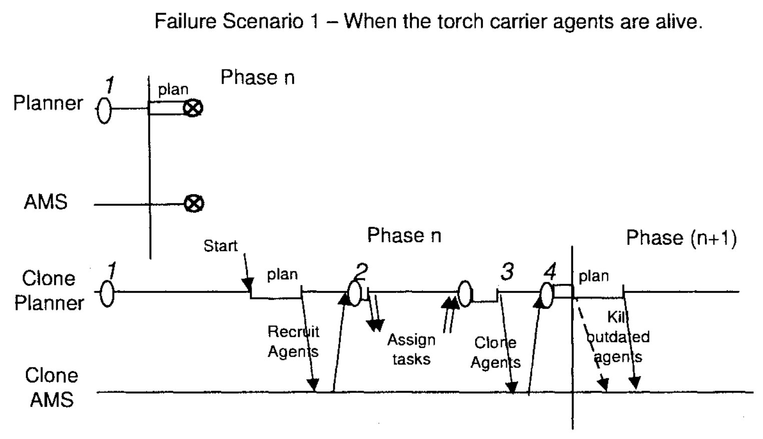


Figure 4.17: Planner recovery protocol

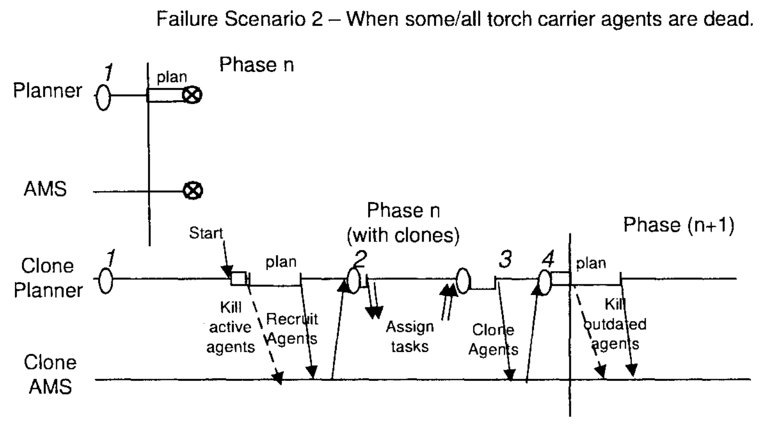


Figure 4.18: Planner recovery protocol

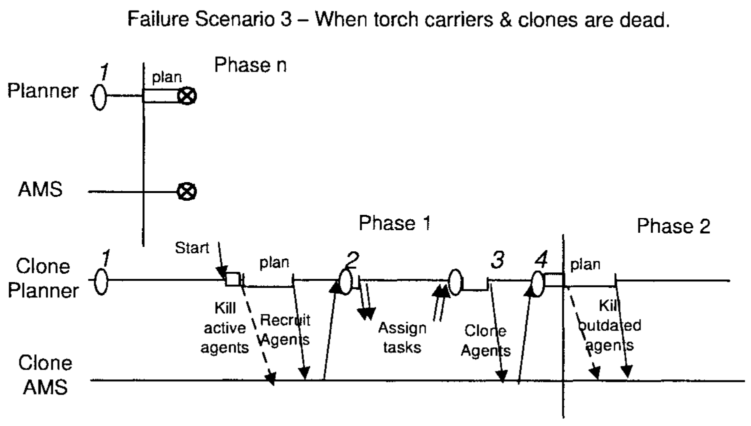


Figure 4.19: Planner recovery protocol

(b) The task planner crashes along with some torch carriers. In this case, the backup planner can start the recovery process by killing rest of the torch carriers and restarting the planning process with clone agents and newly recruited agents. This situation is shown in Figure 4.18

(c) The task planner crashes along with some torch carriers and clone agents. In this case, the planner has permanently lost some mission information and will not be able to continue further. Hence, it has to restart the entire mission using freshly recruited agents as shown in Figure 4.19

2. The task planner crashes during the middle of task execution: The backup planner, upon initialization, re-starts the phase with the available clones (if any) as the states of the actual agents executing the tasks are completely unknown. If the current failure is accompanied with application agents' failure then the recovery process depends upon the failed agents as discussed in cases 1(b) and 1(c). If a worker agent and/or torch carrier agent has failed then the mission will be restarted by default with the available clones. If some torch carriers and their clones have failed then the entire mission has to be restarted as the mission states are permanently lost. The normal planner crash recovery scenario is shown in Figure 4.20.

