A Fault-Tolerant Multi-Agent Development Framework

Lin Wang

A Thesis

in

The Department

Of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2004

Your file Votre référence
ISBN: 0-612-94754-8
Our file Notre référence
ISBN: 0-612-94754-8

ABSTRACT

A Fault-Tolerant Multi-Agent Development Framework

Lin Wang

Application-level fault tolerance incurs significant development-time cost. FATMAD is a fault-tolerant multi-agent development framework that is built on top of a mobile agent platform (Jade) and provides development support for application level fault tolerance. FATMAD aims to satisfy the needs of two communities of users: agent application developers and fault-tolerant protocol developers. FATMAD is based on a generic fault-tolerant protocol whose refinements lead to a broad range of checkpoint and recovery protocols to be used in user applications. This, in turn, can significantly reduce the development time of fault-tolerant agent applications. FATMAD allows application developers to apply suitable protocols and customizable deployment configurations to their applications so that the characteristics and fault tolerance requirements of a particular application can be addressed. FATMAD implements common facilities that are required by different protocols. The protocol-specific parts in each protocol can be extended by protocol designers.

Acknowledgements

I extend my sincere gratitude and appreciation to many people who made this master thesis possible. Special thanks are due to my supervisor Dr. Dhrubajyoti Goswami and Professor Hon F. Li. Without their help this thesis cannot be done properly. I am deeply indebted to Prof. Hon F. Li whose help, stimulating suggestions and encouragement helped me in all the time of research.

I would also like to acknowledge with much appreciation my colleagues in the distributed research group for all their help, support, interest and valuable hints. Special thanks go to Zunce Wei who spared his time in contributing his idea and revising a paper related to this work.

Especially, I would like to give my special thanks to my wife Jian and my parents whose patient love enabled me to complete this work.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

1.1.1 Maturity of agent oriented software engineering

The discipline of agent-oriented software engineering (AOSE) has emerged during the last decade and the potential advantages of agent paradigm have been recognized by both scientific and industry communities. The agent paradigm have been explored and applied in many application areas, such as bioinformatics, e-commerce, supply-chain management, semantic web, etc [Cole98] [Preist99] [PiGa00] [MaFu01] [Mols02] [VeCo02] [KMMK+03] [AGJ04] [DiWo03].

The distributed agent community has done a lot of research and development work to promote AOSE. Many methodologies and modeling techniques have been suggested to support the development process of agent-based systems [ShSt01] [JSMM03] [OdPB00] [SaSh00] [ArLa98]. Some software standards, such as FIPA ACL [FIPA97] and KQML [FiLa97] have been established to promote collaboration and interaction among heterogeneous agent systems [KSTV+01]. Moreover, many development frameworks and agent platforms have been provided to support agent-based system development and deployment. However, the number of deployed commercial agent-based applications is not large. The research and practice on agent-based technology are not mature enough yet and require contribution from different aspects.

Agents are active objects [Liu01] [ArLa98] presenting purposeful and also autonomous behaviours at runtime. Tolerating failures in an agent application is a non-trivial problem for agent developers and system designers. Along with the evolution of agent-based system, adapting fault tolerance techniques into agent-based system becomes an important work that will contribute the maturity of Agent based system development.

1.1.2 Adaptation of fault tolerance techniques

Dependability is one of the most desirable features for all kinds of systems. Fault tolerance, as one of essential techniques to enhance system dependability, has been explored for many years. A lot of research effort has been put into fault tolerance research and a lot of fruitful research results have been delivered [Jalote94] [EAWJ02]. However, there is only a quite small subset of them that ever have been put into practice. The reasons might be the following:

- some techniques might not fit into all application domain
- some techniques might be too complicated to be applied efficiently

Therefore, adapting fault tolerance research work into practice is a valuable effort that could make them become realizable.

1.1.3 Developing fault tolerance feature is a complex job

As all distributed system, agent-based systems are prone to failures. Developing fault tolerant features for agent-based systems is a complex and difficult job due to the following reasons:

1) Different failures have different characteristics. System developers need to distinguish them and address them in suitable ways.

2) Distributed systems involve much more complexity than centralized systems, because the coupling relationships among distributed components usually lead to extra coordination efforts in fault tolerance design. Distributed coordination is usually complex since there is no unified global clock to easily support this coordination.

3) Fault tolerance features produce overhead. The characteristics of an application are required to be considered very carefully in selecting and incorporating suitable fault tolerance technique so that overhead may be minimized.

4) There are different fault tolerance strategies, each of which has different characteristics, assumptions, suitability, and performance.

5) Incorporating fault tolerance techniques in applications requires very careful design and implementation to guarantee that the added features do not conflict with the application and do not bring about any new problem.

Consequently, we can see that designing a fault tolerance feature for a distributed system requires designers to have deep understanding and knowledge of distributed fault tolerance techniques, which are usually not possessed by main-stream application developers.

Therefore, support for fault tolerance can be very helpful in developing fault tolerant applications.

1.1.4 Fault tolerance support from existing agent platforms

There are a lot of multi-agent platforms and frameworks that have been developed to enhance development and deployment of multi-agent system [AGKSW01], such as IBM Aglet, Concordia, Grasshopper, Zeus, Jade, FIPA OS, etc. All these platforms provide system level services to support agent execution at runtime, such as agent management, naming services, agent communication, etc. These systems also provide different levels of support in designing and developing agent-based applications. However, fault tolerance features are not commonly available in existing platforms.

From the existing published works [WaPW98] [SSS+99] [BePR99] [Jade], we can see that fault tolerance support at the system level has been explored in some agent platforms, such as Concordia, James, Jade, etc.

The Concordia [WaPW98] system framework developed by Mitsubishi Electric Research Lab mainly focuses on agent mobility support. It implements a "store (checkpoint) and forward" queuing mechanism and a two phase commit protocol to ensure atomic agent migration between nodes. Its checkpoint and recovery scheme is partially implemented since it takes no consideration on agents running extensively on one node as well as agent communication. Although its persistence storage service can be used to implement checkpoint and recovery schemes, this leaves the burden to application developers.

James [SSS+99] is a mobile agent platform that implements a rich set of reliability and fault tolerance services, such as resource control, checkpoint/recovery, reconfigurable itinerary, etc. Similar to Concordia, its fault tolerance scheme focuses on agent mobility. We notice that James' application level checkpoint strategy is based on each individual agent and it takes no consideration on multi-agent collaboration and communication. This means, a recovered agent will possibly be inconsistent with its environment agents due to lost messages.

Jade [BePR99] [Jade] developed by TILAB, is a multi-agent platform and agent development framework that gains increasing popularity. As compared to the previous two frameworks, besides providing agent mobility support, Jade provides comprehensive support for high-level agent interaction. It implements the FIPA agent specification proposed by the FIPA organization [FIPA], which aims at developing standards to enhance interoperability among heterogeneous agent systems.

Starting from version 3.1, Jade provides two fault tolerance features at system level.

- Reliable message transmission:

  By implementing a persistent delivery mechanism, Jade tries to provide a reliable message transmission service.

- System level replication

  In Jade, agents run within agent containers at different nodes (machines). A Jade platform maintains all the platform management functionalities in a special agent container, i.e. main container. By employing replication techniques on this main container, the platform can survive crash failure. With this facility, we are allowed to run several main containers (nodes) that provide system management services in a distributed Jade platform. As long as there is more than one main container alive, the whole platform remains functional.

However, Jade doesn't provide any scheme on application (agent) level fault tolerance.

In general, we notice that existing fault tolerance enabled systems mainly focus on system level fault tolerance. Application level fault tolerance support is very weak. Agent communication as an important factor in multi-agent system is rarely taken into consideration in these systems.

Therefore, to develop an effective fault tolerant multi-agent application, a developer has to design and implement suitable schemes from scratch.

1.2 Objectives

With the consideration of application level fault tolerance development support, this thesis intends to develop a fault-tolerant multi-agent development framework, which is abbreviated as FATMAD, to pursue the following objectives:

1) Provide support for application developers in developing fault tolerant multi-agent applications.

With the support from FATMAD, the application developer should be able to easily incorporate a specific fault tolerance scheme into his/her agent application so that the agent application can be fault tolerant at runtime.

Considering various requirements from application developers, the framework should provide the following flexibilities:

- The application developer can decide fault tolerance design boundary. Since fault tolerance is usually implemented at the expense of performance, the application developer has the flexibility to implement fault tolerance features within a smaller domain if necessary.
- Since various applications may have different characteristics that lead to different requirements on fault tolerance feature design, FATMAD should provide choices in helping the application developer implement different schemes. With a set of available choices, the application developer can compare and select a fault tolerance scheme that is suitable for the application.

2) Provide support for protocol designers so that they can easily test different fault tolerance schemes and extend the framework by enriching the protocol set.

There are different fault tolerance strategies, such as replication, rollback-recovery. Under each strategy, there are various protocols, each of which has its own characteristics that differ from others, such as assumptions, suitability of application types, performance, etc. Besides the built-in protocols, protocol designers can use the framework to design and test new protocols with minimum implementation effort. The resulting protocol can become part of the framework and be used by application developers.

In general, unlike some existing systems providing system level support, FATMAD aims at providing application level fault tolerance features and also focuses on a protocol developers' perspective.

1.3 Results

With the previous objectives, the current version of FATMAD has been built on top of the Jade agent platform version 3.0b and it currently supports the rollback-recovery fault tolerance strategy.

As an extension of Jade, FATMAD incorporates the following features into Jade:

1) A set of fault tolerance service components (FATMAD runtime):

FATMAD integrates a set of components providing general services that are needed in a variety of checkpoint and recovery schemes, ranging from coordinated checkpoint/recovery, message induced checkpoint/recovery, to log-based recovery. These components can be customized and launched on Jade to form a *FATMAD runtime* environment that is needed by fault tolerant applications embedded with relevant protocols.

2) A set of built-in rollback-recovery protocols (selectable FT schemes)

Through these component services, FATMAD provides a set of complete fault tolerance protocols that can be applied to agent applications.

FATMAD provides a generic programming interface that is extended (by protocol developers) in each specific fault tolerance protocol. Application developers can easily incorporate the selected protocol into their application via this extended API.

3) Protocol skeleton and protocol development API

FATMAD can be extended and enriched with variant protocols. FATMAD provides a generic *protocol skeleton* and a set of APIs that can help protocol developers to implement various rollback-recovery protocols with significantly reduced effort.

4) Deployment tool

Deploying fault tolerance agents requires some parameters be specified at runtime. FATMAD provides a deployment tool that can be used to specify these parameters when loading fault tolerant agents to a Jade platform.

1.4 Organization of the thesis

The thesis is organized as follows:

In chapter 2, we introduce some background knowledge related to FATMAD. In chapter 3, we analyze framework requirements from a user perspective and introduce the framework design approach. In chapter 4, 5, and 6, we introduce the framework model, architecture, and detailed design. In chapter 7, 8, we introduce how FATMAD can be used by framework users: protocol designers and application designers. In chapter 9, we introduce some evaluation. In chapter 10 conclude the thesis.

Chapter 2 BACKGROUND

As we already mentioned, FATMAD framework has been built on top of Jade agent platform and it mainly provides support for fault tolerance techniques based on checkpoint and rollback-recovery strategy. In this chapter, we introduce some related background materials that are involved in this project. We give a brief overview on checkpoint and rollback-recovery techniques, agent based paradigm, and Jade agent platform.

2.1 Introduction to agent-based system technology

2.1.1 Agent

Originating from artificial intelligence, the discipline of agent-oriented software engineering (AOSE) has emerged for more than ten years, and gained a lot of interests in computer science community. As a new paradigm for conceptualizing, designing, and implementing software systems, agent-based technology is intended to emulate or simulate the way human act in their environment, interact with one another, and cooperatively solve problems [Adina98].

An agent-based system is composed of active software entities, i.e. agents, which carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires [a definition of IBM]. Software agents are usually expected to encapsulate some of the following generic characteristics [KSTV+01]:

- Autonomy:

Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

- Social ability:

Agents interact with other agents via some kind of agent-communication language.

- Reactivity:

Agents perceive their environment and respond in a timely fashion to changes in it.

- Pro-activeness:

Agents do not simply act in response to their environment. They are able to exhibit goal-directed behaviour by taking the initiative.

- Learning:

Adaptive changes its behaviour based on its previous experience.

- Mobility:

Agents are able to transport itself from one machine to another.

2.1.2 AOP

Developing software system with such characteristics is quite different from conventional development paradigm, such as object-oriented design (OOD), object oriented programming (OOP). However, this new paradigm, agent-oriented programming (AOP), can be regarded as an extension of object-oriented programming (OOP), with the following key additional features [Tveit01]:

- Agents are active objects with independent threads of execution;
- Agents have autonomous behaviour that can't be directly controlled;
- Agents usually support structures for representing mental components, i.e. beliefs and commitments;
- Agents support high-level interactions (using agent-communication languages) between agents based on the "speech act" theory as opposed to ad-hoc messages frequently used between objects. Examples of such languages are FIPA ACL and KQML.

2.1.3 Multi-agent system

Quite large amount of applications require multiple agents to accomplish some complex tasks. A multi-agent system (MAS) can be regarded as a set of loosely coupled software agents scattering over networks and interacting to solve problems that are beyond the individual capacities or knowledge.

As stated in [CM-MAS], the MAS approach has the following advantages over a single agent or a centralized approach:

1) MAS distributes computational resources and functionalities across a network of interconnected agents. Whereas a centralized system may be plagued by resource limitations, performance bottlenecks, or critical failures, an MAS is decentralized and thus does not suffer from the "single point of failure" problem associated with centralized systems.

2) MAS allows for the interconnection and interoperation of multiple existing legacy systems. By building an agent wrapper around such systems, they can be incorporated into an agent society.

3) MAS models problems in terms of autonomous interacting agents, which is proving to be a more natural way of representing task allocation, team planning, user preferences, open environments, and so on.

4) MAS efficiently retrieves, filters, and globally coordinates information from sources that are spatially distributed.

5) MAS provides solutions in situations where expertise is spatially and temporally distributed.

6) MAS enhances overall system performance, specifically along the dimensions of computational efficiency, reliability, extensibility, robustness, maintainability, responsiveness, flexibility, and reuse.

2.1.4 Available techniques and tool support for MAS

MAS is increasingly getting a lot of attention and the MAS community has done quite a lot of work to promote multi-agent system development. These works mainly involve the following aspects:

(1) Application domain

MAS has been explored by both scientific and industrial communities. Quite a lot of experimental projects have been conducted. [AgentLink] lists more than ninety projects exploring various agent-based applications in different domains.

(2) Development methodology

As a new paradigm, the growth of MAS requires mature methodologies. Many diverse Agent Oriented Software Engineering (AOSE) approaches and methodologies have been proposed, including Gaia [WoJK00], MESSAGE [MaDN02], MaSE [DeLoach99], Prometheus [PaWi02] and Tropos [GiMP02], AAII methodology [KiGR96], AUML [OdPB00], and ROADMAP [JuPS02]. Each of the methodologies has different strengths and weaknesses, and different specialized features to support different aspects of their intended application domains.

(3) Agent standard

To promote collaboration and interaction between agents and interoperation among different agents systems, there are some standards and specifications that have been proposed and can be applied to agent-based systems.

- **KQML**

One of them is the Knowledge Query and Manipulation Language, also known as KQML [FiLa97], which is part of the ARPA sponsored project: Knowledge Sharing Effort. KQML is both a message format and a message-handling protocol to support run-time knowledge exchange among agents. It focuses on an extensible set of performatives, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores. The performatives comprise a substrate on which to develop higher-level models of inter-agent interaction such as contract nets and negotiation. In addition, KQML provides a basic architecture for knowledge sharing through a special class of agent called communication facilitators which coordinate the interactions of other agents.

- **MASIF**

Another standard is MASIF [OMG-MASIF] [MBBC+98]. In 1995 the OMG started working on a standard, called Mobile Agent Facility (MAF), in order to promote interoperability among agent platforms. In 1997, a joint submission by IBM, General Magic, The Open Group, GMD FOKUS, etc., was presented to the OMG. And the standard's name was changed from MAF to Mobile Agent System Interoperability Facility (MASIF). In 1998 this specification was accepted as an OMG standard. The current edition was issued in 2000.

MASIF was developed in order to achieve a certain degree of interoperability between mobile agent platforms of different manufacturers without enforcing radical platform modifications. MASIF is not intended to build the basis for any new agent platform. Instead, the provided specifications shall be used as an "add-on" to already existing systems.

- **FIPA**

Another important standard that is gaining increasing popularity is FIPA specification[FIPA]. The Foundation for Intelligent Physical Agents (FIPA) was formed in 1996 to produce software standards for heterogenous and interacting agents and agent-based systems. It is a non-profit association formed under Swiss law. Its members include companies and universities. FIPA identified a list of agent technologies deemed to be specifiable in 1997 and standardization work started. There is a set of spec called FIPA 97 and another called FIPA 98, both are now on Obsolete Status. The current spec is FIPA 2000, half of which is in the Preliminary Staus, another half on Experimental Status.

FIPA's main effort is towards the production of internationally agreed-upon specifications that provide a standard for the development of agent-based applications, services and equipment. FIPA's vision of the future landscape in agent technology depicts large agent societies, in which agents can co-operate.

At this time, the FIPA's specifications are grouped into 5 categories:

1. Applications
2. Abstract Architecture
3. Agent Communication

4. Agent Management

5. Agent Message Transport

In general, these specifications provide:

- A commonly agreed means by which agents can communicate with each other so that they can exchange information, negotiate for services, or delegate tasks;
- Facilities whereby agents can locate each other (i.e. directory facilities)
- An environment which is secure and trusted where agents can operate and exchange confidential messages
- A unique way of identifying other agents (i.e. global unique names)
- A means of accessing non-agent and legacy systems, if necessary
- A means of interacting with users
- A means of migrating agents between platforms
- etc.

Comparing with other specifications, we can see FIPA gradually integrates some features from other standards. For example, FIPA ACL derives from KQML, and FIPA 2000 specification deals with the mobility aspect of agents. It tries to integrate FIPA and MASIF.

- **Agentcities**

Agentcities [Agentcities] is an initiative that was first conceived in January 2000 to create a next generation Internet that is based upon a worldwide network of services that use the metaphor of a real or a virtual city to cluster services. These services, ranging from eCommerce to integrating business processes into a virtual organization, can be accessed across the Internet, and have an explicit representation of the capabilities that they offer. The ultimate aim is to enable the dynamic, intelligent and autonomous composition of services to achieve user and business goals, thereby creating compound services to address changing needs. Agentcities is based on FIPA.

Since its inception the testbed network has grown rapidly to support a wide range of prototype systems: from small test systems to large demonstrators involving over 100 deployed agent-based services.

(4) Agent platform and tools

There are quite a few agent platforms and frameworks that have been developed to support agent-based system development and deployment. Some mobility-oriented agent platforms include IBM's Aglets, General Magic's Odyssey, ObjectSpace's Voyager, IKV's Grasshopper, Mitsubishi's Concordia, James, AgentTcl, MOA, etc.

FIPA, as an important multi-agent system standard, has gained encouraging support from many publicly available agent platforms: Agent Development Kit, April Agent Platform, Comtec Agent Platform, FIPA-OS, Grasshopper, JACK Intelligent Agents, JADE, JAS (Java Agent Services API), LEAP, ZEUS.

MASIF standard doesn't have as many implementations as FIPA does. Some MASIF-compliant platforms are Grasshopper (by IKA), Open Mobile Agent (SOMA) System (by Universita' di Bologna in Italy).

2.2 Introduction to Jade

2.2.1 Why choose Jade?

In this project, we choose Jade [Jade] as agent platform, on which we implement the fault tolerant framework. Existing multi-agent platforms are not mature and stable enough to be commercialized, although there are a few commercial products announced. One reason to choose Jade is that it has gained increasing popularity and it currently has a relatively large user community and applications. Another reason is that Jade has relatively more complete features for multi-agent systems and it supports the FIPA standard.

2.2.2 What's in Jade?

Jade (Java Agent Development Framework) is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agent. It includes two main products: a FIPA-compliant agent platform and a package to develop Jade agents.

Jade agent platform provides a runtime environment for software agents to execute, to manage their execution, to access system resources, and to guarantee integrity and protection of agents and the platform itself. Jade also provides support for migration, naming, location and communication services.

Jade comes with an agent programming model to help developers implement agents that are supported by the Jade platform. Such programming model captures common requirements for developing a multi-agent system, such as concurrency, asynchronous communication, agent mobility, high-level interaction, etc. Jade development package is the framework that can be used to implement Jade supported agents to run on Jade platform.

In addition, Jade provides support for agent communication at different levels of abstraction specified by FIPA, such as agent communication language (ACL) support, ontology support, content language support and so on.

In general, Jade's implementation complies to FIPA2000 specification, which guarantees that agents running on Jade platform are able to communicate with agents running on other FIPA-compliant platforms.

2.2.3 Jade agent platform

To develop a framework on Jade, we need first to understand the Jade platform. Jade conforms to FIPA standards, which has relevant platform architecture specification and agent management specification.

Figure 2-1 FIPA reference agent platform architecture illustrates the agent management reference model for an agent platform, which is specified in FIPA.

The reference model consists of the following logical components [FIPA], each representing a capability set:

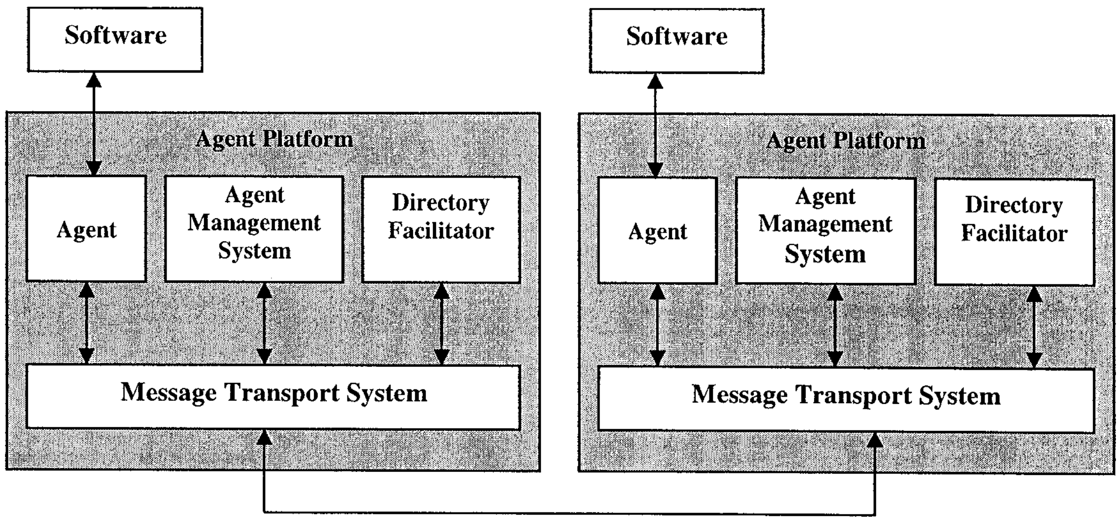


Figure 2-1 FIPA reference agent platform architecture

1) Agent Platform

An Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents.

Figure 2-2 illustrates the Jade platform topology. A Jade agent platform can be split on several hosts. The part of platform on each host is one Java application, and hence one Java virtual machine. Each JVM is basically an agent container that provides a complete run time environment for agent executions and allows several agents to concurrently execute on the same host.

The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

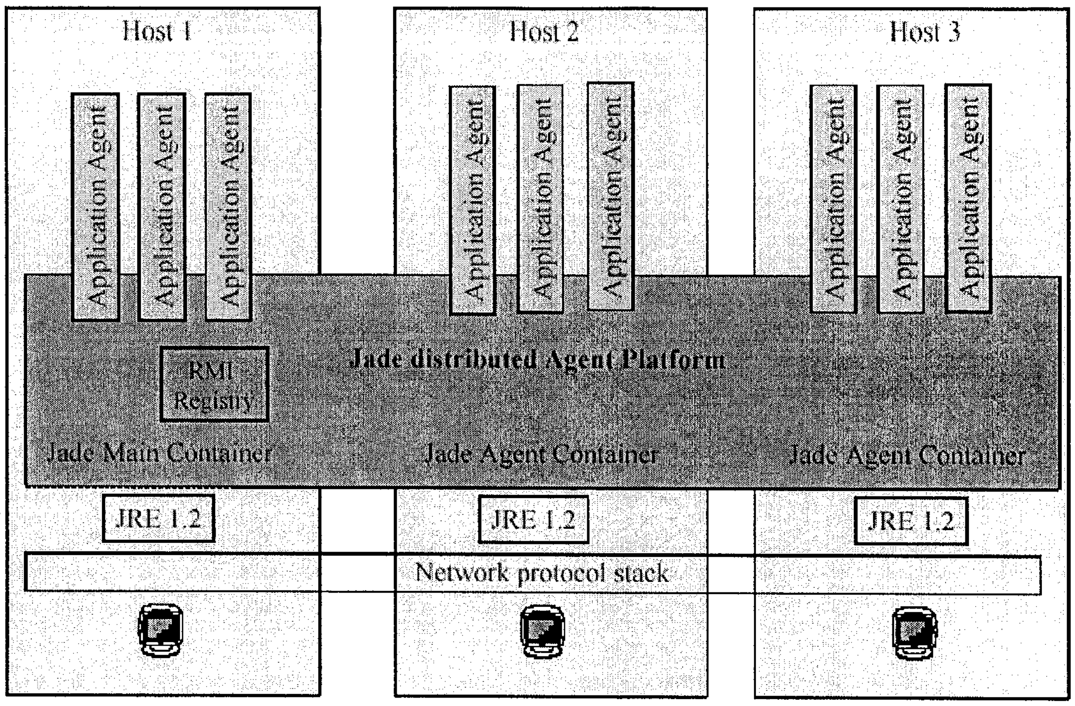


Figure 2-2 JADE Agent Platform distributed over several containers

2) Agent Management System

The Agent Management System (AMS) in Jade is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

3) Agent

An agent is a computational process that implements the autonomous, communicating functionalities of an application. Agents communicate using an Agent Communication Language(ACL). An Agent is the fundamental actor on an agent platform that combines one or more service capabilities into a unified and integrated execution model. Each agent has a unique Agent Identifier (AID) labelling it so that it may be distinguished unambiguously within the Agent Universe.

4) Directory Facilitator

The Directory Facilitator (DF) in Jade is the agent that provides yellow pages services to other agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents.

5) Message Transport Service

The Message Transport Service (MTS) in Jade, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

6) Software

- Software describes all non-agent, executable collections of instructions accessible through an agent. Agents may access software, for example, to add new services, acquire new communications protocols, acquire new security protocols/algorithms or new negotiation protocols, etc.

2.2.4 Jade agent programming model

To design application level fault tolerance features, we need to clearly understand the model for programming Jade agents. This involves Agent class, computational model, agent life cycle control, agent communication, and agent migration.

1) Agent class

The *Agent* class represents a common base class for user defined agents. Therefore, from the programmer's point of view, a JADE agent is simply an instance of a user defined Java class that extends the base *Agent* class. This implies the inheritance of features to accomplish basic interactions with the agent platform, e.g., registration, configuration, remote management, etc., and a basic set of methods that can be called to implement the custom behaviour of the agent e.g., send/receive messages, use standard interaction protocols, register with several domains, etc.

2) Computational model

The computational model of an agent is multitasking, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours, represented by *Behaviour* classes. A scheduler, internal to the base Agent class and hidden to the programmer, automatically manages the scheduling of behaviours. However, Jade agent scheduler doesn't support this concurrency transparently. The developer is required to design the semantics of concurrency among behaviours for an agent.

3) Behaviour class

The Behaviour class is a common base class that can be extended by users to program user defined agent behaviours.

The detailed behaviour action is programmed into the *action()* method, that is invoked each time this behaviour is scheduled to execute by the agent internal scheduler, until this behaviour is removed from the agent internal scheduler.

As soon as the *action()* method returns, the scheduler will invoke another method: *done(),* which returns a Boolean value, to decide whether to remove this behaviour from the internal scheduler.

Therefore, to extend a *Behaviour* class, the developer is required to implement/override the two methods: *action()* and *done().*

The *action()* method is in fact the atomic action block for the agent scheduler. In other words, once a behaviour is schedule to run, its *action()* method will be executed completely. There is no interleaving among different behaviours within any *action()* method. Therefore, we can name one run of *action()* method as ***atomic behaviour action*** (ABA). Figure 2-3 An agent runs with concurrent behaviours shows an example of an agent with three behaviours: b1, b2, and b3, and illustrates how the agent scheduler works.

From this programming model we can see that the developer role is in deciding when to terminate an *atomic behaviour action* at design time. The developer should also carefully design some state variables to remember and control the progress of each behaviour so that the *atomic behaviour action* does not always repeat the same action if it is not designed to do so. The length of an *atomic behaviour action* should be carefully designed to avoid starving some agent behaviours.

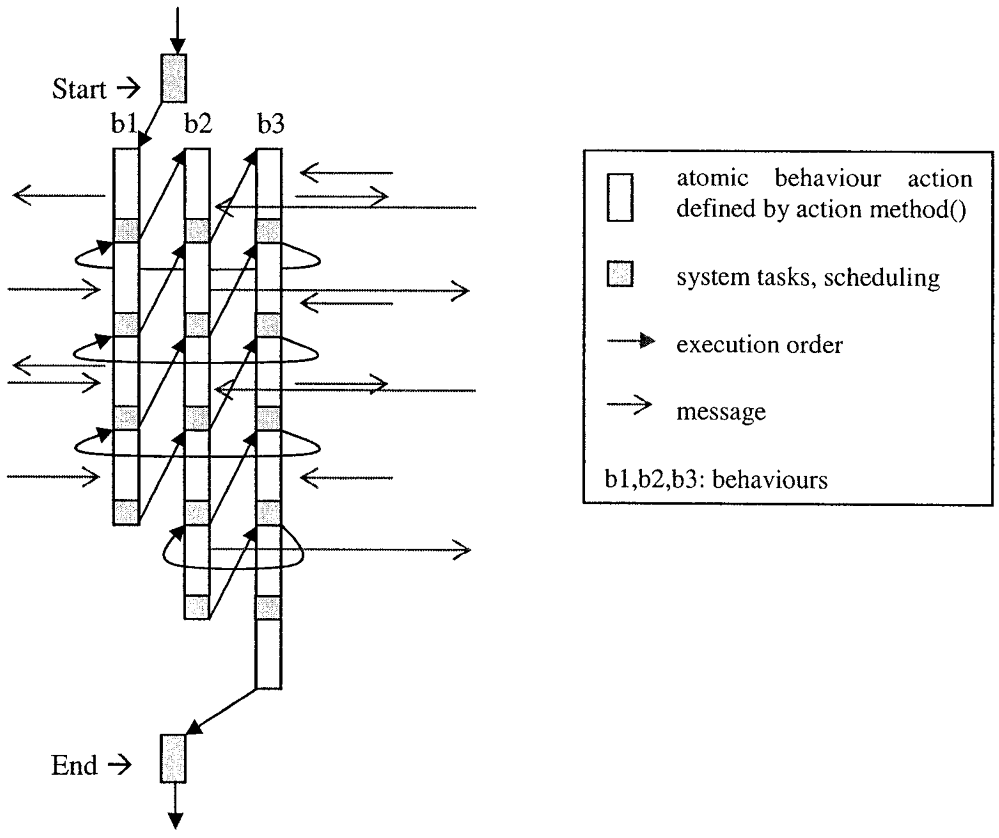


Figure 2-3 An agent runs with concurrent behaviours

4) Inter-agent communication.

Jade agents communicate with other agents through the Jade Agent Communication Channels. The Agent class provides a set of methods for inter-agent communication. According to the FIPA specification, agents communicate via asynchronous message passing, where objects of the *ACLMessage* class are the exchanged payloads. The *Agent.send()* method allows to send an *ACLMessage.* The value of the *receiver* slot holds the list of the receiving agent IDs. The method call is completely transparent to where the agent resides, i.e. be it local or remote, it is the platform that takes care of selecting the most appropriate address and transport mechanism.

5) Accessing the private queue of messages

The platform puts all the messages received by an agent into the agent's private queue. Several access modes have been implemented in order to get messages from this private queue.

The message queue can be accessed in a blocking (using *blockingReceive()* method) or non-blocking way (using *receive()* method). Both methods can be augmented with a pattern-matching capability where a parameter is passed that describes the pattern of the requested *ACLMessage*.

6) Mobility

Jade platform implements weak migration to support agent mobility.

2.3 Rollback-recovery fault tolerance techniques

Fault tolerance is an important strategy to build dependable systems. In this section, we introduce some basic fault tolerance concepts and techniques that are related to our work.

2.3.1 Faults, failures, and fault tolerance techniques

Fault-tolerance is the property of a computer system to continue operation at an acceptable quality, despite the unexpected occurrence of hardware or software failures. As illustrated in [Tanenbaum02] faults are generally classified as transient, intermittent, or permanent. Transient faults are non-repeatable faults. Intermittent faults periodically and unexpectedly. Permanent faults are the faults that continue exist until they are fixed. A program bug in an agent is a typical permanent fault.

A fault may lead to a failure. Failures are various, such as crash failure, response failure, arbitrary failure. The failures we are dealing with in this work are fail-stop failures caused by transient faults. For example, an agent may crash or a agent node (container) may crash.

The key technique for fault tolerance is to use redundancy. As illustrated in [Tanenbaum02] there are three types of redundancy:

- **Information redundancy:**

With information redundancy, extra information is used so that the garbled information can be recovered when failures present. A Hamming code applied in data transmission is a typical example of this type of redundancy.

- **Time redundancy:**

With time redundancy, an action is repeated when a failure occurs. Typical examples are transactions, in which aborted transaction can be redone with no harm.

Rollback-recovery via checkpoints and event log is an example that use both information redundancy and time redundancy.

- **Physical redundancy:**

With physical redundancy, extra processes or equipments are used so that when a failure partially affects system's functionality, the remaining system can continue to function. Replicating software processes is a typical use of this type of redundancy and is widely used.

2.3.2 Rollback-recovery

There are different techniques to handle faults, such as masking, recovery, self-stabilization. The current version of our framework provides fault tolerance support that is based on rollback-recovery technique.

Rollback-recovery is a backward error recovery technique. The main idea of this technique is to bring a system from a failure to a previous correct state when the failure presents. To recover a failed system, some recovery related information such as checkpoints, i.e. program states, is required to reconstruct the system state. Recovery related data are recorded during failure free execution and they should be saved in a stable storage that can survive the failures to be tolerated.

Figure 2-4 illustrates a checkpoint and failure recovery scenario during an application's lifeline. At runtime, the application takes three checkpoints. When a failure occurs, the application is recovered from the most recent checkpoint state (C3).

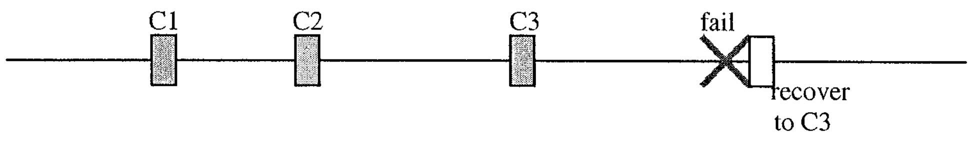


Figure 2-4 Checkpoint and rollback-recovery of an application

A successful recovery depends on whether the necessary information is available to reconstruct the program. The application characteristics determine what information is necessary and sufficient to be saved in order to support a failure recovery.

2.3.3 Rollback-recovery for distributed message-passing systems

2.3.3.1 Issues in distributed systems

A multi-agent system is a distributed system involving many processes (agents) running at different locations and communicating across the network. When dealing with distributed applications, rollback-recovery techniques become relatively complicated and should be carefully applied.

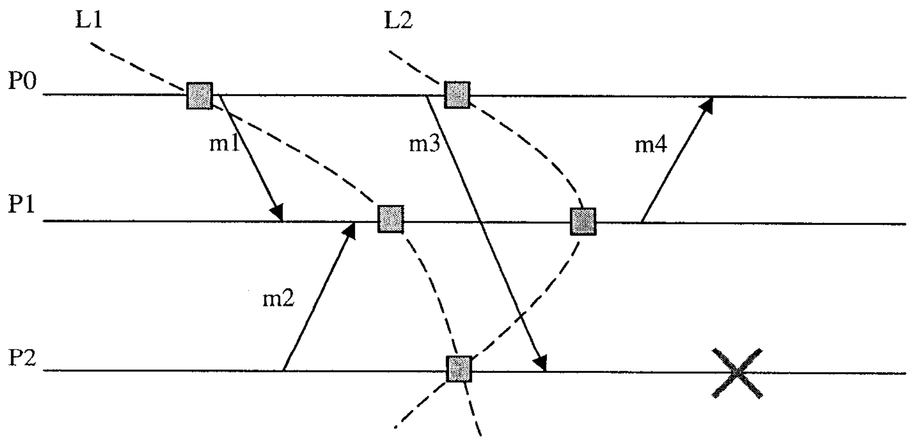


Figure 2-5 Checkpoint and rollback-recovery on distributed application

Figure 2-5 illustrates a checkpoint and recovery scenario of a distributed application. There are three processes in the application and they communicate through message passing. At run time, they take checkpoints according to certain schemes. When a failure occurs, the recovery action may have different choices based on the available checkpoints. In this example, L1 and L2 are two possible global states, each of which is constructed by a set of local states composed of checkpoints at all processes. L1 is an inconsistent global state, in which process P0 has not sent message m1 but process P1 has already received it. L2 is a consistent global state. However, there is a message (m3) *in-transit* in L2, in which P0 has sent a message of m3 but p2 has not received it yet. This leads to missing of m3 at the recovery if it is not saved since P0 will not send it again.

Inconsistency and message in-transit are two main issues that need to be considered in the design of a message passing based distributed checkpoint and recovery algorithm.

2.3.3.2 Protocols

There are quite a few distributed checkpoint-recovery protocols (algorithm). In general, as illustrated in [EAWJ02], they can be classified into two categories: checkpoint-based protocols and log-based protocols.

1. Checkpoint based rollback-recovery

Checkpoint-based rollback-recovery relies only on checkpoints to achieve fault tolerance. Upon a failure, checkpoint-based rollback-recovery restores the system state to the most recent consistent set of checkpoints, which form a recovery line [Randell75].

As illustrated in [EAWJ02], checkpoint-based rollback-recovery techniques can be classified into three categories: uncoordinated checkpointing, coordinated checkpointing, and communication induced checkpointing.

1) Uncoordinated checkpointing

Uncoordinated checkpointing, also known as independent checkpointing, allows each process to record its local state from time to time in an uncoordinated fashion. This technique has the advantage that each process has maximum autonomy to take checkpoints. However, this approach may lead to possible domino effects, in which large amount of checkpoints are useless so that processes have to be rolled back to the beginning of the computation. Those useless checkpoints incur overheads and cannot contribute to the recovery. Since a recovery line is not known at runtime, each process is required to maintain multiple checkpoints that consume large storage space.

In this technique, if a failure occurs the recovery process has to collect all checkpoint dependency information, which is recorded in each checkpoint, in order to determine a consistent recovery line for recovery. Construction of recovery line is usually complicated.

Some techniques in constructing recovery lines for uncoordinated checkpointing are provided in [EAWJ02].

2) Coordinated checkpointing

In coordinated checkpointing, all processes are synchronized to take checkpoints in order to form a consistent global state during failure free executions. When a failure occurs, each process is rolled back to its most recent checkpoint.

The main advantage of coordinated checkpointing is that the saved checkpoints are automatically consistent. It makes the recovery process easier than uncoordinated checkpointing and is not susceptible to the domino effect. However, coordinated checkpoint may incur overhead caused by checkpoint coordination.

There are generally two ways of checkpoint coordination:

- Blocking checkpoint coordination

This technique use block communications for checkpoint coordination. [TamSeq84] is a typical example that applies a two phase commit protocol when taking checkpoints.

- Non-blocking checkpoint coordination

This technique tries to avoid large coordination overhead by applying non-block communication when taking checkpoints.

Typical examples of this type are described in [ChanLamp85], [LaiYang87], [ElnZwa92], [Silva97], etc.

Some protocols such as [CriJah91] [TongKT92] apply synchronized clock mechanism as an assistant to achieve checkpoint coordination with reduced overhead.

3) Communication induced checkpointing

Communication induced checkpointing, also known as quasi-synchronous checkponiting [ManSing99], tries to avoid the domino effect without requiring all checkpoints to be coordinated. In this technique, processes take two kinds of checkpoints:

- Basic checkpoints: They are taken independently.
- Forced checkpoints: They are message induced.

Forced checkpoints are taken to prevent the creation of useless checkpoints, which will never be part of a consistent global state. Communication induced protocols do not exchange any special coordination messages to determine when forced checkpoints should be taken. Instead, they piggyback protocol specific information on each application message. Then the receiver uses this information to decide if it should take a forced checkpoint.

[ManSing99] and [EZWJ02] systematically present details of this technique.

2. Log-based rollback-recovery

Checkpointing is an expensive operation that involves blocking the process's execution, serializing the state of the process and its data, and saving them into stable storage. Log based rollback-recovery is the technique that tries to reduce checkpoint overhead.

Log-based rollback-recovery relies on the assumption based on the *piecewise deterministic* model (PWD) [StroYam85], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's determinant.

During failure-free execution, each process logs the determinants of all the non-deterministic events on to stable storage. It also takes checkpoints to reduce the rollback distance during recovery. After a failure is detected, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding nondeterministic events precisely as if they occurred during the pre-failure execution.

In message-passing systems, message events are regarded as main non-deterministic events in log-based rollback-recovery. Message logging is the key technique for log-based rollback-recovery protocols. Messages can be logged at sender side or receiver side, which leads to the difference during recovery. Messages can be logged synchronously or asynchronously, which leads to the difference on system performance. Detailed techniques for log-based rollback-recovery can be found in [EZWJ02].

Chapter 3 Overview of the framework

This chapter gives an overview of FATMAD, including generic framework requirements, user perspectives, and design strategy.

3.1 Objectives of FATMAD

3.1.1 Failure model

The eventual objective of FATMAD is to handle application level failures of an agent-based system. There are different types of agent failures such as crash failure, omission failure, Byzantine failure, etc [Tanenbaum02]. The current version of FATMAD is aimed at handling fail-stop type agent crash failure.

As illustrated in Figure 3-1, an agent system can be divided into several layers. An agent crash may be caused by faults occurring at any layer. For example, some timing error occurred at a lower layer may cause an agent crash. An agent container crash will lead to a crash on all agents in that container.

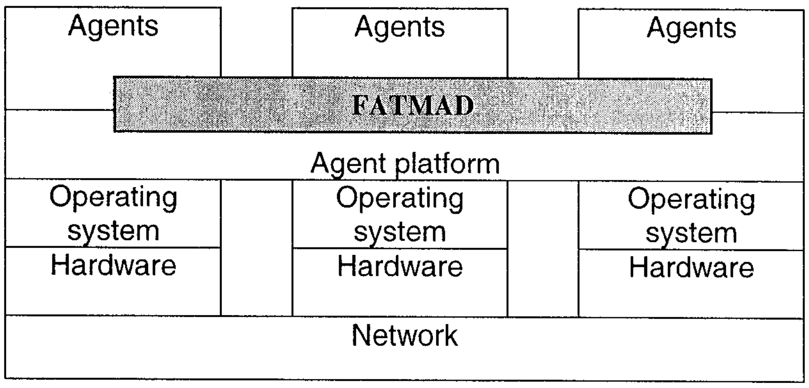


Figure 3-1 A layered view of an agent system.

We position FATMAD as an add-on layer that exists between agent layer and agent container layer. In general, FATMAD should be aimed at handling agent crash caused by faults occurring at any layer under the following assumptions:

1) The required network services can survive;
2) The agent platform can survive;

3) The FATMAD can survive.