3. The task planner fails after the task execution: This situation is applicable only for

Failure Scenario 4 : Failure During task execution

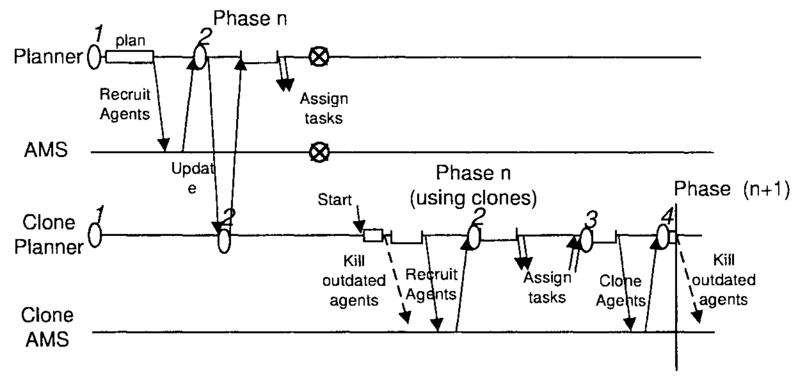


Figure 4.20: Planner recovery protocol

Failure Scenario 5 : Failure after task execution

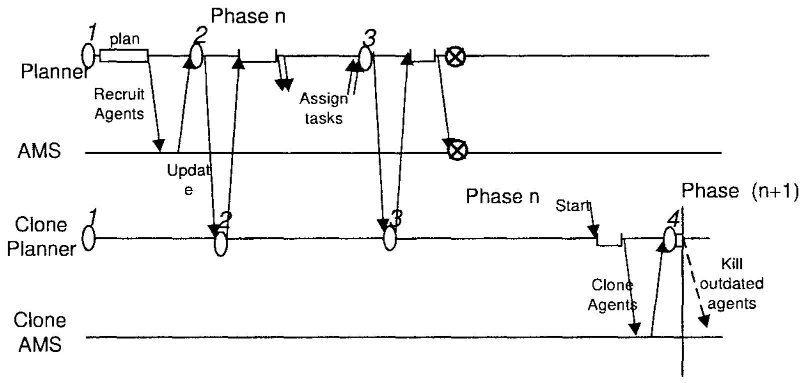


Figure 4.21: Planner recovery protocol

the task planners performing cloning at the end of a phase. When the primary main container fails the backup task planner can simply re-perform the cloning and agent management process upon initialization. Again, if the planner failure is accompanied by a torch carrier agent failure then the entire phase must be re-performed. A sample recovery strategy is shown in Figure 4.21

All the above cases deals with the failure of primary during the operation part of the commit protocol. Similar to other service agents, it is possible that the crash can occur during a backup update process. This situation is handled in a slightly different manner. The backup upon detecting a failure can proceed further with its current state as rollback is clearly not required in this case.

4.2.5 Other Services

The main container has platform services such as container registration or de-registration. The container connection related information is stored and maintained in a container registry. When a new container is added to the system the container table is updated with the container name and the connection parameters. When a container is shut down the related container information is deleted from the container registry. Absolute consistency of the container registry must be ensured at the primary and backup containers. The primary-backup protocol involves transfer of the container registry whenever a container registration or de-registration takes place.

4.2.6 System Interface

The application requires the services of the system to manage its missions. The system provides an interface residing outside the agent world in the application's operation environment, to receive and send the system service requests. This interface plays a very important role in fault tolerance. When the primary system services fail then the interface, with the use of a timeout mechanism, detects the failure and finds the backup service to complete the operation.

4.3 Summary

In this chapter, the design of the node crash tolerance mechanisms has been presented. In the first section, a simple periodic handshake based application container failure detection mechanism is presented and followed by that the modifications to the generic planner and application agent model to accomplish application container fault tolerance is described. The primary-backup commit protocol and the mission recovery protocols are explained.

Similarly, the main container failure detection mechanism and modifications to the main container services to accomplish the main container fault tolerance is presented in the later section. The primary-backup commit protocol and recovery protocols pertaining to the important services are explained.

Chapter 5

Performance Analysis

In the previous chapters, we discussed the model and design of our fault tolerance mechanisms. In this chapter, we try to the observe effects of incorporating fault tolerance on the application through a set of experiments. At first, we try to determine the efficiency of our fault tolerance mechanisms and later we conduct tests to measure the overhead and performance effects on the application.