3.1.2 Framework objectives

As we have already introduced in chapter 1, FATMAD is a framework that is aimed at building fault tolerant multi-agent applications, based on rollback-recovery fault tolerance techniques. Another goal of FATMAD is to serve as a test-bed to try out variant rollback-recovery techniques.

We chose Jade as our targeting agent platform. FATMAD is designed to be an extension or an add-on feature to Jade. It provides application level fault tolerance development support.

The main objectives of FATMAD are twofold:

1) FATMAD can be used to easily incorporate fault tolerant features into Jade agent applications.

2) FATMAD itself as a reusable and extensible system can be enriched by protocol developers so that they can try out and implement innovative fault tolerance protocols with minimum effort.

3.2 User requirements

Based on our objectives, FATMAD is targeted for two different groups of users: application developers and protocol developers. We briefly describe the perspectives of these two different user groups in the following:

3.2.1 Application developer

Application developers focus on agent application development. Fault tolerance for them is an add-on feature that can be incorporated into their applications. We don't assume that application developers have enough knowledge on detailed fault tolerance schemes. They should be relieved from the responsibility of implementing distributed fault tolerance schemes. What they ought to do is to select an available scheme (e.g., protocol in FATMAD) and use it correctly according to some relevant usage guides.

Accordingly, an application developer should not know about implementation details of FATMAD. When programming with FATMAD, they should only be concerned with how the applied fault tolerance features can be used and what is the expected behaviour and performance.

Hence, from an application developer's perspective, we can identify the requirements for FATMAD as follows.

- FATMAD is a middleware system that sits between Jade platform and Jade application. FATMAD provides a set of rollback-recovery fault tolerance schemes that allow application developers to select and apply to their applications.
- FATMAD should be as transparent as possible to application developers. In other words, coding with FATMAD should require minimal knowledge on the internal details of FATMAD.
- FATMAD APIs should be easy to use.

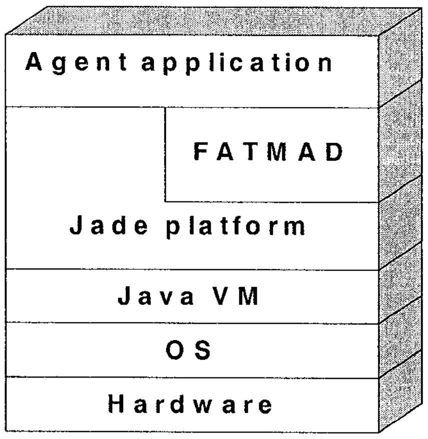


Figure 3-2 FATMAD as a middleware

In addition, FATMAD should provide the following flexibilities to the developer:

- An application developer should be able to define fault tolerance boundary so that only selected agents are fault tolerant.

- An application developer should be able to choose her own deployment plan so that she can address different failure assumptions or even optimize the performance.

The typical usage scenario of FATMAD for application developers should be the following:

1. Develop an agent application;
2. Analyze application characteristics and select a suitable protocol from FATMAD protocol library;
3. Refactor the agent application so as to incorporate the selected protocol into it;
4. Decide on proper deployment configuration and deploy the application.

3.2.2 Protocol developer

Unlike the application developers, protocol developers focus on designing fault tolerance schemes. As designers, they are assumed to have detailed knowledge of fault tolerance techniques. They bear the responsibility to guarantee that the protocol will behave correctly under their intended assumptions.

From the protocol developers' perspectives, the FATMAD framework is a set of reusable components that capture some common features of various rollback-recovery algorithms. Developing a specific protocol with FATMAD should not require implementing generic functions that also appear in other protocols. Instead, the development effort should focus on the protocol specific functionalities.

Hence, from a protocol developer's perspective, the requirements are as follows:

- FATMAD should provide functions that are common to different protocols and those functions should have an easy-to-use interface for protocol developers.
- FATMAD should provide effective support for implementing protocol specific functionalities.

When programming with FATMAD, the protocol designer must understand the framework structure and the framework model before implementing her own protocol.

Moreover, the protocol designer needs to clearly understand the semantics behind each interface she uses so that the invoked service behaves exactly as desired.

The typical usage of FATMAD for protocol designers in developing a specific protocol should be as follows:

1) Design the protocol;
2) Analyze the feasibility of implementing the protocol using FATMAD by looking at its design constraints;
3) Decompose the protocol into sub-protocols and identify those generic sub-protocols that are already available in FATMAD;
4) Implement the rest of the functions and integrate them with FATMAD;
5) Test the protocol;
6) Write the documents of the protocol including user's manual.

3.2.3 Relationships between the two user groups

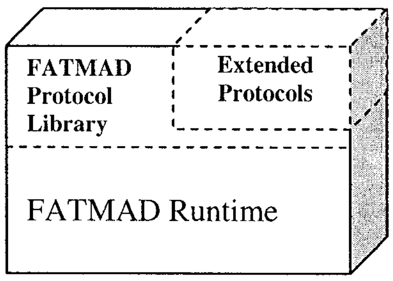


Figure 3-3 FATMAD framework

Apparently, the two user groups are related to each other since they all aim at making agents to be fault tolerant and they share responsibilities towards this common goal. One way of viewing the relationship between the two user groups is like "producer and consumer". The protocol designer delivers a usable protocol package and its relevant documents regulating how to use the package. The application developer can select that protocol and integrate it with her application. The application developer must conform to the usage constraints of the integrated protocol so that the hard coded fault tolerance mechanisms can function correctly.

3.3 framework design approach

The framework development, especially in this case, is a complex task since it involves many design variables. The following discussion gives a brief overview of the framework from the software engineering perspective.

3.3.1 What is a framework?

A framework is a generic application that allows a set of specific application variations to be extended. In general, a framework should contain the following:

1) An generic application model, usually an abstract algorithm, that makes framework generated applications functional;
2) Frozen spot functions that are immutable in different framework applications [Pree94];
3) Hot spot functions that vary in different framework applications. [Pree94]

By plugging in the hotspot functions, a concrete framework application can be generated.

3.3.2 Framework design path

Based on the above understanding of a framework, our framework design mainly follows the following design procedures:

1) Specify a generic application to establish the framework model:

By analyzing the requirements, we can specify the generic application that covers all the framework requirements in an abstract level. This generic application establishes the behavioural model of the framework. It tells how the framework application behaves in general. We apply a role model technique in modeling our requirement.

2) Design the system architecture:

Since the generic application illustrates the behavioural model of the framework, a set of framework components can be defined by decomposing the application functionality. In our case, these components are agents.

3) Design "FATMAD runtime":

From architecture design, we can identify those components and services that are immutable and are required for all protocol variations and for all applications. These components and services form a runtime environment that provides support for protocols and applications.

4) Build up a common protocol and application skeleton:

A framework, as a semi-complete application [JohFoo88], has hard-coded functionality (frozen spots) and some blank spaces to be filled in. Based on the runtime services, we can design the application functional structure to organize those frozen spot functions and hot spot functions. In our case, we designed a *protocol skeleton* and an application skeleton.

The *protocol skeleton* captures the common functions in different protocols. The application skeleton allows application developers to integrate their agent application with a FATMAD protocol. These issues are discussed in detail in the following chapters.

Chapter 4 Framework model and architecture

4.1 Generic application - framework model

Designing a framework is to design a generic application. Regardless of the variations in different framework-extended applications, the framework itself can be regarded as a semi-complete application at the abstract level. Any potential application should be able to be mapped into the framework. In our current version of FATMAD, the general application for FATMAD framework is a fault tolerant Jade agent application that implements a distributed rollback-recovery protocol. The generic distributed rollback-recovery protocol captures the framework functionality, which represents a large set of applications.

As we already discussed, a common Jade application is usually composed of a set of Jade agents, each of which is a single Java thread running in a Jade agent container. Jade agents are treated as distributed processes so that the distributed rollback-recovery protocol can be easily applied. At runtime, a concrete FATMAD supported rollback-recovery protocol serves as a middleware system providing fault tolerance services to a set of agents sitting on the Jade platform. The protocol functionality should be able to survive tolerated failures so that it can function whenever a tolerated failure occurs.

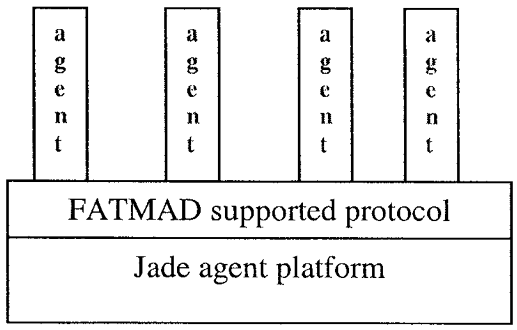


Figure 4-1 FATMAD supported fault tolerance

The generic rollback-recovery protocol is a protocol set that consists of three sub protocols: checkpoint/logging protocol, failure detection protocol, and recovery protocol.

A checkpoint/logging protocol is responsible for recording runtime information of a set of agents that need to be fault tolerant. The recorded information, which may be agent states, i.e. agent checkpoints, events, as well as other recovery related information, can be used by a recovery process to reconstruct agents' execution whenever a tolerated failure occurs and is detected. The recorded information should be saved in some stable storage that can survive tolerated failures.

A checkpoint/logging protocol runs when agents are in failure-free execution. It usually includes a checkpoint algorithm, which specifies when and how to take checkpoints of a set of agents at runtime, and an event logging algorithm, which specifies what runtime events such as ACL message events should be recorded.

A failure detection protocol is responsible for monitoring agents that need to be fault tolerant in order to detect tolerated failures and triggering recovery protocol if a failure is detected. Similar to checkpoint/logging protocol, failure detection protocol runs when agents are in failure free execution. Whenever a failure is detected, this protocol should stop its service until the failed agent is recovered. Apparently, the service component that carries failure detection task should be able to survive tolerated failures.

Recovery protocol is responsible for recovering failed agents and rolling back some related agents to assist the recovery. The recovery protocol is triggered by the failure detection protocol when a failure is detected. Then it utilizes the checkpoints, events, and other relevant information, which are produced by the checkpoint/logging protocol, to decide and roll back a set of related agents to a recovery line. When recovery process is done, the failure detection protocol and checkpoint/logging protocol can resume their execution.

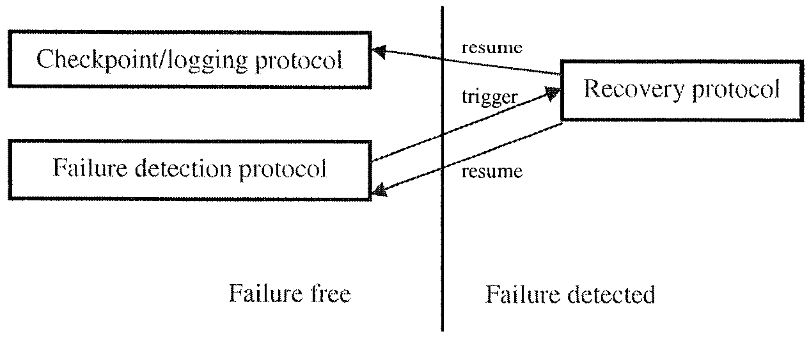


Figure 4-2 Rollback-recovery protocol set

These tree protocols must work together to make agents to be fault tolerant. The diagram in Figure 4-2 illustrates the relations among the three protocols. In general, the FATMAD framework is designed to build such a protocol set and that can be transformed into different variations.

4.2 Role model analysis

With the understanding of the model of the framework, we need to further analyze the functional requirement so that we are able to build a sound architecture. Role modeling provides a way of abstracting a design from the original problem. In our case, the framework application can be represented in a corresponding agent role model. We follow the role model proposed by E. A. Kendall [Kendall00] to illustrate our solution.

Roles represent certain functionality and interaction parties. Therefore, an application can be decomposed into a set of roles. First of all, we need to identify roles that are involved in the three-protocol set.

4.2.1 Checkpoint/logging protocol

In checkpoint/logging protocol, we identified four roles: *application role, checkpoint controller, message event logger*, and *storage manager*.

An *application role* is the application part in an agent that needs to be fault tolerant and therefore it has no fault tolerance functionality. An *application role* represents all application functionalities within one agent. It might communicate with other *application roles* in other agents.

The other three roles are all fault tolerance function roles. *Checkpoint controller* is responsible for taking checkpoints for an agent. *Message event logger* is responsible for logging agent messages. *Storage manager* is responsible for maintaining recovery related information such as checkpoints and message logs.

At runtime, the *checkpoint controller* needs to communicate with a *storage manager* to save checkpoints. It might need to talk to the *message event logger* to change logging policy. It might also need to talk to the *checkpoint controller* of other agents for checkpoint coordination.

For a specific fault tolerant agent, the *checkpoint controller* and the *application role* don't communicate with each other. However, the two roles are closely related since the *checkpoint controller* is responsible for taking checkpoints for the agent that includes its *application role* state. In addition, some constraint relation is required to be maintained between the two roles so that checkpointing actions can be performed correctly.

The *message event logger* also has a special relation with *application role.* It captures all necessary message events that are related to the *application role* and send logged information to a *storage manager.* It might also talk to the *checkpoint controller* of either the same agent or other agents and/or other agents' *message event logger* according to the specific protocol.

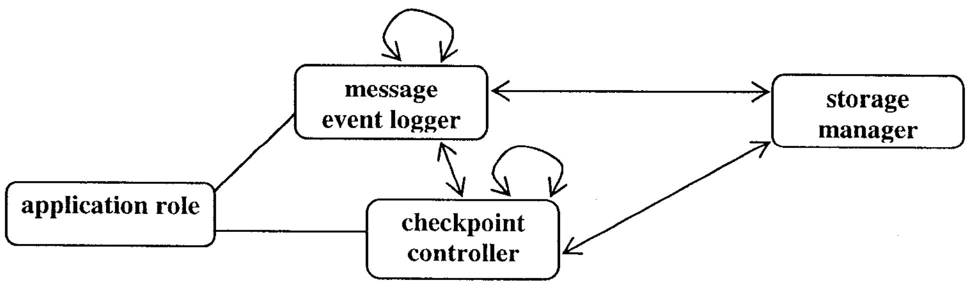


Figure 4-3 Role model for checkpoint/logging protocol

The role diagram in Figure 4-3 illustrates role relations in a checkpoint/logging protocol. We use rounded rectangles to represent roles, and lines to represent relations among roles. A directed line connecting two roles represents communication from one to the other. The communication may be either messaging or procedure call. An undirected line connecting two roles represents a non-communication relation between the two roles.

4.2.2 Failure detection protocol

In a failure detection protocol, we identified four roles: *application role, FD (failure detection) reporter, FD monitor, and failure reactor.* The role diagram in Figure 4-4 illustrates roles and their relations in a failure detection protocol.

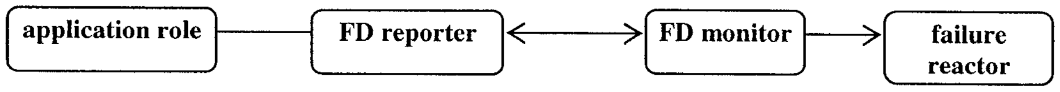


Figure 4-4 Role model for failure detection protocol

The *application role* is the same as the one in checkpoint/logging protocol. It is the part to be monitored. *FD reporter* is short for failure detection reporter. It is the part in an application agent that reports its relevant state to a *FD monitor.* The reporting information should reflect whether the *application role* has a failure occurring. Therefore, the *FD reporter* role and the *application role* are usually closely related.

FD monitor role is short for failure detection monitor. An *FD monitor* is responsible for collecting reporting messages from application agents, namely *FD reporter* roles in those agents to be monitored. In addition, by checking collected data, the *FD monitor* can identify those failed agents and inform the relevant *failure reactor* to react upon a failure.

Failure reactor role is responsible for triggering relevant reaction process whenever a failure is detected.

4.2.3 Recovery protocol

In failure detection protocol, we identified seven roles that are involved: *failure reactor*, *recovery manager*, *storage manager*, *recovery executor*, *post-recovery controller*, *execution controller*, and *application role*. The role diagram in Figure 4-5 illustrates these roles and their relations in a recovery protocol.

The *failure reactor* here is the same as in the failure detection protocol in which it triggers relevant failure reaction, e.g., failure recovery. The role of a *recovery manager* is to handle the following recovery procedure after being triggered by *failure reactor*. The *recovery manager* needs to talk to the *storage manager* to get recovery related information including checkpoints and message event logs in order to decide a recovery line. Once the recovery line is decided, the *recovery manager* needs to dispatch the recovery actions and some *recovery executor* roles carry out these actions. *Recovery executor* role is responsible for recovering a failed agent from a checkpoint or rolling back a running agent to a checkpoint. To rollback a running agent, the *recovery executor* needs to inform the *execution controller* role of an agent so that the agent's execution can be frozen and then wait for a *recovery executor* to roll it back.

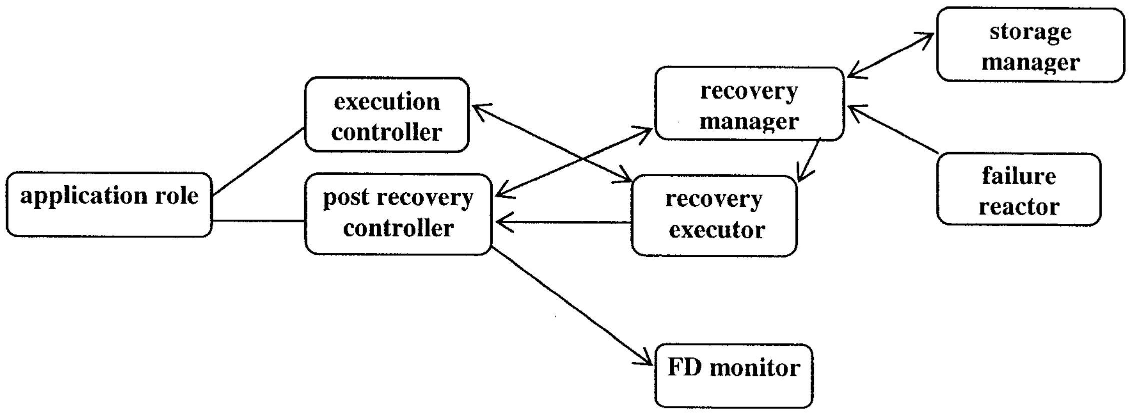


Figure 4-5 Role model for recovery protocol

When an agent is rolled back to a previous checkpoint, the *post recovery controller* role is going to control the subsequent agent execution and it also handles some remaining actions such as recovery coordination, channel flushing, message event handling, service resumption etc.

Both *execution controller* role and *post-recovery controller* role have non-communication relations with the *application role* so that they can control its execution.

4.3 Architecture

4.3.1 Design issues

The above analysis of the generic framework application discusses a role model for our fault tolerance framework. Based on that, we can design the system architecture by mapping these roles into different physical components.

Since the Jade platform provides a set of useful services, such as naming service and communication mechanism for agent development, our system design is agent based. In other words, all framework components in FATMAD are implemented as agents and the roles are appropriately mapped to these agents.

Mapping fault tolerance roles into agents requires us to consider the following issues:

1) Locality constraint:

Some roles have some locality constraints on the role assignment. In our case, the following roles have to coexist with the *application role* in the application agent.

- *Checkpoint controller*: A checkpointing action has to access the state of the application agent (where the *application role* resides) and during checkpointing the *application role* has to be frozen.
- *FD responder*: This role has to be mapped to the application agent in order to provide effective application agent state to the relevant *FD monitor* role.
- *Agent execution controller*: Similar to *checkpoint controller*, this role has to be allocated onto the application agent in order to control the agent execution.
- *Post recovery controller*: This role also needs to control application's execution and therefore, it has to be in the application agent.

2) Failure survivability:

Whenever an agent crashes, all functional roles within that agent can not survive the crash failure. To make an agent to be fault tolerant, some roles in the rollback-recovery protocol should be able to survive the tolerated failure. These roles include: *storage manager, FD monitor, failure reactor, recovery manager*, and *recovery executor.*

Usually these roles should be mapped to agents that are outside of the failure domain. For example, to tolerate a node failure, the agents encapsulating these fault tolerance roles should be deployed on different nodes.

3) Performance:

Fault tolerance feature usually generates a lot of overhead, such as consumption of resources including CPU cycles and memory, extra message communication. A good architectural design should minimize overhead and avoid performance bottleneck as much as possible.

4) Flexibility:

Different agent application may have different deployment requirements for the fault tolerance support. The architecture of the framework should provide some flexibility to allow developers to adapt FATMAD for their specific application needs.

4.3.2 Architecture design

Based on the role model of the framework and the above concerns, we design a framework architecture, in which five agent types are designed to realize the role model for rollback-recovery protocol. The diagram in Figure 4-6 illustrates the role mapping strategy in our architectural design. We briefly discuss these architectural agent components as follows:

1) *FT application agent*

This agent is designed to augment application agents with some additional functionalities so that they can become fault tolerant. In other words, application agents that need to be fault tolerant should extend this agent type. An agent of this type should encapsulate *application role*, *checkpoint controller* role, FD responder role, *execution controller* role, and *post-recovery controller* role.