5.1 Failure Recovery Tests

In this section, we try to measure the efficiency of our fault recovery mechanisms. The application and main container recovery times are measured by the injecting node crashes.

5.1.1 Application Container Failure Test

In order to study the efficiency of the recovery mechanisms in the presence of a failure, we consider an application with 20 missions. To simulate a real world application, we have a mixture of periodic and sporadic missions. All the missions are equipped with planners capable of performing agent cloning during the mission execution. The application agents are distributed over 5 application containers running on 4 different hosts. We inject faults in a manner that it affects the required number of missions and we measure the recovery time of the application. The application recovery time is the time taken to recover all the failed missions.

As seen in Figure 5.1 the recovery time of the application differs only by a small value (25ms) irrespective of the number of missions that failed. The difference in the recovery

No of Failed Missions	Recovery Time
1	75
2	79
3	77
4	85
5	81
6	86
7	79
8	89
9	94
10	87
11	80
12	93
13	95
14	92
15	91

Table 1: Application Container Recovery Time

time is mainly due to the single AMS processing more requests as the number of missions increase. The obtained result is mainly due to an efficient recovery strategy and the corresponding system design changes.

5.1.2 Main Container Failure Test

In this section we will consider the recovery of the system and the application in the presence of a primary main container failure. In order to determine the effects of the application

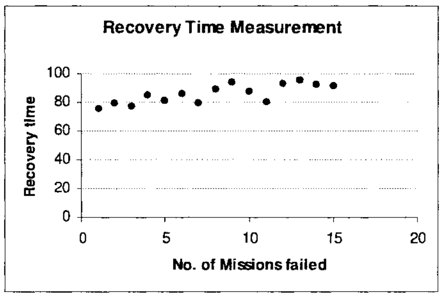


Figure 5.1: Application Container Recovery Time

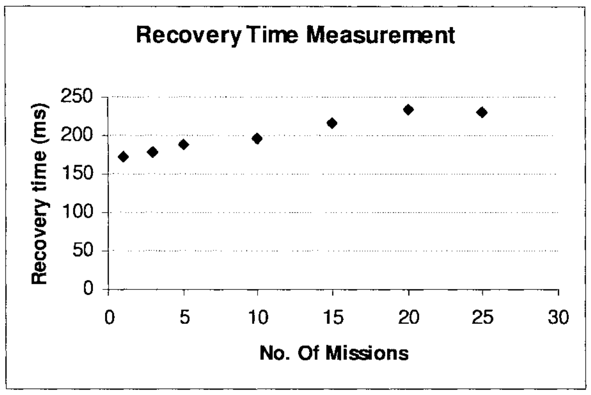


Figure 5.2: Main Container Recovery Time

on the main container recovery mechanism, we conduct this test with different application configurations by varying the number of missions running. Main container failures are injected in each of the above configuration and recovery time of the application is measured. Here the recovery time is the sum of time required to recover the system components and all application missions. The Table 2 shows the average recovery times recorded for different mission configurations.

No of Failed Missions	Recovery Time
1	172
3	178
5	189
10	196
15	216
20	234
25	230

Table 2: Main Container Recovery Time

As seen in the figure 5.2, the time taken to recover the application increases very small as the number of missions increase confirming a very low dependency between the application and the fault-tolerance. Also the time taken to recover the main container is higher than that of the application container failure because of the time taken to recover the system and all the application missions.

	10 Missions	15 Missions	25 Missions
No FT	9848	10227	10757
Main Container FT	10165	10846	11508
Main Container FT and Application Container FT (Selected Missions)	10514	11115	12195
Main Container FT and Application Container FT	10735	12484	13615

Table 3: Percentage of Completion Table

5.2 Overhead Measurement

In this section, fault-tolerance overhead on the application is measured. This test measures the overhead in terms of the average completion time of the missions of a customized application run on different system configurations.

In order to measure the overhead we consider to run an application with different number of missions on systems with and without fault tolerance. The mission has a deadline of 15 seconds and always executes one solution with an acceptable quality. The mission requires 8 agents to complete all its tasks. Table 3 shows the average completion times of the missions running on the 4 different system configurations.

From Figure 5.3 it is seen that, as the number of missions increase the completion time increases slightly. The increase in the execution time is mainly due to the message overhead as a result of increase in the number of missions. As more missions are admitted in the system, the cost to maintain a hot backup also increases accordingly.