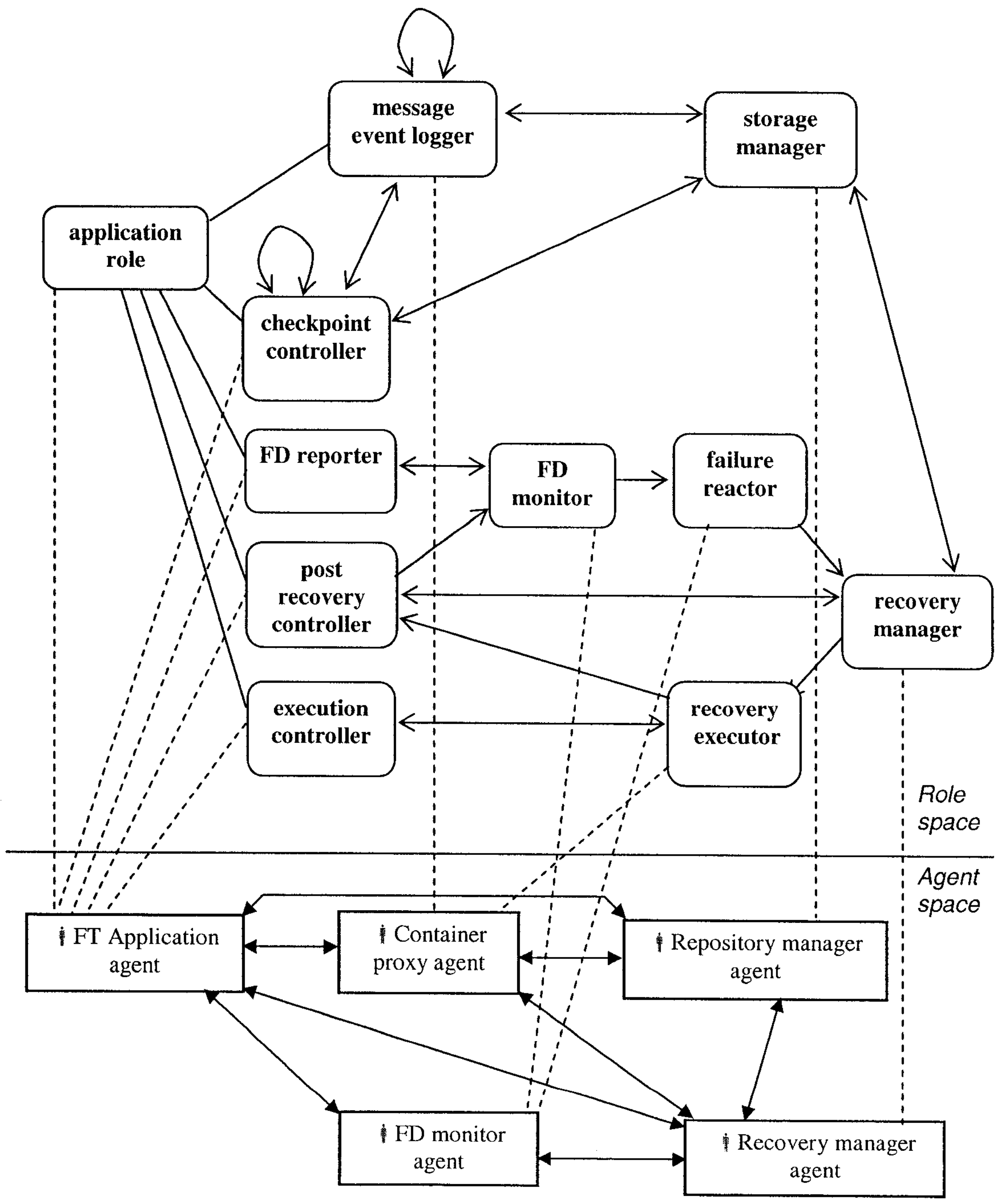


Figure 4-6 Role mapping strategy

2) Container proxy agent

This agent type, also named as *Container FT proxy agent*, is designed as a service agent to delegate some fault tolerance tasks at each node, i.e., Jade agent container. Each Jade agent container will have exactly one agent of this type.

At runtime, a *container proxy agent* serves as the *message logger* roles for all *FT application agents* in that container. It captures message events related to each *FT application agent* and sends necessary information to a *repository manager agent* according to predefined scheme. In addition, a *container proxy agent* also serves as a *recovery executor* role to in some recovery protocol to recover an agent to a checkpoint state.

3) Repository manager agent

This agent type is designed as a service agent to provide data service for *FT application agents*. It maintains recovery related information for a set of *FT application agents* and serves as *storage manager* role in the rollback-recovery protocol.

One agent of this type can serve many agents. Application developers can decide where to create the *repository manager agents* and which set of *FT application agents* can be served by each *repository manager agent*.

4) *FD monitor agent*

This agent type is designed as a service agent to play *FD monitor* roles in failure detection protocols for a set of *FT application agents*. One *FD monitor* agent can monitor many *FT application agents*. Application developers can decide where to create *FD monitor agents* and which set of *FT application agents* can be monitored by each *FD monitor agent*.

In addition, one *FD monitor agent* also plays the *failure reactor* role for each *FT application agent*. Once a failure is detected, the *FD monitor agent* will decide what reaction should be taken according to preconfigured protocol action scheme.

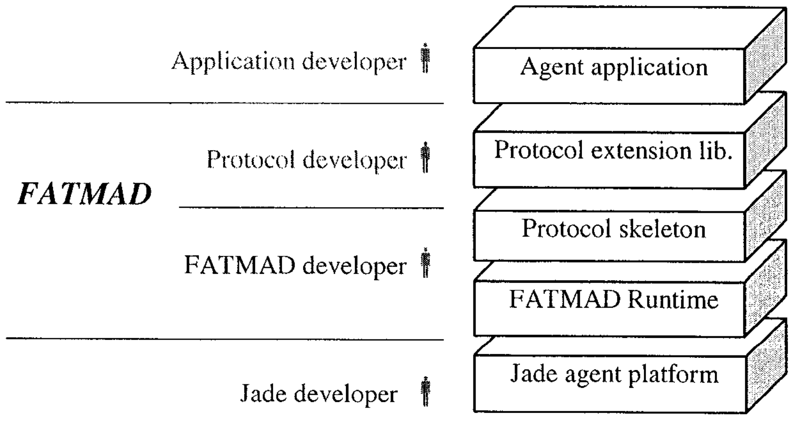
5) Recovery manager agent

This agent is designed to play the *recovery manager* role in the recovery protocol. A *recovery manager agent* should be dynamically created when a failure is detected, if such an action is configured in the relevant *FD monitor agent.* The *recovery manager agent* may access relevant *repository manager agents* to get recovery related information, compute a recovery line, dispatch recovery tasks for each agent involved, and synchronize post-recovery execution.

4.4 Framework layers

Up to now, we have introduced the generic framework application, the role model analysis, and the physical design of the framework application. However, a framework is not a specific application and it is designed to be extended. The development of a concrete framework application is a combinational effort from different groups working at different layers. Before we go into the design details of the framework, we first elaborate the multiple layers of our framework topology.

As we already discussed, the FATMAD framework serves two user groups: agent application developer and protocol developer. Figure 4-7 Layered view of FATMAD framework illustrates a layered view of FATMAD and the different



developer groups involved in each layer.

Figure 4-7 Layered view of FATMAD framework

As the diagram illustrates, FATMAD framework consists of three layers: *FATMAD runtime*, *protocol skeleton*, and *protocol extension library*.

1) FATMAD runtime

The ***FATMAD runtime*** is composed of a set of architectural agent components (that we discussed before) that form a runtime environment running on the Jade agent platform and providing fundamental services to the application agents that need to be fault tolerant.

In general, *FATMAD runtime* should provide some basic services that are required by fault tolerant application agents in different rollback-recovery protocols. These services include monitor service for failure detection, data services for storing and accessing recovery related information, message intercepting services for logging message events, and agent control services for recovering agents and rolling back agents on the Jade platform.

2) Protocol skeleton

The *FATMAD runtime* provides some basic services for rollback-recovery protocol. To make an agent application fault tolerant, we need to implement and deploy a detailed plan, e.g., a protocol, to serve the application. FATMAD intends to support different protocol variations. We extract common protocol features and design a *protocol skeleton* that serves as a template to help protocol designers develop various concrete protocols.

The *protocol skeleton* not only provides a protocol template for protocol developers, but also implements some common functionalities of the generic rollback-recovery protocol.

From protocol-development point of view, *FATMAD runtime* and the *protocol skeleton* together implement the frozen spot functionality of the framework, and the *protocol skeleton* also provides hotspot adaptation methods to allow protocol developers to plug in their protocol details.

3) Protocol extension library

This is the part that protocol developers should work on. To design a concrete protocol, a protocol developer needs to extend the *protocol skeleton* by filling its hotspots. This results in a ***protocol extension***, which is intended to be applied to an agent application that needs to be fault tolerant.

All workable *protocol extension*s form a ***protocol extension library*** that offers a set of choices to an application developer for developing fault tolerant agent applications on Jade.

We can summarize the relationships among different layers in FATMAD and an application as follows.

1) *Protocol skeleton* + *protocol extension* = protocol kernel;

2) Protocol kernel + agent application = fault-tolerant agent application;

3) A fault-tolerant agent application requires Jade and *FATMAD runtime* as runtime environment.

In the following chapters, we elaborate the detailed design of each layer.

Chapter 5 FATMAD runtime

A *FATMAD runtime* provides runtime environment for application agents that need to be fault tolerant. Whenever a tolerated failure occurs to an application agent, as long as its relevant *FATMAD runtime* service can survive the failure, the agent should be able to be recovered. Therefore, a *FATMAD runtime* should carry all necessary services dealing with tolerated failures.

Agents may apply different checkpointing/logging schemes which lead to differences on agent recovery. Therefore, the required fault tolerance services can be classified into two categories: protocol dependent services and protocol independent services. Protocol independent services are generic while protocol dependent services usually change with different protocols. In *FATMAD runtime*, we hardcode the functionality of protocol independent services and provide interfaces to allow protocol dependent services to be integrated dynamically. In this chapter, we introduce the components in *FATMAD runtime* as well as their services.

As already mentioned in last chapter, a *FATMAD runtime* consists of a set of agent components providing the following services:

- Monitor service

  This service plays the *FD monitor* role in a failure detection protocol.

- Data service

  This service plays the *storage manager* role in both a checkpoint/logging protocol and a recovery protocol.

- Message interception and logging service

  This service plays the *message event logger* role in checkpoint/logging protocol.

- Agent control service

  This service plays the *recovery executor* role in a recovery protocol.

From the role mapping strategy diagram in Figure 4-6, we can figure out that these services are provided by different agent types in our generic framework architecture.

A *FD monitor agent* can provide monitor service. A *repository manager agent* can provide repository service. A *container proxy agent* can provide message logging service and agent control service. Hence, we can see that a *FATMAD runtime* consists of three types of agents: *FD monitor agent*, *repository manager agent*, and *container proxy agent*.

5.1 ***FD monitor agent***

FD monitor agent is designed to play the *FD monitor* role, which is for detecting a failure, and the *failure reactor* role, which is for reacting to a detected failure, in any FATMAD supported rollback-recovery protocol. Hence, its duty includes two parts: failure detection and failure reaction.

1) Failure detection

In the current version of FATMAD, we assume that tolerated failures are of progress failure type. We implemented a heartbeat detection method to detect progress failures. With this method, an application agent on the platform advertise to a *FD monitor agent* that it is alive, every *prescribed interval of time* (by using timers). If the *heartbeat is missed*, the path, the agent or the node is declared as failed and a failure reaction is performed.

The implementation of this protocol involves two parts: *FD reporter* role that should reside in an *FT application agent*, and *FD monitor* role that should reside in *FD monitor agent*. An *FT application agent* advertises "alive" message to an *FD monitor agent* at a prescribed interval of time. The *FD monitor agent* receives "alive" messages and examines if the elapsed time since receiving last "alive" message has exceeded a timeout value so that a failure is going to be declared.

Before executing such a protocol, each *FT application agent* needs to know the following values:

- A heartbeat interval value,

- The Agent ID (AID) of the *FD monitor agent* that monitors it.

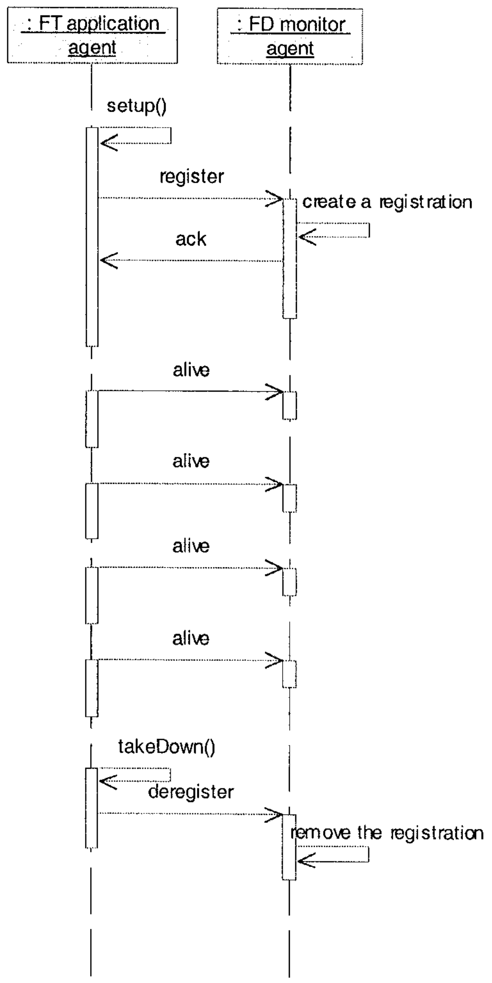


Figure 5-1 A failure detection protocol scenario in the absence of failures

And for each *FT application agent* to be monitored, an *FT monitor agent* needs to know the following:

- The agent ID (AID) of the *FT application agent*,
- A timeout value that determines a failure,
- An event reaction object that defines relevant failure reaction.

Except for the *event reaction object*, all these parameters should be specified when each *FT application agent* is deployed. We will explain the event reaction object later on.

Since an *FD monitor agent* serves a set of agents, each *FD monitor agent* maintains a registration table for agents that need to be monitored. Each registration in the table is associated with an agent that needs to be monitored and contains parameters that should be known by the *FD monitor agent* in relevant failure detection protocol.

The registration table in an *FD monitor agent* changes dynamically. At runtime, an *FT application agent* needs to register itself to an *FD monitor agent* by sending it a message containing necessary parameters before it can be monitored. The FT monitor agent then saves the registration to the table and starts to monitor the agent. When the *FT application agent* removes itself from the platform, it needs to inform the FT monitor agent to deregister it. Upon receiving a deregistration message, the *FD monitor agent* will remove the relevant item from the registration table and stop monitoring it. The sequence diagram in figure 5-1 illustrates a scenario of the failure detection protocol in the absence of failures.

2) Failure reaction

When an *FD monitor agent* detects a failure, it should trigger relevant reaction dealing with the failure. A recovery protocol as reaction process is usually initiated for it. However, detailed recovery reaction may vary in different protocols. This needs to be specified in a protocol design in upper layers.

In order to implement a generic failure reaction mechanism, we designed a Java interface *EventRecovery*, which defines a method: *void reactToEvent(...)*. To specify relevant failure reaction, a concrete class should be defined to implement *EventRecovery* interface by providing implementation of the *reactToEvent()* method. When an *FT application agent* registers to an *FD monitor agent*, an object of this class as a parameter will be passed to the *FD monitor agent* and saved during the registration. Whenever a failure is detected, the *FD monitor* will get the relevant reaction object from the registration table and invoke its *reactToEvent()* method that triggers the recovery process. The sequence diagram in figure 5-2 illustrates one scenario of failure detection in the presence of a failure.

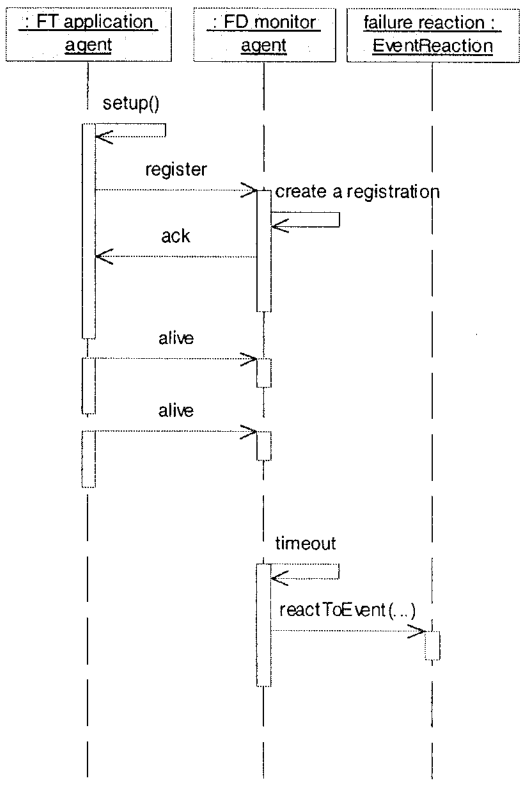


Figure 5-2 A failure detection protocol scenario in the presence of a failure

3) After failure recovery

When an agent is declared as failed by its *FD monitor agent*, its monitor service is then suspended. However, we still need this service to be resumed when the failed agent is recovered. The recovered agent should notify its relevant *FD monitor agent* to resume the monitor service. The *post-recovery controller* role is responsible for this action.

Except for the reaction process that may vary in different rollback-recovery protocols, the failure detection sub protocol remains the same in all protocol variations that deal with progress failure. Therefore, we hard code this service into the *FD monitor agent* and provide APIs to allow the service to be customized and altered at runtime.

5.2 Repository manager agent

Repository manager agent is designed to provide data services in variant rollback-recovery protocols. As its name suggests, it maintains a data repository to store agent recovery related information. Similar to *FD monitor agent*, a *repository manager agent* can serve a set of *FT application agent*s.

5.2.1 Repository structure

The data repository of a *repository manager agent* stores recovery related data including agent checkpoints, ACL message event log, etc, on its local hard disc. By applying *proxy* [GOF94] design pattern, we designed an ***AgentStorage*** class that provides a set of operations to query and manipulate repository data on hard disc. An *AgentStorage* object is in fact a *storage manager* and it is responsible for manipulating and querying repository data that are only related to one particular *FT application agent.*

In order to serve more agents, a *repository manager agent* maintains a *storage manager (AgentStorage* object reference*)* table in order to manage agent data respectively. In the table, each agent *storage manager* can be identified via its related agent ID (AID). Therefore, in order to perform a data operation, the reference to relevant *storage manager* must be retrieved from the *storage manager* table. The data operation can be done by invoking relevant methods of the retrieved *AgentStorage* object. These methods are provided in the *AgentStorage* class API.

5.2.2 Data services

Based on such a repository structure, a *repository manager agent* provides *FT application agent*s with the following services:

1) Accept registration and deregistration

Each *FT application agent* is required to register to the *repository manager agent* in order to be served. Upon receiving a registration message from an *FT* *application agent*, the *repository manager agent* records this registration information and creates a *storage manager* for this newly registered agent. Then it replies to the *FT application agent* via a message to inform that it is registered. After being registered, any data service request related to the registered agent can be served by the *repository manager agent.*

Whenever the *FT application agent* no longer needs this service, it should deregister the service by sending a deregistration message to the *repository manager agent.* Upon receiving such a message, the *repository manager agent* will remove the relevant *storage manager* as well as all data related to this deregistered agent from the repository and stop providing data services related to this agent.

2) ***Receive and save related information***

A *FT application agent* can send agent checkpoints to the repository manager it registered with. A *container proxy agent* can also send logged message event data as well as other recovery related data to repository mangers accordingly. Upon receiving a data message from an *FT application agent* or a *container proxy agent*, the *repository manager agent* will identify the ID (AID) of the agent that the message belongs to, get the agent's *storage manager* according to the agent ID, and save the data that the message carries to the repository via the *storage manager.*

3) ***Accept query and manipulation to the repository***

Data saved in the repository of a *repository manager agent* is mainly used for agent recovery. A *repository manager agent* allows *recovery manager agent*s to make queries to the repository so that they can retrieve checkpoints and message log to recovery failed agents.

In addition, a *repository manager agent* also allows *recovery manager agent*s or some predefined action to manipulate repository data, e.g., trimming off some useless data.

Data query and manipulation can be done directly by invoking the repository API, or indirectly by sending ACL messages, depending on the runtime locality condition.

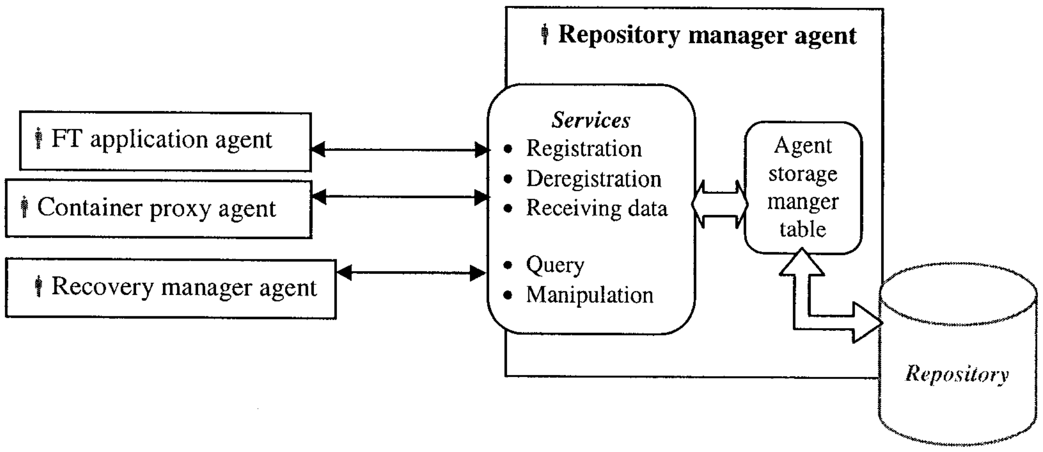


Figure 5-3 Repository manager

The diagram in Figure 5-3 illustrates the structure of a *repository manager agent* as well as its services.

5.3 Container proxy agent

*Container proxy agent*s are designed to be working closely with each Jade agent container and provide FATMAD system services that are related to Jade agent containers. Only one *container proxy agent* is deployed in one Jade agent container, providing services that are limited to be within its riding container. As its name suggests, each *container proxy agent* serves as a proxy to local *FT application agents* for message logging, and to *recovery manager agents* for agent recovery control.

5.3.1 Message logging service

A message logging service involves two types of actions:

1) Message event interception

This is to capture concerned message events for a specific *FT application agent.*

On Jade platform, a message transmission relies on platform system services and it includes three actions: send, deliver, and receive.

- Send

  The application behaviour object in the sender agent invokes send() method to send a message.