5.3 Application Performance

In this section, we test the effects of the fault tolerance on the application performance. To simulate a real world application, we have missions with different levels of tightness in deadlines (tight, medium and loose) and in equal proportion. We consider 18 missions, 6 missions in each of the above mentioned category. We run the application on 4 different system configurations: 1. with no fault tolerance, 2. with main container fault tolerance, 3. main container and application container fault tolerance on selected missions only and 4.

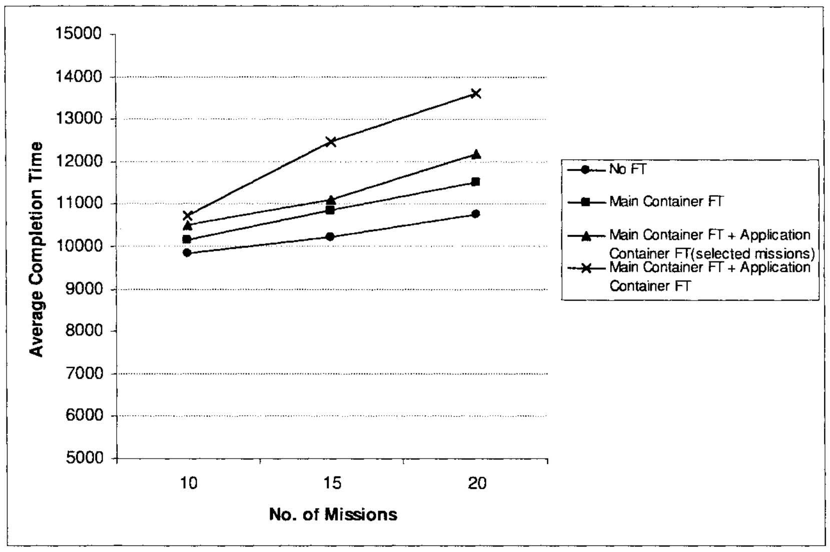


Figure 5.3: Overhead Measurement

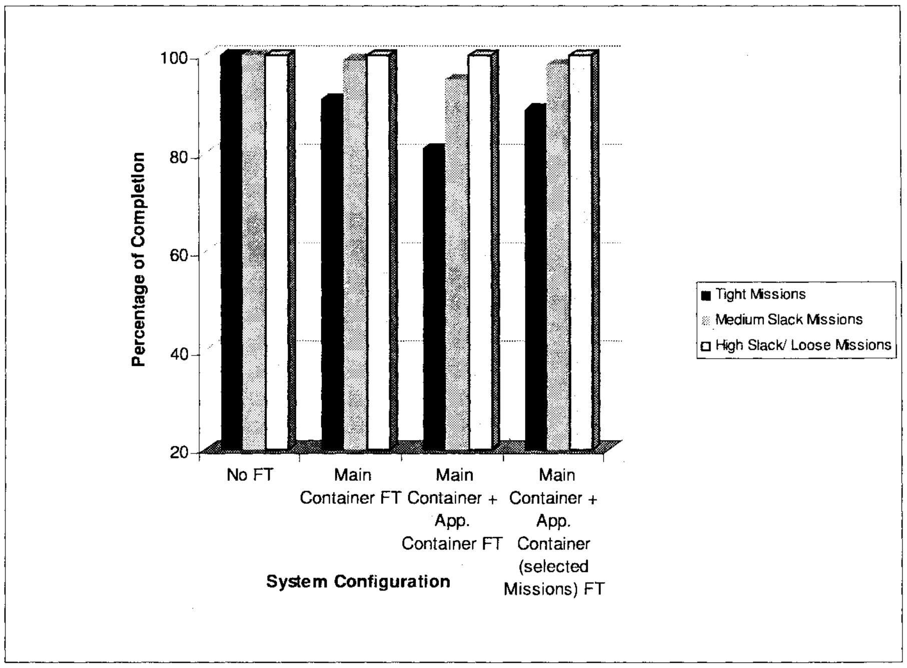


Figure 5.4: Application Performance

	Tight Deadline Missions	Medium Slack Deadline Missions	Loose Deadline Missions
No FT	100	100	100
Main Container FT	91	99	100
Main Container FT and Application Container FT (Selected Missions)	81	95	100
Main Container FT and Application Container FT	89	98	100

Table 4: Percentage of Completion Table

main container and application container fault tolerance. Table 4gives the system configuration and the corresponding mission failure rate.