- Deliver

  The underlying system, namely the sender agent's riding container, takes the message and sends it to the container where the message receiver agent resides. The receiver container then delivers it to the receiver agent's message queue.

- Receive

  The application behaviour object in the receiver agent invokes receive() method and get the message from its message queue.

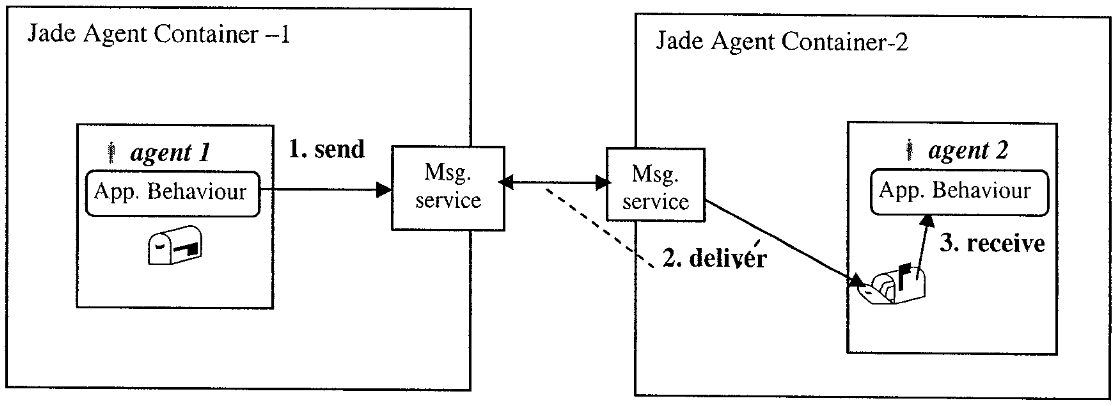


Figure 5-4 Message transmission mechanism

The diagram in Figure 5-4 Message transmission mechanismillustrates the message transmission path.

From the message transmission path, we can identify three important message event types for each agent:

- Message-sent event, means a message has been sent by an agent;
- Message-posted event, means a message has been delivered to an agent's message queue, but not received by the agent yet;

- Message-received event, means a message has been picked up from the message queue by the agent program.

In order to capture these types of events, we designed *container proxy agent*s to be working closely with each Jade agent container. By applying a notification mechanism, as soon as a message event occurs in an agent container, the *container proxy agent* in that container will be notified for further processing.

2) Message event logging.

When a message event is captured, the *container proxy agent* is required to take further action to deal with such an event according to certain logging policy. The message event information may be logged and sent to relevant *repository manager agent*, it may be discarded, or some other actions may be taken. This should be specified by each individual protocol and *FATMAD runtime* should not hard code it.

We designed a ***LoggingAction*** class that can be used to define logging policy as well as logging related functions. A protocol designer can customize it or extend it in each individual protocol. This class will be explained more in the next chapter, since it is part of *protocol skeleton.*

A *container proxy agent* can serve multiple agents in its riding container. Each *container proxy agent* maintains a registration table, in which each item is a *LoggingAction* object that encapsulates relevant logging function for a specific agent. Whenever the *container proxy agent* is notified of a message event, the *container proxy agent* will retrieve the relevant *LoggingAction* object from the table and invoke a relevant method of the object with the parameter of this message event data.

Similar to the *repository manager agent*, a *container proxy agent* requires *FT application agent*s in its riding agent container to register to it in order to be served and deregister the service when it is no longer needed. The registration/deregistration process is merged with the registration/registration process for the *repository manager agent.*

An *FT application agent* needs to register only to the local *container proxy agent* by passing a set of necessary parameters including a *LoggingAction* object and the agent ID (AID) of a *repository manager agent*. The *container proxy agent* will then save the *LoggingAction* object into its registration table and forward the registration message to the *repository manager agent*. When the agent deregister the service, the *container proxy agent* will remove its relevant item from the registration table and forward the deregistration message to the relevant *repository manager agent*.

5.3.2 Agent control service

A Jade agent can only be deployed on an agent container of a Jade platform. Jade containers as well as the platform manage and control the life cycle of each agent. In order to recover an agent from a checkpoint or roll back a running agent to a previous checkpoint, we have to embed some functions that directly control agents' execution at platform level.

Since Jade platform doesn't provide APIs to allow us to directly control agents at platform level, we modified Jade source code to insert some necessary functions that are required in all rollback-recovery protocols.

These functions include:

1) ***Isolate and kill failed agent***

   This is to kill an agent executing on the platform. The killing action includes two parts: remove the agent from the platform and trigger the failed agent inner mechanism so that it can terminate itself automatically.

   We also implemented a *reincarnation control* mechanism to isolate the failed agent from the outside world. We will explain reincarnation control in the next chapter.

2) ***Create an agent***

   This function is to create an agent on the agent platform by using a checkpoint so that the new agent can run starting from the checkpoint state.

This is a useful service when we intend to recover an agent provided the failed agent has been already eliminated from the platform.

3) ***Replace an agent***

If we intend to recover an agent on the same agent container, we can simply replace the old agent with a new one without notifying the Jade agent management service (AMS).

This includes three actions:

i) Replace the reference of the current agent object in the container with a new agent object reference;

ii) Recover the runtime configuration state of the new agent;

iii) Isolate and kill the old agent object.

In each agent container, these agent control services are accessible by the *container proxy agent*. Therefore, recovery actions in a recovery protocol can be done by sending a message to *container proxy agent*s to invoke these services.

5.4 Communication mechanism

In previous sections we have introduced three system agents in *FATMAD runtime*. At runtime, a rollback-recovery protocol involves *FATMAD runtime* agents, *FT application agent*s as well as other fault tolerance related agents such as *recovery manager agent*. The collaboration among these agents is crucial. The communication mechanism should be supportive of system collaboration.

5.4.1 Collaboration methods

In general, there are two ways of communication among these agents:

1) ***Method invocation*** **:**

This is an efficient and synchronous communication method. However, it is limited to the condition that communicating agent components are locally accessible from one to another. For example, if a *recovery manager agent* is in the same container where a *repository manager agent* resides, it can access the repository API directly to query recovery data. An *FT application agent* can directly register itself to the local *container proxy agent* by invoking its registration method.

2) ***Message passing*** **:**

As we already know, the Jade platform implements an asynchronous message passing mechanism, by which an agent can transmit FIPA compliant ACL messages. The FIPA ACL message format is implemented inside the *ACLMessage* class in Jade. All Jade agents can communicate via ACL messages.

5.4.2 FATMAD messaging mechanism

ACL as a high-level agent communication language is designed especially for agent communication. However, this message format is not very supportive in fault tolerance protocol development. We designed and developed a special messaging mechanism, which is implemented on top of an ACL message, to support distributed collaboration among different FATMAD components effectively. We applied a *visitor* [GOF94] design pattern in the design so that dynamic behaviours can be supported.

We define a *FATMAD system message* format, which can be applied to compose FATMAD system messages in order to communicate among different FATMAD agents. An FATMAD system message is wrapped by an *ACLMessage* object so that it can be transmitted as an ACL message by the Jade message transmission system.

FATMAD system message format is implemented as a Java interface *RRFTMessageObject*, which defines a method signature:

void processThisMessage(Agent receiver_agent, Behaviour receiver_behaviour)

This message type can be recognized by all FTMAD agent components, including *FATMAD runtime* agents, *FT application agent*, etc., and they are always ready to receive this FATMAD system message automatically. Whenever a FATMAD system message is received, the receiver component will unwrap the ACL message to get the FTMAD system object and invoke its *processThisMessage()* method so that the code inside this method can be executed. The runtime parameter of this method is the reference of receiver agent or receiver agent Behaviour object, using which this method can invoke the receiver agent's services.

In order to program with FATMAD system message, one can design a message class to implement *RRFTMessageObject* interface and *processThisMessage()*. Note that, in order to transmit such a message, one only needs to program the sender side and not the receiver side since it is received and processed automatically.

Through this mechanism, a system message object can be used to transmit not only data but also dynamic behaviour. This enables some distributed collaboration to be coded with ease.

The sequence diagram in Figure 5-5 FATMAD messaging mechanism illustrates a scenario where a system message is transmitted between two FATMAD agent components.

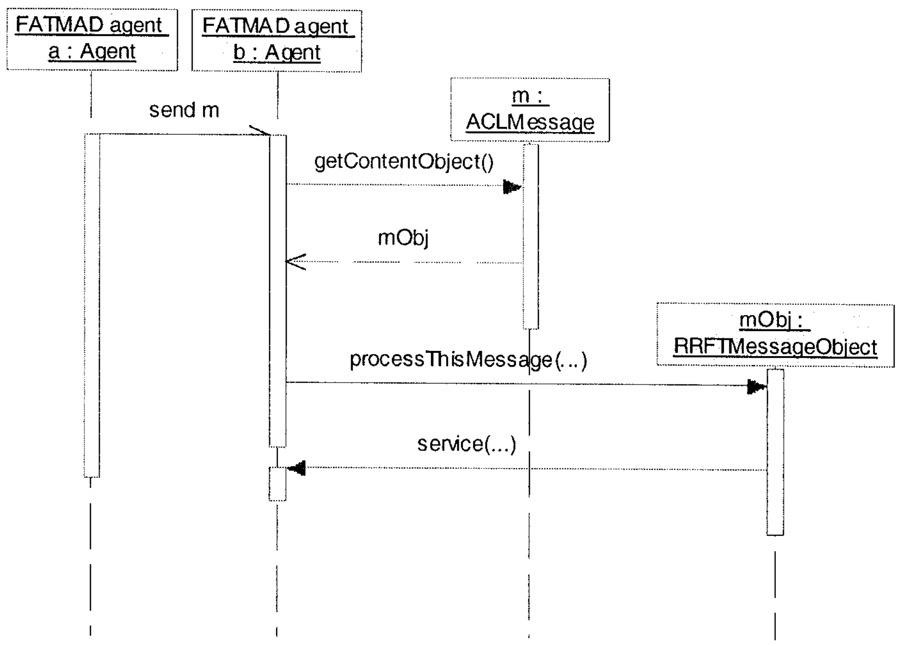


Figure 5-5 FATMAD messaging mechanism

5.4.2 Two options of system message transmission

There are generally two ways of transmitting a FATMAD system message:

i) Transmitting via a dedicated system message:

This mechanism can be applied in all occasions.

ii) Piggybacked by an application message:

For some checkpoint/logging protocols, the designer can use agent application message as a carrier to transfer protocol messages.

5.5 Deployment

FATMAD runtime provides flexibility on the deployment of its agent components. Except for the *container proxy agent*, that each agent container should have exactly one *container proxy agent* deployed, the locality and the quantity of *FD monitor agent* and *repository manager agent* are allowed to be configured by a system deployment manager.

Usually a deployment strategy should be carefully designed based on the characteristics of the application agents, e.g., failure assumption, locality and performance issues.

Chapter 6 Protocol skeleton

In the last chapter, we introduced the *FATMAD runtime*. The *FATMAD runtime* provides basic services and a runtime environment that allows an application developer to deploy a fault-tolerant application with an embedded rollback-recovery protocol. In order to provide support towards protocol development, we design a *protocol skeleton* that can help protocol designers to implement a protocol with much reduced workload. In this chapter we introduce the *protocol skeleton* in detail.

6.1 The generic rollback-recovery protocol

The objective of the protocol skeleton is to provide a template that outlines the generic rollback-recovery protocol and can be easily extended to implement a broad range of protocol variations.

In general, the checkpoint/logging protocol for an agent can be viewed as a sequence of atomic actions combined with coordination actions. There are two types of atomic actions: checkpointing actions and message logging actions. All actions are triggered when some particular conditions are satisfied. A *policy* hence can generally refer to an action and its triggering condition. Coordination actions induce dependencies among agents. These features can be supported as a generic checkpoint protocol (i.e., a generic behavioral pattern) by the framework, which is shown in a high level of abstraction as follows:

```
Upon checkpoint event for agent $a_i$:
   Take a local checkpoint;
   Update logging policy locally;
   Send checkpoint request to a subset of agents;
   Wait for feedback from a subset of agents;
   Send checkpoint commitment to a subset of agents;
   Do logging coordination with a subset of agents
   and update group logging policy;
```

A specific checkpoint protocol is a refinement of the generic protocol (by the protocol designer). In general, in a checkpoint protocol, the designer needs to specify the following for an agent:

i) The checkpoint policy: When a local checkpoint is taken (i.e. triggered by some specific checkpoint events like application's flag-to-checkpoint) and optional checkpoint actions (e.g. whether the mailbox is included in a checkpoint);

ii) The message logging policy related to that agent as well as changes to the policy (e.g. start or stop logging a channel);

iii) Dependencies among checkpoint and logging events taken at different agents, if any;

iv) Defining the coordination group and the coordination method.

The action of message event logging is implemented as a FATMAD runtime service that is governed by a logging policy. All logged messages and checkpoints are retrievable from some *storage manager* that can survive node crash.

When an agent failure has been detected, a recovery protocol will usually perform the following:

i) Gather necessary information (checkpoints and message logs) from the repository manager and decide on a recovery line involving one or more agents that should rollback. This is protocol specific.

ii) Enforce the rollback with an appropriate recovery policy such as replay and discard of messages.

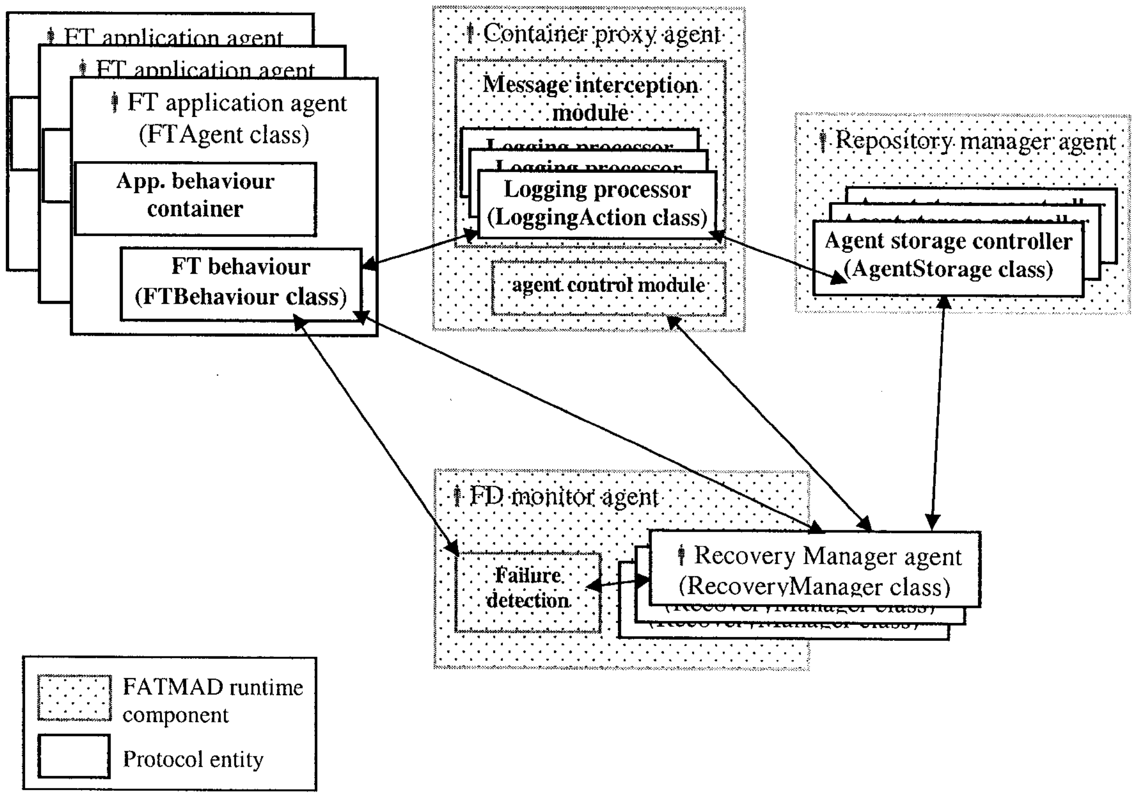
The above two steps can be modeled as a sequence of two atomic actions. Hence the framework involves a simple abstract recovery protocol. While a checkpoint protocol involves a logging policy, a recovery protocol similarly involves a message handling policy upon agent recovery. In addition, the recovery policies may also include recovery synchronization, message channel flush, and reincarnation control.

The generic checkpoint/logging and recovery protocols can be refined to many different rollback-recovery protocols. For example, during failure-free execution, uncoordinated checkpoint protocols only involve checkpoint actions, while coordinated checkpoint protocols may involve all types of atomic actions but differ in their policy control and coordination schemes.

As we already introduced, *FATMAD runtime* implements the essential framework services supporting all protocols. The protocol skeleton actually implements the atomic action control and the policy triggering mechanism for the generic protocol(s).

6.2 Structure of the protocol skeleton

The *protocol skeleton* is designed with a set of classes including *FTAgent* class, *FTBehaviour* class, *LoggingAction* class, *AgentStorage* class, and *RecoveryManager* class. Designing a concrete protocol requires a protocol designer to extend or customize these classes in order to implement protocol specific behaviours.



FT application agent (FTAgent class)
App. behaviour container
FT behaviour (FTBehaviour class)
Container proxy agent
Message interception module
Logging processor (LoggingAction class)
agent control module
Repository manager agent
Agent storage controller (AgentStorage class)
FD monitor agent
Failure detection
Recovery Manager agent (RecoveryManager class)
FATMAD runtime component
Protocol entity

Figure 6-1 Protocol skeleton structure

Each *FT application agent* embedded with a concrete rollback-recovery protocol is associated with a set of *protocol entities*, i.e., a set of objects instantiated from the *protocol skeleton* classes or their subclasses. Each *protocol entity* is integrated into the *FT application agent* or the relevant *FATMAD runtime* component accordingly when the associated agent is launched into the platform. *Protocol entities* are unloaded when the *FT application agent* is removed from the Jade platform. The diagram in **Figure 6-1 Protocol skeleton structure** illustrates a refinement of the FATMAD architecture, which shows a set of *FATMAD runtime* components and the deployed protocol entities. The directed lines in the diagram illustrate the communication among different protocol entities.

In the following we introduce these *protocol skeleton* classes:

1. **FTAgent class**

   This class defines the structure of an *FT application agent* that integrates application behaviours, namely application code, with fault tolerance behaviours, namely fault tolerance code. In addition, *FTAgent* class is designed to integrate a rollback-recovery protocol, taking into account all relevant protocol entities, and deploy them to *FATMAD runtime* at runtime.

2. **FTBehaviour class**

   This class is designed to construct fault tolerance modules, which are embedded into each *FT application agent*, namely an *FTAgent* object. An *FTBehaviour* instance, integrated into each *FT application agent*, carries out fault tolerance roles as illustrated in the diagram in Figure 4-6 Role mapping strategy. With the *FTAgent* class support, an *FTBehaviour* object is able to perform functions such as checkpointing, transmitting "alive" report, execution control, etc.

3. **LoggingAction class**

   As was already mentioned in the last chapter, this class is designed to play the *message event logger* role in a checkpoint/logging protocol. When an *FT application agent* registers to the system, a relevant *LoggingAction* object will be set into the local *container proxy agent*. It is removed when the agent deregisters.

At runtime, this object will be notified whenever a related message event occurs. *LoggingAction* class provides a set of options to allow protocol designers to specify their own logging policy, such as what message events should be logged or should not be logged. The specified logging policy can also be updated dynamically. Protocol designers can either customize existing options provided by this class or even override relevant message event processing methods to write their own options.

4. **AgentStorage class**

   This class is designed to play the *storage manager* role in a checkpoint/logging protocol. It manages recovery related data for one registered agent in a *repository manager agent* and provides interface for other components to access its stored data. An *AgentStorage* class object is automatically generated by a *repository manager agent* when an agent registers to it. It is removed automatically when the agent deregisters.

5. **RecoveryManager class**

   This class is designed to play the *recovery reactor* role and *recovery manager* role in a recovery protocol. When an *FT application agent* registered to a *FD monitor agent*, a *RecoveryManager* object should be supplied to encapsulate a predefined failure reaction scheme. Whenever a failure is detected, this object will be notified and the predefined action will be triggered by the *FD monitor agent.*

   Upon triggering, the *RecoveryManager* object is transformed into an active agent on the agent container that allows it to directly access the failed agent's *storage manager.* However, the detailed recovery plan must be specified by a protocol designer in the *recoveryAction(..)* method of a subclass of *RecoveryManager* class.

The detailed functionality of the protocol skeleton classes will be explained in section 6.3.

6.3 *FT application agent* skeleton

Designing a fault tolerant application requires a concrete rollback-recovery protocol to be integrated with the application agents. FATMAD provides an application skeleton class, i.e. *FTAgent* class, which can be used to construct *FT application agents* and perform this kind of integration. As a skeleton, it is expected to be extended in different protocol extensions. In this section, we briefly introduce the *FT application agent skeleton.*

6.3.1 Super class of *FT application agents*

The class diagram in Figure 6-2 **Class diagram of the *FT application agent*** illustrates the design of the generic *FT application agent.* A generic *FT application agent* is defined by *FTAgent* class that extends the Jade Agent class. *FTAgent* class is designed to be a super class for all application agents that need to be fault tolerant. In other words, all FATMAD supported fault tolerant application agents must inherit *FTAgent* class. *FTAgent class* is usually extended in different protocol extension. Hence, the extension class must be inherited by the application agent classes in order to apply the fault tolerance protocol to an application using FATMAD.