As shown in Figure 5.4, the application performance improves when only a selected number of missions are provided with agent cloning. And it was also found that the failures were mainly of the missions with the tight deadlines.

5.4 Summary

In this chapter, performance of the fault tolerance mechanisms are tested using a customized SPAF application. The application container and main container failure recovery scenarios are simulated first and results show that the failure recovery mechanisms are effective and the application recovery time is reasonable. In the next two sections, the fault tolerance overhead and performance impact on the application is measured. The results confirm that the fault tolerance overhead is fair and the overall performance impact on the application upon the inclusion fault tolerance is very small.

Chapter 6

Conclusion

In this thesis, an application-transparent node-crash tolerance mechanism for a soft real-time self-planned agent framework is presented. Fault tolerance is incorporated without changing the actual functionalities of the system components. By application-transparency, we mean that the API of SPAF remains the same even after inclusion of fault tolerance. Hence, an application developed for SPAF would successfully run on the modified system with minimal performance difference, even in the presence of node crashes.

Fault tolerance is achieved by conventional primary-backup approach in conjunction with flexible task planning provided by SPAF and a periodic handshake based node crash detection mechanism. The application fault tolerance is accomplished maintaining a cold backup of the missions in a different location. A novel agent cloning technique is used to significantly improve the cold backup and recovery process in a soft real-time system. The system fault tolerance is accomplished maintaining a hot backup of all the services in a different location. For the missions that are unable to afford the time overhead caused by the maintenance of a cold backup, a simple mission reset strategy is used. This strategy, unlike the cold backup, does not have any backup maintenance overhead during the mission, rather chooses to reset the mission completely upon a failure. Fault tolerance strategy for a mission is appropriately chosen during the admission time.

The practicality of the framework is tested using a custom built SPAF application. To simulate real world applications, tests were conducted with varying number of missions and deadlines. The recovery time of the application is almost constant due to an efficient failure recovery mechanism. In the absence of failures, the overhead offered is mainly due to the primary backup update messages and increases as the number of missions increase.

This increase in the overhead is mainly due to the serial processing of requests by a single agent management system. As the number of missions increases, the number of requests increases. The performance tests show that the application performance is almost the same even after inclusion of the fault tolerance mechanisms. This is due to the flexible choice in the fault tolerance strategies for different kinds of missions.

Ongoing research work includes improvement of the application mission recovery using selective rollback instead of a complete rollback during a node failure. When more than one disjoint agent group works on a task then the agent crash need not necessarily involve a complete rollback. Ongoing work also includes implementation and testing of the node crash tolerant SPAF using a real-world application

Bibliography

[AHIL78] Jr. A.L. Hopkins, T.B. Smith III, and J.H. Lala. Ftmp - a highly reliable fault-tolerant aircraft control. *Proceedings of IEEE*, 30(4):1221–1239, oct 1978.

[AOSM05] R. Al-Omari, A.K. So, and G. Manimaran. An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems. *ACM Journal of Parallel and Distributed Computing*, 65(5):595 – 608, May 2005.

[BB03] P. Garbacki B. Biskupski. *Transparent Fault Tolerance for Parallel Java Applications.* Master's Thesis, July 2003.

[BH98] V. Botti and L. Hernndez. Control in real-time multiagent systems. In *Proceedings of Second Iberoamerican workshop on DAU and MAS*, pages 137–148, 1998.

[BN93] Budhiraja and Navin. *The primary backup approach in Distributed systems, 2 ed.* Mullender (ed.), ACM Press, 1993.

[BRNB] Mark A. Breland, Steven A. Rogers, Kenneth L. Nelson, and Guillaume P. Brat. Transparent fault tolerance for distributed ada applications.

[FIP00a] FIPA. Fipa acl message structure specification,xc000261. *Available online at http://www.fipa.org/specs/fipa00061/*, 2000.

[FIP00b] FIPA. Fipa agent management specification,xc000231. *Available online at http://www.fipa.org/specs/fipa00023/*, 2000.

[GBB05] P. Garbacki, B. Biskupski, and H. Bal. Transparent fault tolerance for grid applications. *In Proceedings of the European Grid Conference (EGC2005), Amsterdam, The Netherlands*, February 2005.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[HGNG03] Huang-Ming Huang, Christopher Gill, Bala Natarajan, and Aniruddha Gokhale. Replication strategies for fault-tolerant real-time corba services. In *Proceedings of OMG Workshops, Needham, MA, U.S.A.*, 2003.