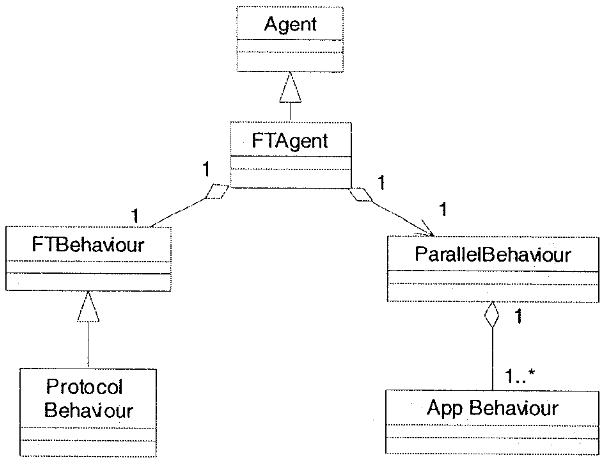


Figure 6-2 Class diagram of the *FT application agent* skeleton

6.3.2 Internal structure of an *FTAgent* object

An object inheriting *FTAgent* class should contain an *FTBehaviour* object and a *ParallelBehaviour* object. An *FTBehaviour* object encapsulates fault tolerance roles in an *FT application agent*. *ParallelBehaviour* class defined in Jade is a composite behaviour class that can contain many application behaviour objects and execute these behaviour codes concurrently. The *ParallelBehaviour* object in an *FTAgent* object is designed as an application behaviour container that contains all application behaviour objects. Under such a structure, both fault tolerance roles and application roles are integrated into an *FT application agent*.

6.3.3 Concurrency mechanism

As was introduced in chapter 2, a Jade agent executes its contained behaviour programs concurrently. The concurrency mechanism in the application behaviour container, i.e., the *ParallelBehaviour* object, is similar to the counterpart in the Jade Agent class. They all implement a scheduler that rotates behaviour programs based on *atomic behaviour action* blocks defined by the developer.

Since an *FTAgent* object only has two behaviour objects, according to Jade agent scheduler mechanism, the *FTBehaviour* object always has a chance to execute before or after any *atomic behaviour action* of application behaviours. By this way, some necessary fault tolerance actions can be inserted into the agent's execution.

6.3.4 Agent registration

FTAgent class implements a registration mechanism that allows *FT application agents* to request *FATMAD runtime* services. When an *FT application agent* is loaded to a Jade platform, the agent will register itself to the local *container proxy agent*, a *FD monitor agent*, and a *repository manager agent.* A deregistration action is automatically performed when an *FT application agent* is removed from the platform. During agent registration these *FATMAD runtime* components require relevant protocol entities as parameters. For example, a *LoggingAction* object is required when registering to the local *container proxy agent*; a *RecoveryManager* object is required when registering to a *FD monitor agent.*

FTAgent class provides a interface to allow protocol developers to specify these parameters.

6.3.5 API

FTAgent class is expected to be extended by protocol developers to construct their own *protocol extensions*. The extended class can then be used by application developers to construct *FT application agents*.

FTAgent class provides APIs for protocol development, application development, and agent deployment. The relevant APIs will be explained when we introduce how to use FATMAD to develop protocols and fault tolerant applications in the next two chapters.

6.4 Functional design of the *Protocol skeleton*

The design of the *protocol skeleton* focuses on outlining generic protocol patterns and providing methods to allow protocol-specific behaviour to be integrated.

In this section, we introduce the functionalities of the *protocol skeleton* and how it can support protocol development in detail.

6.4.1 Checkpoint/logging protocol

The diagram in Figure 6-3 illustrates a static view of a checkpoint/logging protocol supported by the *protocol skeleton*. A checkpoint/logging protocol executed by a set of agents can be modeled as the following:

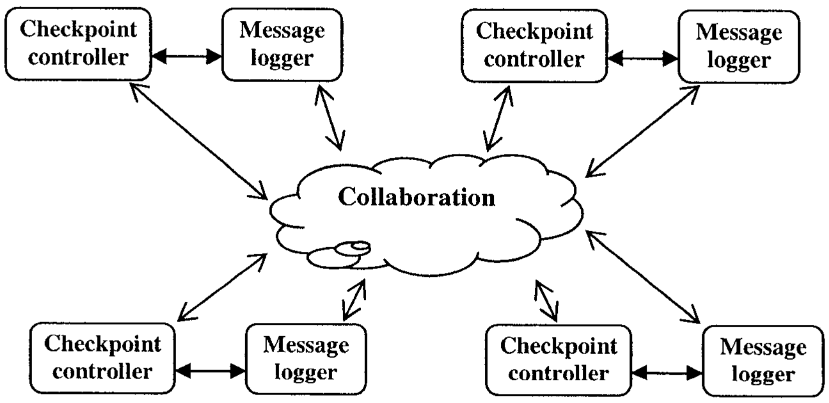


Figure 6-3 Checkpoint/logging protocol over a set of agents

(i) For each agent, there are two concurrent and correlated roles involved: *checkpoint controller* and *message event logger*.

The *checkpoint controller* role is implemented in the *FT application agent skeleton* (*FTAgent* class and *FTBehaviour* class) that encapsulates the functionalities of a checkpoint action and its triggering mechanism. The *message event logger* role is implemented in *LoggingAction* class that encapsulates the functionalities of a logging action and its triggering mechanism (in the form of logging policy).

(ii) In a checkpoint/logging protocol, *checkpoint controllers* and *message event loggers* of different agents execute collaboratively.

When designing a checkpoint/logging protocol, one should specify the checkpoint scheme as well as how logging actions and collaboration actions are involved. The specification includes:

(a) How to take a checkpoint;

(b) How to trigger a checkpoint protocol;

(c) How logging action and collaboration action are involved;

(d) Initial logging policy or logging policy that is unrelated to checkpoint actions.

The protocol skeleton implements the above mechanisms and allows protocol developers to specify these details in their protocol extensions. We will examine these issues in the remaining of this subsection.

1) Checkpoint action

In FATMAD, the *checkpoint controller*, namely an *FTBehaviour* object, in each *FT application agent* performs checkpoint actions for an agent. Once a checkpoint is taken, it will be sent to a *repository manager agent* that it has registered to.

Checkpoint = Agent object state

In FATMAD, an agent checkpoint is a snapshot of an agent state, which can be used to reconstruct an agent's execution. Since the Java implementation of Jade doesn't allow us to capture the Java thread execution state, we cannot checkpoint an agent state at arbitrary point of the agent program. In other words, checkpoints should only be taken in the code where the agent can be restarted without loading the execution context and the stack of the threads.

In FATMAD, we utilize the Jade agent programming model, as illustrated in chapter 2, to checkpoint only those agent states that satisfies the following conditions:

i) The object state is a consistent state involving all *application roles* in the agent.

ii) All application roles in the agent can be reconstructed and resumed by restoring from the checkpoint.

If the above conditions can be satisfied, the thread execution state of an agent is not required to checkpoint.

Checkpoint timing constraint

However, conforming to the above conditions leads to an implementation constraint that a checkpoint can not be created in the middle of an ***atomic behaviour action.*** Checkpoints can only be taken between any two consecutive ***atomic behaviour actions.*** The *FT application agent skeleton* implements a checkpoint controlling mechanism to guarantee that all checkpointing actions satisfy these conditions. However, this assumes that no additional thread is generated by application agents.

This method has the following consequences:

i) The states of all application behaviours objects are saved into a checkpoint;

ii) *FTBehaviour* object state must be ignorable.

By this way a checkpoint can only be taken either before or after each *atomic behaviour action* of an agent.

The checkpoint timing constraint must be carefully considered when designing a checkpoint protocol. For instance, some protocol, such as the protocol specified in [ChanLamp85], must be modified in order to adapt to this implementation constraint.

Implementation condition

When taking a checkpoint for a process, usually we need to freeze the process execution in order to create a checkpoint correctly. Under the structure of an *FT application agent*, checkpoint actions implemented in *FTBehaviour* class is handled in the agent native thread. This condition is automatically guaranteed if no any application behaviour in the agent spawns additional thread.

Take message queue with checkpoint

As we already introduced in chapter 2, each agent in Jade has a message queue that stores incoming messages of this agent. From Jade programming API, we found that application developers are allowed to selectively pickup incoming messages from its message queue. This kind of message picking actions may violate FIFO (first in first out) message delivery order, which is required by many distributed protocols. In order to solve this problem, we provide an option to allow the message queue to be included into an agent checkpoint, since message delivery path from message-sent event to message-posted event conforms to the FIFO ordering rule.

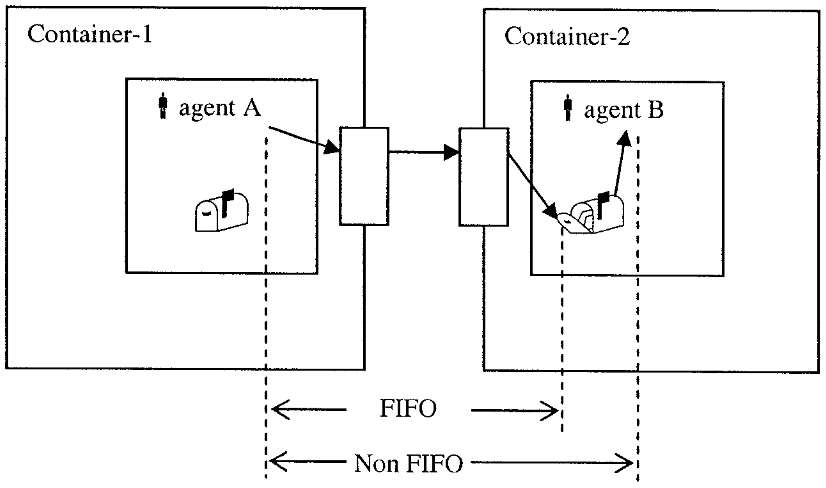


Figure 6-4 FIFOness of message delivery

By this way, an agent's message queue becomes a part of its checkpoint state and the message delivery path is shortened to the segment from sending a message by an agent to posting it to the receiver agent's message queue by the platform. The diagram in Figure 6-4 illustrates the differences between the two message delivery paths. *FTBehaviour* class provides an interface to allow protocol designers to specify this option.

2) Checkpoint triggering mechanism

A checkpoint action is triggered by a checkpoint event. Upon receiving a checkpoint event a specific checkpoint protocol at each individual agent will start. The checkpoint triggering mechanism is implemented in *FTBehaviour* class in the *protocol skeleton*, where the following types of checkpoint events are defined:

a) **Application-alerted checkpoint event**

   Events of this type can be generated by an agent application in which the application developer can decide when to trigger a checkpoint.

   FTAgent class provides a *flagToCheckpoint()* method so that application developers can invoke it from their program. Invoking this method does not trigger a checkpoint action right away. Instead, it sets a flag to notify the underlying *checkpoint controller* so that a checkpoint action will be triggered when the current *atomic behaviour action* is done.

b) **Timer-alerted checkpoint event**

   Events of this type are generated by the checkpoint timer service implemented in *FTBehaviour* class. When an *FT application agent* is loaded into a Jade platform, the checkpoint timer service will start to count number of atomic behaviour actions and check if the number satisfies predefined timing condition. Whenever the condition is satisfied it will generate a checkpoint event.

   The timing condition can be specified either by programming it into application code or by setting it through the deployment tool provided in FATMAD when the agent is loaded. FATMAD provides a deployment tool that can be used for this purpose.

c) **Message-alerted checkpoint event**

   A message-alerted checkpoint event is generated upon receiving an alert-to-checkpoint message. An alert-to-checkpoint message is usually sent from other agents by the relevant *checkpoint controller* or *message event logger*.

A checkpoint alert message can be either transmitted as a system message or piggybacked by an application message.

d) **Predicate-alerted checkpoint event**

FATMAD allows protocol developers to define some complex checkpoint events by specifying predicate conditions on triggering checkpoint events. The predicates can be specified by extending *FTBehaviour* class and overriding a Boolean method *evaluateCheckpointPredicate()* in the subclass.

3) Checkpoint related action

In a generic checkpoint/logging protocol, a checkpoint action is usually combined with logging policy change and some collaboration actions. FATMAD skeleton divides these related actions into two parts:

i) Actions that must be combined with the checkpoint action atomically

For example, some protocol requires that a checkpoint action must be combined with updating logging policy and notifying other agents to checkpoint atomically.

To support this atomicity, the *protocol skeleton* implements an atomic action block for each checkpoint action. Such an atomic block includes a sequence of actions: invoke the *preCheckpointAction(...)* method, take a checkpoint, and invoke the *postCheckpointAction(...)* method. Protocol developers can override the two methods in a sub class of *FTBehaviour* class to incorporate those actions that must be tightly combined with a checkpoint action.

Whenever a checkpoint event occurs, the *preCheckpointAction(...)* method will be invoked before the checkpoint action and the *postCheckpointAction(...)* method will be invoked after the checkpoint action. They are executed atomically. The atomicity guarantees that no other program in this agent is executed and no incoming message is posted to the agent's message queue during the execution of this atomic action.

ii) Actions that are not required to be combined with the checkpoint action atomically

These actions may involve checkpoint/logging collaboration or logging policy change. They are usually executed in an asynchronous manner.

Agent collaborations in a checkpoint/logging protocol are implemented through message communication, in particular, via FATMAD system messages defined by *RRFTMessageObject* interface. FATMAD *protocol skeleton* provides some predefined message formats for some particular purposes. For example, *AlertToCheckpoint* class defines alert-to-checkpoint message that can be used to generate message-alerted checkpoint events. When an agent receives an *AlertToCheckpoint* object, it will automatically trigger a local checkpoint action.

The protocol skeleton also provides a *LoggingMessage* interface that defines collaboration message format for sending a message to the event logger of an agent.

4) Logging policy

As we already mentioned, message event logging in FATMAD is implemented as a service that is governed by the relevant logging policy. FATMAD implements a set of optional logging policy control in the LoggingAction class. The *LoggingAction* class and the *container proxy agent* together realize the logging mechanism. A protocol designer can specify the logging policy by instantiating the *LoggingAction* class, customizing the instantiated objects, and integrating the objects with relevant agents, so that message events related to the agent can be logged according to the predefined policy.

The logging policy for an agent is usually initialized when an agent is loaded into a platform and/or can be updated dynamically.

Logging policy for message recording can be classified into two categories:

i) Checkpoint related message logging

   In a checkpoint protocol, logging actions are usually regarded as recording channel state, which is part of a checkpoint.

ii) Non-deterministic message logging

This type of message logging is used to record messages from outside world process [EAWJ02].

LoggingAction class provides a set of methods to allow protocol developers to specify the logging policies. Such as:

i) What types of message events should be recorded:

   - message-sent: sender side logging

   - message-posted: receiver side logging with FIFOness messaging,

   - message-received: receiver side logging without FIFOness messaging.

ii) Which message channel should be logged or which message channel should not be logged.

Some protocol may require an agent's message event logger to record the number of messages that have been sent to some particular channels since last checkpoint so that these numbers can be used to avoid duplicate messages being sent after recovery. *LoggingAction* class can also be specified to record messages that should be blocked from sending out after recovery.

Many logging policies including those mentioned above can be specified through the *LoggingAction* class. However, the available logging control options provided in the *LoggingAction class* may not be adequate for some protocols. The protocol designer can extend *LoggingAction* class and override some methods to implement her specific logging policy.

5) A generic checkpoint protocol illustration

The diagram shown in Figure 6-5 illustrates a generic checkpoint/logging protocol as well as separated responsibilities among the FATMAT kernel (including *FATMAD runtime* and *protocol skeleton*), *protocol extensions* and application behaviour for each agent. In the diagram, there are two rectangle blocks, each of which represents an agent. The left agent takes an independent role in a checkpoint-logging protocol, which means that the agent itself generates checkpoint events, and the right agent takes dependent role in the protocol, which means that checkpoint actions of this agent are triggered by message-alerted checkpoint events.

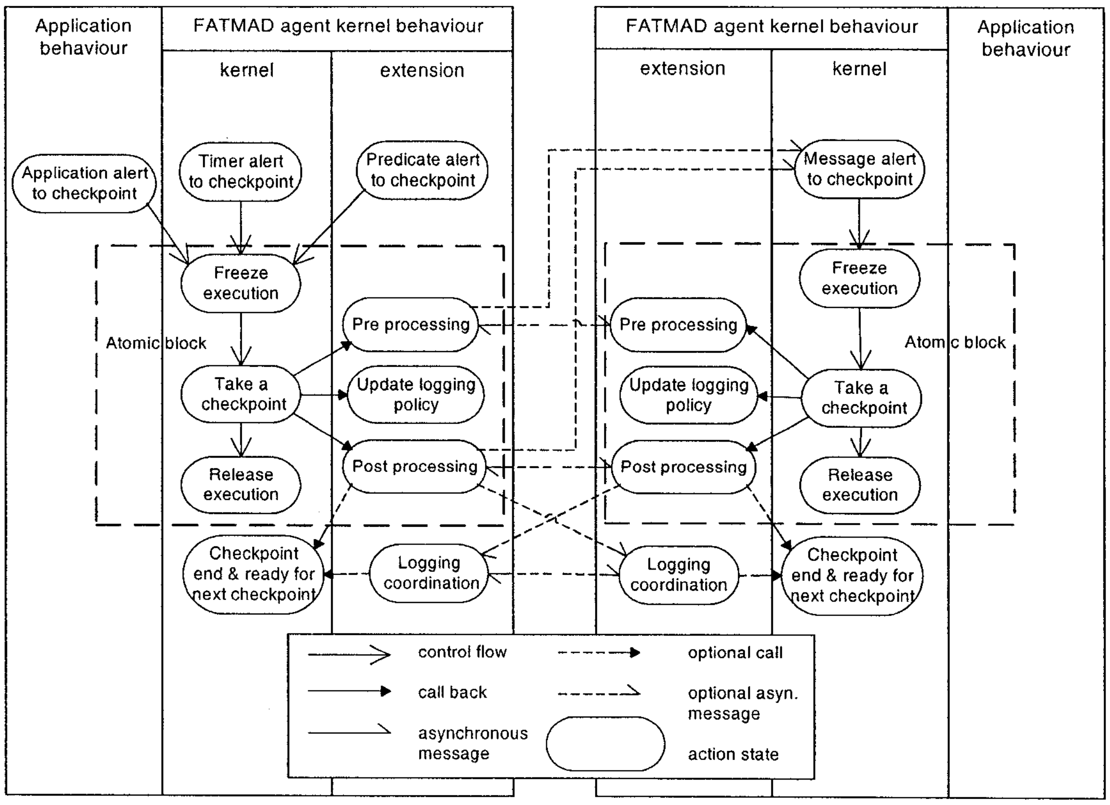


Figure 6-5 A generic checkpoint/logging protocol

6) Distributed timing service

There is no global clock in distributed systems. Some checkpoint/logging protocols may need to implement distributed timing scheme, such as logical clock [Lamport78] or vector clock [SchMat92], to implement synchronous actions among distributed agents.

FATMAD skeleton implements a message tagging mechanism in order to provide timing services.

The distributed timing service in FATMAD is designed as follows:

(i) Each agent in a protocol maintains a clock.

(ii) Whenever an agent sends a message, it is tagged with the local clock value and then the local clock value increases by one.

(iii) When the message is posted to the receiver agent's message queue, the tagged clock value will be merged with the receiver agent's clock value and the merged value increases by one. Then the receiver agent's clock value will be updated with the new clock value.

The diagram in Figure 6-6 illustrates this mechanism. By modifying the source code of the Jade *ACLMessage* class, FATMAD allows application messages (ACL) to be tagged with a clock value defined by *Clock* class.

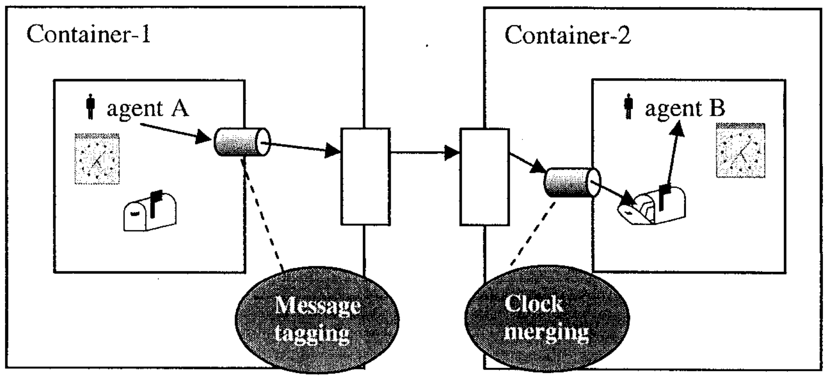


Figure 6-6 Distributed-timing service

We design the Clock class as an abstract class to represent clock values and apply a strategy pattern to implement this mechanism so that different clocking schemes can be easily applied to adapt to different protocol requirements.

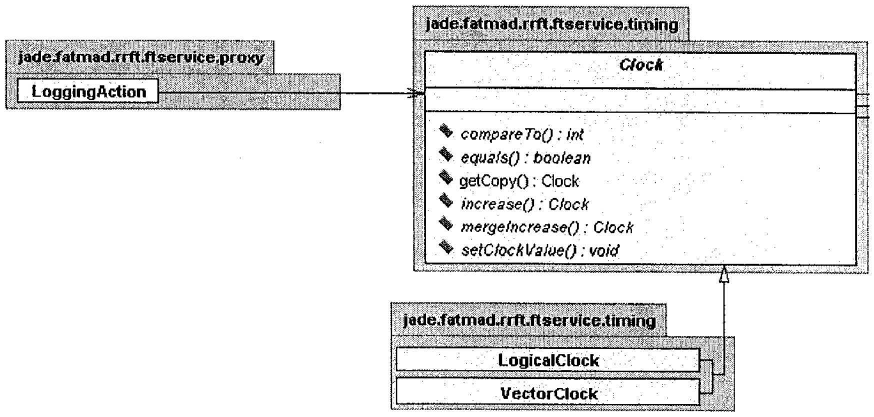


Figure 6-7 Class diagram for distributed timing service

To design a clocking scheme, a protocol designer needs to design a subclass of Clock class and implement a set of methods, such as *increase()*, which defines how a clock clicks, i.e. increases its value, *mergeIncrease(..)*, which defines how a clock merges with other clock value and increases the merged value, etc.

FATMAD *protocol skeleton* currently provides two built-in clocking schemes: logical clock and vector clock.

6.4.2 Recovery protocol

1) Recovery procedure

As was introduced in the last chapter, the *FD monitor agent* triggers a recovery protocol whenever an agent failure is detected. The preconfigured recovery scheme is encapsulated into a *RecoveryManager* object. Upon triggering, it will be transformed into a *recovery manager agent* on the agent container where the failed agent's *storage manager* is locally available.

The *protocol skeleton* leaves the remaining implementation to the protocol developer when designing a *protocol extension*. The remaining action should be implemented in a sub class of RecoveryManager class by overriding the method:

public abstract void recoveryAction(...)

As discussed in section 6.1, a generic recovery protocol includes three actions:

i) Gather necessary information:

The *recovery manager agent* needs to access recovery related information of the failed agent as well as other related agents.

Recovery related data for a set of agents can be either centralized or distributed. For centralized data, the *recovery manager agent* may directly access the local repository via each agent's *storage manager*'s API. For distributed data, it needs to access remote data by sending messages.

Protocol designer can utilize FATMAD messaging mechanism to design protocol-specific system messages to access distributed data.

ii) Decide on a recovery line:

Based on collected recovery-related information, the recovery manager agent needs to decide on a recovery line involving a set of agents.

iii) Enforce the rollback:

Once a recovery decision is made, it should be implemented right away. The failed agent need to be recovered, other related agents should be rolled back.

The first two actions are to be implemented by protocol developers. Since the third action involves a set of complex recovery options, the protocol skeleton encapsulates the complex recovery functionalities and provides a customizable service to the protocol developer. By this way, the protocol developer can implement her recovery decision by specifying necessary recovery parameters in *recovery messages* (defined by *RecoveryMessage* class). The protocol developer is not required to be involved in detailed implementation.

After dispatching the customized recovery messages, the rest of work will be handled by FATMAD automatically.

2) Agent recovery options

One *recovery message* can be used to roll back only one agent. The dispatched *recovery message* will be sent to the container proxy agent in the container where the agent is to be rolled back. Upon receiving the message, the recovery plan specified in the *recovery message* will be executed by the container proxy agent. It will automatically set up post recovery options for the agent to be rolled back according to the recovery plan, and then reincarnate the agent.

After agents in the recovery line are reincarnated on the platform, there are still some problems that should be handled.

Since agents are reincarnated in distributed location concurrently, old agents and new agents may coexist for a period of time. Such coexistence may cause messages to be messed up and lead to inconsistency. The diagram in Figure 6-9 illustrates two possible scenarios that messages may be incorrectly delivered if we don't handle them well. The two scenarios are:

Scenario 1: A message sent from an old agent may be received by a new one.

Scenario 2: A message sent from a new agent may be received by an old one.

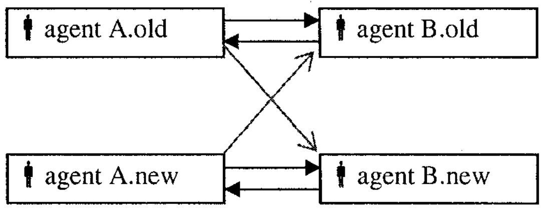


Figure 6-8 Messages may mess up

To solve such a problem, several techniques may be applied.

(a) Flush message channel

In this method, we request each recovered agent to flush its outgoing channels with a marker message before sending any new messages. Each newly recovered agent will discard messages from each incoming message channel until receiving a marker from that channel.

This method can eliminate the problem in scenario 1.

(b) Recovery synchronization

All recovered agents freeze their execution until they are all synchronized. In the protocol skeleton, the synchronizer is designed to be played by the *recovery manager agent.*

This method can eliminate the problem in scenario 2.

(c) Reincarnation control

As we mentioned in last chapter, Java VM doesn't allow a Java thread to be killed right away by other Java threads. After reincarnating an agent, the old agent and the new agent with same identity may coexist for a period of time. Therefore, the old agent must be isolated from its environment. For example, it should not be allowed to send a message out to any other agent.

FATMAD utilizes reincarnation number to identify old agents. Each agent has an initial incarnation number when it is loaded to the platform. When it is reincarnated, that number will be increased by one. The *FATMAD runtime* environment always keeps the up-to-date reincarnation number value for each *FT application agent.* Whenever an agent sends a message, the reincarnation number will be validated if it is identical to the one stored in the *FATMAD runtime* system. If the result is false, the message will be blocked and the agent will trigger a self-deletion mechanism automatically.

Besides the above actions, an agent has to handle messaging actions invoked by the application behaviours. This involves the following message handling policy to be specified:

i) Some logged messages need to be replayed when the application behaviour invokes *receive() method*;

ii) Some duplicated messages to be sent by the new agent should be blocked and discarded.

Moreover, a recovered agent needs to notify the *repository manager agent* and the *FD monitor agent*, to which it has registered, so that their services for the agent can be resumed. Upon receiving such a notification, the *repository manager agent* will continue to accept new checkpoints and new message events related to this agent, and the *FD monitor agent* will continue to detect failures for this agent. This is the default action implemented in the *protocol skeleton.*

All the above functionalities are common to different recovery protocols. These services are already implemented in *FTBehaviour* class. Protocol developers can specify these options in the recovery messages to be dispatched.

Chapter 7 Protocol extension and case study

In the last chapter, we have introduced FATMAD *protocol skeleton*, which provides support for protocol development. In this chapter, we introduce how to develop specific rollback-recovery protocols with FATMAD.

7.1 Protocol extension

As we already discussed in chapter 4, a *protocol extension* is a concrete protocol implementation that can be applied to agent applications in order to achieve fault tolerance. The reason why we name it as *protocol extension* is because the implementation of a protocol is an extension of our *protocol skeleton.*

The *protocol skeleton* provides a generic structure that can be applied to variant protocols. Moreover, it provides customizable services that can significantly reduce implementation time in protocol development. Developing a protocol using FATMAD, in fact, is to develop a *protocol extension*, which concretizes the *protocol skeleton* by filling in protocol-specific behavioural details and providing an interface for application developers so that the protocol can be integrated into agent applications.

Developing this protocol in FATMAD involves the following steps:

1) Protocol decomposition

The protocol can be decomposed into a set of actions or decision points as follows:

a) How checkpoint action is triggered?

b) What checkpoint option should be applied?

c) How checkpoint affects logging?

d) How different agents are collaborated in terms of checkpointing and logging?

e) What is the initial logging policy?

f) How to decide a recovery line?

g) How recovery of each agent should be performed?

2) Design and code extension classes

The designer needs to analyze each decomposed protocol actions to see if they can be implemented by customizing relevant *protocol skeleton* classes. If a protocol action is not customizable, an extension of relevant skeleton class might need to be created so that the protocol designer can implement the protocol action into it.

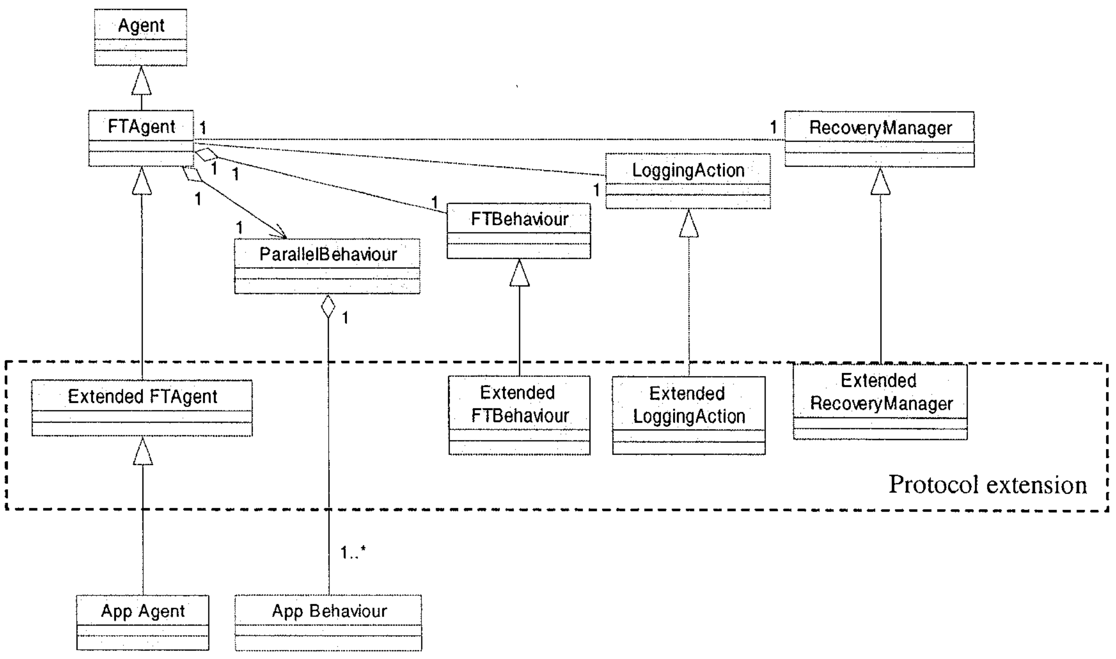


Figure 7-1 Class diagram for *protocol skeleton* and generic protocol extension

The following *protocol skeleton* classes are generally related:

- **FTBehaviour class**

  With this class or its subclass we can specify checkpoint triggering mechanism, checkpoint option, checkpoint related action, agent post-recovery control, etc.

- **LoggingAction class**

  With this class or its subclass one can specify logging policy as well as logging related collaboration among a group of agents.

- **RecoveryManager class**

  With this class or its subclass one can specify recovery line decision and recovery plan for each agent.

- **FTAgent class**

  The subclass of *FTAgent* class plays a wrapper role for an *FTBehaviour* object. We may utilize this in protocol action implementation.

In addition to the above classes, a protocol designer may design other classes to assist implementing protocol actions. For example, the protocol designer may design some message class to utilize FATMAD messaging mechanism to implement checkpoint collaboration. A protocol designer may need to implement his/her own clocking scheme for the checkpoint/logging protocol.

The class diagram in Figure 7-1 Class diagram for *protocol skeleton* and generic protocol extension illustrates the relationship between *protocol skeleton* classes and generic *protocol extension* classes.

3) Integration

The designer needs to integrate all implemented protocol actions and provides an interface that can be used by application developers.

The integration job should be done by designing a subclass of *FTAgent* class. In the subclass, the protocol designer needs to override the method: *ftSetup()*, in which protocol entities are instantiated, customized, and integrated.

In the *ftSetup()* method, the protocol designer may use the following methods to specify protocol entities that are to be used at runtime.

setFTBehaviour(...); -- This method is used to specify an *FTBehaviour* object.

setLoggingAction(...); -- This method is used to specify a *LoggingAction* object.

setEventReaction(...); -- This method is used to specify a *RecoveryManger* object.

The *ftSetup()* method in an *FT application agent* is invoked when the agent is launched into the platform. The protocol designer can also provide some protocol specific initialization code in this method.

The protocol designer can also override *ftTakedown()* method in a sub class of *FTAgent* class to implement some protocol-specific clean-up action if necessary. This method will be invoked when an agent is removed from a platform.

In addition, we must be aware that the subclass of *FTAgent* class in a protocol is the only interface that application developers need to know and work with. Therefore, a protocol designer needs to clean up the necessary interface for application developers.

7.2 Case study: Log-based rollback-recovery protocol

In this section, we exemplify the design of a protocol using FATMAD. The protocol example is Log-based rollback-recovery protocol.

7.2.1 Protocol description

The Log-based protocol is suitable for implementing fault tolerance features for individual agents. By recording all ACL message events, log-based rollback-recovery protocol enables a failed agent to be recovered to the most recent message event without requesting other agents to roll back. It can largely reduce recovery cost compared to checkpoint based rollback-recovery protocols. However, recording all message events is a considerable overhead during error free execution and it could lower the application performance if the agent has large volumes of message events with high frequency.

In this example, we implement the pessimistic logging algorithm [AlMa98], which can be decomposed into the following actions:

1) **Triggering checkpoint:**

   A checkpoint action can be triggered automatically in a periodical pattern or triggered from the application code written by an application developer. Periodically triggered checkpoint action can be implemented by timer-alerted checkpointing service provided by *FTBehaviour* class. *FTAgent* class also provides a wrapper method: *flagToCheckpoint(),* that can be used to trigger checkpoint action by application developers.

2) **Checkpoint action option:**

   A checkpoint includes the agent's message queue. *FTBehaviour* class provides a method to specify this.

3) **Message logging policy:**

   All message-sent events should be counted and all message-posted events should be recorded. Whenever a checkpoint is taken, message-sent event counter should be reset to 0. These actions must be executed in a blocking mode, which means the agent application must freeze its execution when a message is recorded.

   This is already implemented in *LoggingAction* class. We can simply customize the provided service to implement this policy.

4) **Recovery line decision:**

   The failed agent should be recovered to the most recent checkpoint and rolled forward with all events recorded since the most recent checkpoint.

   We should implement this action in an extension of *RecoveryManager* class.

5) **Post recovery control:**

   Logged incoming messages since last checkpoint will be played back while the agent application invokes *receive()* method. Duplicate messages will be discarded when the agent invokes *send()* method.

   The implementation of this action is provided in *FTBehaviour* class. We can simply specify it in a recovery message (*RecoveryMessage* object) that will forward the directive to the recovered agent.

7.2.2 Design

Based on the above protocol actions, we implemented a *protocol extension* for log-based rollback-recovery protocol. The *protocol extension* includes two classes:

LogBasedFTAgent class

This class is designed to integrate the following protocol entities: an *FTBehaviour* object, a *LoggingAction* object, and a *LogBasedRecoveryManager* object (see Figure

7-2 LogBasedFTAgent class source code). In this class, we implement a method: *ftSetup()*, in which these three classes are instantiated, customized, and integrated into LogBasedFTAgent class.

```
public abstract class LogBasedFTAgent extends FTAgent
{
 public void ftSetup()
 {
  try
  {
   FTBehaviour rrcb = new FTBehaviour(this);
   rrcb.setTakeMsgqueueWithCheckpoint (
       FTBehaviour.CHECKPOINT_WITH_WHOLE_MESSAGE
QUEUE );
   super.setFTBehaviour(rrcb);
   LoggingAction la = new LoggingAction(){};
   la.disableCacheCounters();
   la.startLogAllPosted(null);
   la.startCountAllSent(null);
   super.setLoggingAction(la);

   EventReaction reaction =
     new LogBasedRecoveryManager();
    super.setEventReaction(reaction);
  }
  catch (Exception ex1)
  {
   ex1.printStackTrace();
  }
 }
}
```

Figure 7-2 LogBasedFTAgent class source code

LogBasedRecoveryManager class

This is a subclass of *RecoveryManager* class, in which we implement recovery protocol actions. In the *recoveryAction()* method, the supplied argument provides a way to allow us to access the failed agent's storage manager, by which we can get the agent's most recent checkpoint as well as recorded messages and message sent event counters.

To recover the failed agent, we create a *RecoveryMessage* object and we customize it by specifying recovery checkpoint, messaging control policy, and the agent container to be recovered. Finally we dispatch the message. The Figure 7-3 shows the detailed code of this method.

```
public class LogBasedRecoveryManager extends RecoveryManager
{
  public void recoveryAction(RepositoryManagerBehaviour storage)
    throws Exception
  {
    // 1. get failed agent's storage controller:
    AgentStorage as = storage.getAgentStorage(super.getAgentToRecover());

   // create a recovery message
    RecoveryMessage rmsg = new RecoveryMessage(as.getRegistration());

  // 2. set last checkpoint to this message
    if (as.getNumberOfCheckpoints()>0)
      rmsg.setCheckpoint(as.getAgentCheckpoint(0));

  // 3. specify message blocking for duplicated message-sent events
    AgentEventBag aeb = as.getAgentEventBag(0);
    rmsg.enableBlockSendMessages(aeb.getSentCounterTable());

  // 4. specify message replay of recorded message-posted events
    Vector playbackmsgsV = new Vector(aeb.size());
    for(int i=0; i< aeb.size(); i++)
    {
      ACLMessageEvent ae = (ACLMessageEvent)aeb.getAgentEvent(i);
      if(ae.getACLMessageEventType()==ACLMessageEvent.POSTED)
        playbackmsgsV.add(ae.getACLMessage());
    }
    ACLMessage playbackmsgs[] = new ACLMessage[playbackmsgsV.size()];
    playbackmsgsV.toArray(playbackmsgs);
    rmsg.setPlaybackMessages(playbackmsgs);

   // check available agent container
    if (super.isContainerAlive(as.getAgentContainer()))
      rmsg.dispatch(this, as.getAgentContainer());
    else
    { String[] containers = super.getBackupContainers();
      String ct = null;
      for(int i=0; i<containers.length && ct == null; i++)
        if (isContainerAlive(containers[i]))
          ct = containers[i];

    // dispatch the recovery message
      if(ct!=null)
```

```
        {
          rmsg.dispatch(this, ct, true);
        }else
        {
          System.out.println("Recover process for agent "+
                  super.getAgentToRecover()
                  +" fail since no predefined container is reachable.");
        }
      }
    }
  }
```

Figure 7-3 Source code of LogBasedRecoveryManager class

The class diagram in Figure 7-4 illustrates the design of the *protocol extension* for log-based rollback-recovery protocol.

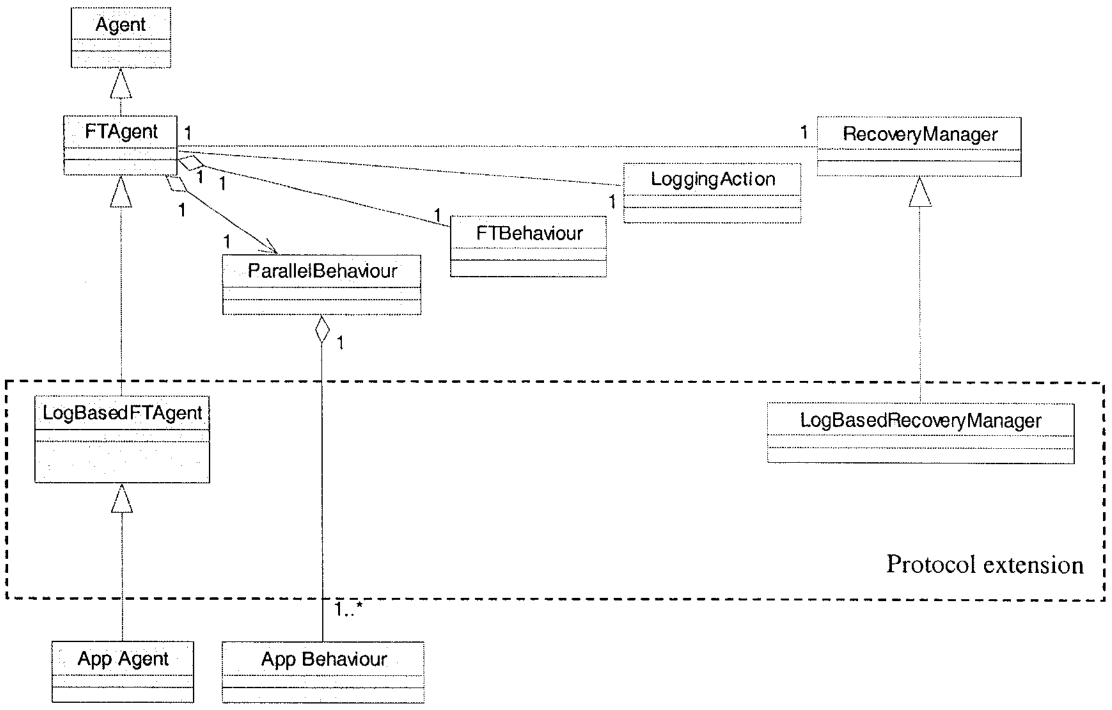


Figure 7-4 Class diagram for Log based rollback-recovery protocol extension

7.3 Case study: synchronized checkpoint protocol

Log-based rollback-recovery protocol can be applied on individual agent and doesn't involve collaboration among different agents. In this section, we introduce another protocol design that involves collaboration among a group of agents. The protocol is a synchronized checkpoint-recovery protocol based on [TamSeq84]. We made some modifications to reduce messaging cost.

7.3.1 Protocol description

This protocol scheme provides fault tolerance based on a predefined group of agents running cooperatively via message passing. We checkpoint the group at runtime synchronously and periodically so that whenever an agent failure is detected the whole group of agents are rolled back to their most recent checkpoints. Message logging in this protocol is minimal during failure free execution. During failure recovery process, no roll forward recovery control is required in this scheme. However, checkpoint coordination and recovery coordination are certainly required.

The protocol is detailed into the following actions:

1) Triggering checkpoint:

In this protocol, the agent group members are static and are known before execution. A checkpoint coordinator is elected in advance to be in charge of checkpointing coordination. A checkpoint protocol on the coordinator agent is triggered independently by either an application-alerted checkpoint event or a timer-alerted checkpoint event. These two services are already provided in FATMAD.

A checkpoint protocol on each non-coordinator agent is triggered by message-alerted checkpoint event sent by the coordinator. FATMAD provides a message type *AlertToCheckpoint* class that can be used to generate message-alerted checkpoint event by sending this type of messages.

2) Checkpoint action option:

In this protocol, each checkpoint should include the agent's message queue. *FTBehaviour* class provides a method to specify this.

3) Checkpoint collaboration action:

The checkpoint protocol implements a logical clock mechanism, in which each agent message is labelled with a logical clock. With this clocking mechanism support, a two-phase synchronous checkpointing protocol is implemented as follows: i) Whenever the coordinator starts a checkpoint protocol, it will notify all group members to be ready to checkpoint; ii) Upon receiving the checkpoint message, each non-coordinator agent acknowledges the coordinator that it is ready to checkpoint; iii) After collecting Ack messages from all group members, the coordinator sends a checkpoint message to all group members, takes a checkpoint, and resumes its execution; iv) Upon receiving this checkpoint message, each member agent takes a checkpoint including its mailbox and excluding all messages with a clock value larger than the checkpoint message's clock value. It also starts to log all messages with a clock value smaller than the checkpoint message's clock value. Then it will resume its execution.

These actions can be implemented in two methods: *preCheckpointAction()* and *postCheckpointAction()*, which will be invoked and executed atomically with checkpointing action.