[HLVW05] Bryan Horling, Victor Lesser, Regis Vincent, and Thomas Wagner. The Soft Real-Time Agent Control Architecture. *Autonomous Agents and Multi-Agent Systems*, 2005.

[Jad05] Jade. Java agent development environment. *Available online at http://jade.cselt.it/*, 2005.

[Jin05] De Jin. *A Soft Real-Time Self-Planned Multi-Agent Framework, Masters Thesis*. Concordia University, Montreal, 2005.

[Joh89] B.W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley Pub. Co. Inc., 1989.

[KBS91] D. Krieger, G. Burk, and R.J. Sclabassi. Neuronet: A distributed real time system for monitoring neurophysiologic function in the medical enviroment. *IEEE Computer*, 24(3):45–55, Mar 1991.

[KRK03] A. Kassler, B. Reiterer, and J. Kaiser. A framework for scheduling soft real-time multimedia applications. *Internet and Multimedia Systems and Applications (IMSA) 2003, Honolulu, USA*, 2003.

[LKR02] Kam-Yiu Lam, Alan Kwan, and Krithi Ramamritham. Rtmonitor: Real-time data monitoring using mobile agent technologies. In *Proceedings of the 28th VLDB Conference*, 2002.

[LV62] R.E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal*, pages 200–209, April 1962.

[LWS94] Jonathan Litt, Edmond Wong, and Donald L. Simon. A prototype lisp-based soft real-time object-oriented graphical user interface for control system development. *GLTRS Technical report, E-9155*, Oct 1994.

[MK86] L. Mancini and M. Koutny. Formal specification of n-modular redundancy. *Proceedings of the 1986 ACM fourteenth annual conference on Computer science*, pages 199 – 204, 1986.

[MSM04] P.M. Melliar-Smith and L.E. Moser. Progress in real-time fault tolerance. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, Oct 2004.

[Pra86] D.K. Pradhan. *Fault Tolerant Computing: Theory and Techniques*. Prentice Hall, NJ, 1986.

[RT93] S. Ramol-Thuel. *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*. Phd Thesis, Carnegie Melon University, 1993.

[SD02] Riddhi Burman Mridul Sankar Barik Chandan Mazumdar Surajit Dutta, Sudip Dutta. Design and implementation of a soft real time fault tolerant system. *Proceedings of the 4th International Workshop on Distributed Computing, Mobile and Wireless Computing*, 2571:319 – 328, 2002.

[SKAS] Seejo Sebastine, Kyoung-Don Kang, Tarek F. Abdelzaher, and Sang H. Son. A scalable web-based real-time information distribution service for industrial applications. In *The 27th Annual Conference of the IEEE Industrial Electronics Society, Denver, Colorado, USA*, pages 1810–1815.

[Skl76] J.R. Sklaroff. Redundancy management techniques for space shuttle computers. *IBM Journal of Research and Development*, pages 20–28, Jan 1976.

[SR91] J.A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[SS99] S. Subramanian and M. Singhal. A real-time protocol for stock market transactions. In *In proceedings of International Workshop on Advance Issues of E-Commerce and Web-based Information Systems*, April 1999.

[SSK] Y. Weiand S. H. Son, J. A. Stankovic, and K. D. Kang. Qos management in distributed real-time databases. In *24th IEEE Real-Time Systems Symposium (RTSS '03), Cancun, Mexico,*, pages 86–97.

[T.B94] T.Becker. Application-transparent fault tolerance in distributed systems. *In Proc. of the Second International Workshop in Configurable Distributed Systems*, May 1994.

[TG02] Chiu-Che Tseng and Piotr J. Gmytrasiewicz. Real time decision support systems for portfolio management. In *Proceedings of the 35th Hawaii International Conference on System Sciences, HICSS-35*, 2002.

[TK91] J.M. Long T.J. Kriewall. Guest editor's introdution: Computer based medical systems. *IEEE Computer*, 24(3):9–12, May 1991.

[Vla03] N. Vlassis. A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam, September 2003. http://www.science.uva.nl/~vlassis/cimasdai.

[Woo02] M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley and Sons, UK, 2002.

[YC98] Zhang Y and Tsien CL. A data collection and data processing system to support real-time trials of intelligent alarm algorithms. In *In proceedings of AMIA Symposium*, 1998.

[Ziq03] Liao Ziqi. Real-time taxi dispatching using global positioning systems. *Communications of the ACM*, 46(5):81–83, May 2003.