4) Recovery line decision:

When a failure is detected, the failed agent as well as its group should be rolled back to their most recent checkpoints with message queue restored. Any incomplete checkpoint should be removed from storage.

We implement this action in a subclass of *RecoveryManager* class.

5) Post recovery control:

Messages coming with each checkpoint should be restored into relevant agent message queue (queue) before the agent's execution is resumed.

There are some additional actions to be performed before resuming the execution of each group member that has rolled back:

- Recovery process among the group should be synchronized.
- Message channel should be cleared.

The implementation of this action is provided in the *FTBehaviour* class. We can simply customize it via a recovery message (*RecoveryMessage* class object).

7.3.2 Design

Six classes are designed to implement the *protocol extension* for synchronized checkpoint protocol. The diagram in Figure 7-5 illustrates the class design of the *protocol extension* of the synchronized checkpoint protocol.

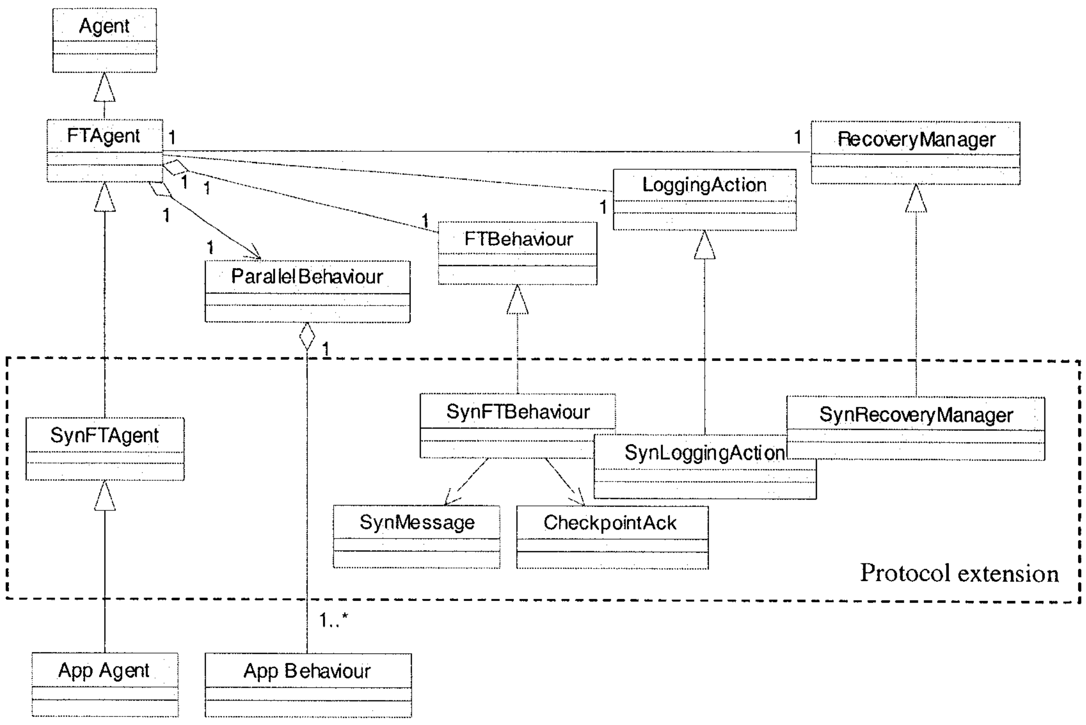


Figure 7-5 Class diagram for synchronized checkpoint protocol

SynFTAgent class

This class is designed to integrate protocol entities: a *SynFTBehaviour* object, a *SynLoggingAction* object, and a *SynRecoveryManager* object that are configured in the ***ftSetup()*** method.

Since *ftSetup()* is the initiation method, we setup the timing service with logical clock option and implement the coordinator selection in this method. If the agent is selected to be a coordinator, it is set to allow for triggering checkpoint protocol independently, otherwise it is set to be prohibited.

The source code is shown in Figure 7-6.

```
package jade.fatmad.rrft.protocol.syncheckpoint;
...
public class SynFTAgent extends FTAgent
{
  public final static String protocol = "synchornized checkpoint protocol";
  private AID coordinator;
  private boolean isCoordinator = true;
  private boolean protocol_state = false;
  public boolean isCoordinator () { return isCoordinator;}
  public AID getCoordinator(){ return coordinator; }

  public void ftSetup()
  {
    // set up protocol entities
    setFTBehaviour(new SynFTBehaviour(this));
    setLoggingAction(new SynLoggingAction());
    setEventReaction(new SynRecoveryManager());

    // enble timing service with Logical clock option
    super.enableTimingService(super.CLOCK_ON_POST, new LogicalClock());

    // elect a coordinator among group members
    AgentGroup g = getAgentGroup();
    if (g!=null&&g.size()>1)
    {
      AID memberlist[] = g.getGroupMemberList();
      if (memberlist[0].equals(getAID()))
      { isCoordinator = true;
      }
      else
      { coordinator= memberlist[0];
        isCoordinator = false;
      }
    }

    // Only allow coordinator to initiate checkpoint protocol independently
   if (!isCoordinator)
   {  super.disableFlagToCheckpoint();
      super.disableFrequentCheckpoint();
   }
    else
   {  super.enableFlagToCheckpoint();
      super.enableFrequentCheckpoint();
   }
  }
}
```

Figure 7-6 SynFTAgent class source code

SynFTBehaviour class

This class is designed to implement checkpointing control, especially checkpoint coordination for our synchronized checkpoint protocol. The two-phase checkpoint coordination protocol is mainly implemented in the *preCheckpointAction(...)* method since this method will be invoked before checkpoint is taken. In other words, once the coordination is done the checkpoint will be immediately taken.

We setup the checkpoint option in the class constructor: *SynFTBehaviour(...)*, in which messages in each agent's message queue (message queue) are selectively recorded when taking a checkpoint. The selection criteria is evaluated by the Boolean method:

public boolean takeWithCheckpoint(ACLMessage m).

We override this method for non-coordinator agents to filter out messages with clock value bigger than the checkpoint message clock value.

The source code is shown in Figure 7-7.

```
package jade.fatmad.rrft.protocol.syncheckpoint;
...
public class SynFTBehaviour extends FTBehaviour
{
  LogicalClock synchrony;
  public SynFTBehaviour(SynFTAgent owner)
  {
    super(owner);
    // set the option that when taking a checkpoint messages
    // in message queue will be selectively recorded.
    super.setTakeMsgqueueWithCheckpoint (
        FTBehaviour.CHECKPOINT_WITH_SELECTED_MESSAGES);
  }

  public void preCheckpointAction(LoggingAction la)
  { // the agent is triggered to take checkpoint by a checkpoint event

    if (!((SynFTAgent)myAgent).isCoordinator())
    { synchrony = null;
      ((SynLoggingAction)la).reset();
    }
    if(((SynFTAgent)myAgent).isCoordinator())
    { // send alert-to-checkpoint message to group members
      AlertToCheckpoint m = new AlertToCheckpoint(myAgent.getAID());
      AgentGroup g = ((SynFTAgent)myAgent).getAgentGroup();
      AID mlist[]=g.getGroupMemberList();
      for(int i=1; i<mlist.length; i++)
        sendSysMessage(m, mlist[i],SynFTAgent.protocol);

      // waiting for Ack messages from each group memeber
      for(int j=1; j<mlist.length;j++)
        blockingReceiveSysMessage("CheckpointACK");

      // send synchorny message to group memebers to commit checkpoints
      SynMessage sm = new SynMessage();
      for(int i=1; i<mlist.length; i++)
        sendSysMessage(sm, mlist[i],"Checkpoint commit");
    }else
    {
      // send Ack message to the coordinator
      SynCheckpointAck m = new SynCheckpointAck();
      sendSysMessage(m, ((SynFTAgent)myAgent).getCoordinator(),
                        "CheckpointACK");

      // waiting for checkpoint synchrnonization message(commit)
      blockingReceiveSysMessage("Checkpoint commit");
      synchrony = (LogicalClock)getClock();
    }
  }

  public void postCheckponitAction(LoggingAction la)
  { // non-coordinators need to log messages with clock value smaller than
    // that of the synchrony message.
    if (!((SynFTAgent)myAgent).isCoordinator())
    { ((SynLoggingAction)la).stopLoggingAfterClock(super.getClock());
      la.startLogAllPosted(null);
    }
  }

  public boolean takeWithCheckpoint(ACLMessage msg)
  { if (!((SynFTAgent)myAgent).isCoordinator())
    { try {
        return synchrony.compareTo(msg.getClock())>0;
      } catch (Exception e) {   e.printStackTrace();   }
    }
    return true;
  }
}
```

Figure 7-7 SynFTBehaviour class source code

SynLoggingAction class

This class is designed to support message logging in a checkpoint protocol, in which messages with clock value smaller than the clock value of the checkpoint message should be recorded. Three methods are implemented: *stopLoggingAfterClock(...)* method is designed to set the logging policy, *logPosedPredicate(...)* method is designed to return a Boolean result that will be used to decide if a message is to be logged, and *reset()* method is designed to initiate the setting when a checkpoint protocol starts.

The source code is shown in Figure 7-8.

```
package jade.fatmad.rrft.protocol.syncheckpoint;
...
public class SynLoggingAction extends LoggingAction
{
  LogicalClock clock;

  public void stopLoggingAfterClock(Clock clock2)
  {
    if (clock2 instanceof LogicalClock)
    clock = (LogicalClock)clock2;
  }

  public void reset(){ clock = null;      }

  public boolean logPostedPredicate(ACLMessage msg)
  {
    if (clock!=null && clock instanceof LogicalClock)
    {
      try {
        return msg.getClock().compareTo(clock)<0;
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    return false;
  }
}
```

Figure 7-8 SynLoggingAction class source code

SynRecoveryManager class

This class is designed to implement the recovery process of the protocol, in which a triggered recovery manager object needs to decide recovery line and dispatch recovery task for each agent in the group. We override the method: *recoveryAction()*, in which we implemented the following:

1) Check the completeness of the last available checkpoint group. If the last checkpoint group is not complete, then remove it from the repository. The last complete checkpoint group can be used to form a recovery line.
2) Create a recovery message (*RecoveryMessage* class object) for each agent, configure the message to specify recovery options, and dispatch them.

   In each recovery message, we specify the following:

   - The checkpoint that will be used for recovery;
   - Channel messages that will be restored;
   - The agent container where the agent will be recovered;
   - Enable synchronization process so that all group members are synchronized before resuming their execution;
   - Enable the agent group to flush message channels after recovery.

The source code is shown in Figure 7-9.

```
package jade.fatmad.rrft.protocol.syncheckpoint;

public class SynRecoveryManager extends RecoveryManager
{
 private int nSeqNo;

 public void recoveryAction(RepositoryManagerBehaviour rm) throws Exception
 {
  // get the storage info of the failed agent
  AgentStorage a_storage = rm.getAgentStorage(super.getAgentToRecover());

  //Get group IDs
   AID[] a_list= a_storage.getAgentGroup().getGroupMemberList();

   // physical checkpoint sequence # of the failed agent
   nSeqNo=a_storage.getNumberOfCheckpoints();

   //Determine a valid recovery line
    //Determine the min seq # that everybody should get back
   if (a_list.length > 1)
   {
     for (int i = 0; i < a_list.length; i ++)
     {
       a_storage=rm.getAgentStorage(a_list[i]);
       a_storage.suspend();
       if (nSeqNo > a_storage.getNumberOfCheckpoints())
       {
         nSeqNo=a_storage.getNumberOfCheckpoints();
         System.out.println("NEW MIN CHK-PNT # : " + nSeqNo);
       }
     }

     //Delete incomplete checkpoint group if exists
     for (int i = 0; i < a_list.length; i ++)
     {
       a_storage=rm.getAgentStorage(a_list[i]);
         if (nSeqNo < a_storage.getNumberOfCheckpoints())
        a_storage.removeLastCheckpointWithBag() ;
     }
   }
  //end of recovery line determination

 // Cerate recovery meassages for all the agents
   RecoveryMessage rMsg ;
   AgentCheckpoint  last_Chkpnt ;
   AgentEventBag last_Eventbag;
   Vector playback_MsgsV ;
   ACLMessageEvent msg_Event;

  RecoveryMessage[] rmg = new RecoveryMessage[a_list.length];

 // Customize each recovery message
 for (int i = 0; i < a_list.length; i ++)
   {
     // set checkpoint
     a_storage=rm.getAgentStorage(a_list[i]);
     rmg[i] = new RecoveryMessage(a_storage.getRegistration());
     last_Chkpnt = a_storage.getAgentCheckpoint(0);
     rmg[i].setCheckpoint(last_Chkpnt);

     // set channel messages
     last_Eventbag= a_storage.getAgentEventBag(0);
     playback_MsgsV=new Vector();
     for(int j=0; j< last_Eventbag.size(); j++)
     {
       msg_Event = (ACLMessageEvent)last_Eventbag.getAgentEvent(j);
       if(msg_Event.getACLMessageEventType()==ACLMessageEvent.POSTED
        ||msg_Event.getACLMessageEventType()==ACLMessageEvent.INQUEUE)
        playback_MsgsV.add(msg_Event.getACLMessage());

     }
     ACLMessage playback_Msgs[] = new ACLMessage[playback_MsgsV.size()];
     playback_MsgsV.toArray(playback_Msgs);
     rmg[i].setPlaybackMessages(playback_Msgs);

    //  enable flushing message channels all channels
     rmg[i].enableFlushChannels(a_list);
```

```
      // request each agent to be synchronized during post-recovery
       rmg[i].enableRecoverySynchronization(this.getAID());

      // specify agent container for the agent
      if (super.isContainerAlive(a_storage.getAgentContainer()))
        rmg[i].setAgentContainer(a_storage.getAgentContainer());
      else
      { String[] containers = super.getBackupContainers();
        String ct = null;
        for(int i=0; i<containers.length && ct == null; i++)
          if (isContainerAlive(containers[i]))
            ct = containers[i];
         if(ct!=null)
          rmg[i].setAgentContainer(ct.getContainerName());
         else
         {
          System.out.println("No predefined recovery container alive. Recovery action
aborted");
           return;
         }
       }
    }

    // dispatch recovery messages
    for (int i=0; i<=rmg.length; i++)  rmg[i].dispatch(this);

    // set the this agent to be a synchronizer for the recovery synchronization
    super.enableRecoverySynchrony(a_list);
  }
}
```

Figure 7-9 SynRecoveryManager class source code

CheckpointAck class and *SynMessage* class are designed to define checkpoint Ack message and checkpoint message that are used in checkpoint collaboration in *SynFTBehaviour* class (see Figure 7-7).

Chapter 8 Fault tolerant application development and deployment

Although FATMAD provides comprehensive support to enhance fault tolerance protocol development, its ultimate objective is to achieve application level fault tolerance. In this chapter we introduce how to use FATMAD to develop a fault tolerant agent application and deploy it.

Unlike protocol designers who need to understand *protocol skeleton* in order to develop a protocol, application developers are required to know very little about internal structure of FATMAD. From application developer's perspective, FATMAD is a fault tolerance extension to Jade agent platform. It consists of a fault tolerance protocol library, which can help us build up fault-tolerant agents, and a set of Runtime components, by which we can deploy fault tolerant agents.

8.1 Developing fault tolerant agents

8.1.1 Approach

Developing a fault-tolerant application with FATMAD does not require an application developer to do fault tolerance design from scratch. Each protocol provided in FATMAD allows an agent application to be easily transformed into a set of fault tolerant agents. The only development effort towards fault tolerance for an application developer is to do the required transformation.

$$\text{Application agents} \xrightarrow{\text{Transformation}} \text{FT application agents}$$

Figure 8-1 Transformation

Suppose an application is well developed and tested, the application developer needs to do the following in order to achieve fault tolerance:

1) **Protocol selection**

FATMAD has a protocol (extension) library that archives a set of rollback-recovery protocols. The application developer needs to select a protocol that is deemed to be suitable for his/her application, based on the characteristics of the agent application and the characteristics of the protocol.

2) **Application transformation**

Once a protocol is selected, the application developer needs to perform an application transformation, i.e. integrate the selected protocol into his/her application by applying the API of the selected *protocol extension*.

In order to properly use a FATMAD supported protocol, the developer needs to understand the following: i) important characteristics of the selected protocol, including the protocol scheme, suitability for specific application types, benefits, etc, and ii) the programming interface for the selected protocol.

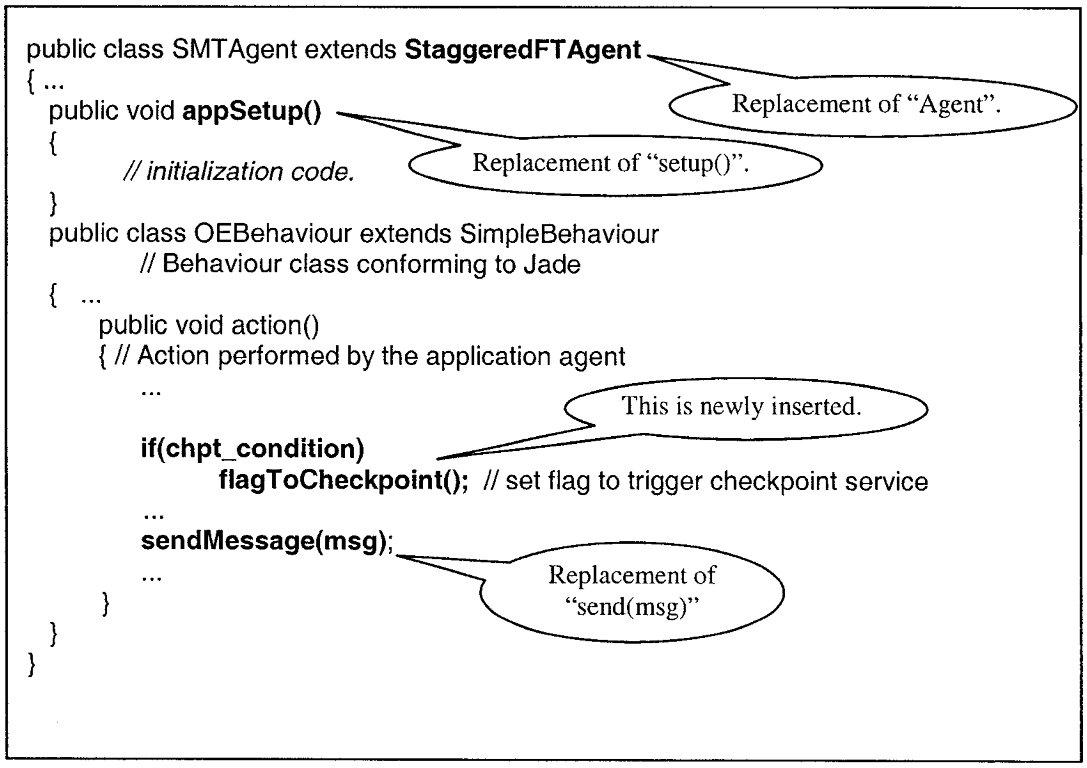
8.1.2 Framework protocol API

FATMAD has simple APIs for application developers. This makes application transformation work easy. Although different protocols have different characteristics, APIs of different protocols are very similar from the programming perspective.

Usually to transform a normal application agent into a fault-tolerant agent, a developer needs to do the following:

- **Extend protocol specific agent class**: Programming in Jade involves creating an application agent class by extending the Jade agent class. Similarly, to create a FATMAD supported fault-tolerant application agent, one needs to extend a protocol-specific FATMAD agent class (which is in fact a sub-class of the Jade agent class).
- **Replace a set of agent class methods**: In Jade agent programming, users need to override methods, such as *setup()* and *takedown()*, which perform user-defined application-specific initialization/clean-up operations. In using FATMAD, these methods are replaced by *appSetup()* and *appTakeDown()*.

- **Replace messaging function calls**: In Jade, a developer needs to use methods such as *send(...)* and *receive(...)*, to do messaging. In FATMAD, these methods are replaced by *sendMessage(...)* and *receiveMessage(...)*, which augment Jade messaging services with system controlled and user-transparent message handling services.
- **Flag to trigger checkpoint/logging**: Some protocols require the application to set flags at desired points in order to trigger a checkpoint. Application designers should program accordingly by calling some pre-defined methods provided either by the kernel or by the protocol.



```
public class SMTAgent extends StaggeredFTAgent
{ ...
  public void appSetup()
  {
        // initialization code.
  }
  public class OEBehaviour extends SimpleBehaviour
        // Behaviour class conforming to Jade
  { ...
      public void action()
      { // Action performed by the application agent
          ...
          if(chpt_condition)
                flagToCheckpoint();  // set flag to trigger checkpoint service
          ...
          sendMessage(msg);
          ...
      }
  }
}
```

Figure 8-2 Transformed code example

Figure 8-2 shows code fragments of an agent application named *SMTAgent,* which employs the staggered checkpointing protocol [Vaidya99] by extending the protocol-specific class named *StaggeredFTAgent.* It also demonstrates how the code is modified for the transformation. Bold words in the code are the modified parts, e.g., the calling of the *flagToCheckpoint()* method, as is required by the protocol.

8.2 Deploying a fault-tolerant application

Once an application is transformed and complied, we are ready to deploy it. Deploying *FT application agents* requires a runtime environment providing system services to support embedded protocol entities to be deployed and be active. The required runtime environment should be composed of Jade (modified) platform and *FATMAD runtime*. Therefore, a complete deployment process involves the following three steps:

1) Start the modified version of Jade platform

As we discussed before, during the development of FATMAD we modified the source code of Jade to implement some services that cannot be implemented directly on top of original Jade framework. Therefore, we need to run the modified version of Jade platform.

2) Start *FATMAD runtime*

A *FATMAD runtime* consists of three types of service agents: *container proxy agent*, *FD monitor agent*, and *repository manager agent*.

Since *container proxy agents* are automatically loaded to each agent node (container) during node start-up, we only need to consider loading *FD monitor agents* and repository managers. Two factors should be considered for setting up the two types of service agents: quantity and locality of each type of service agent that should be loaded.

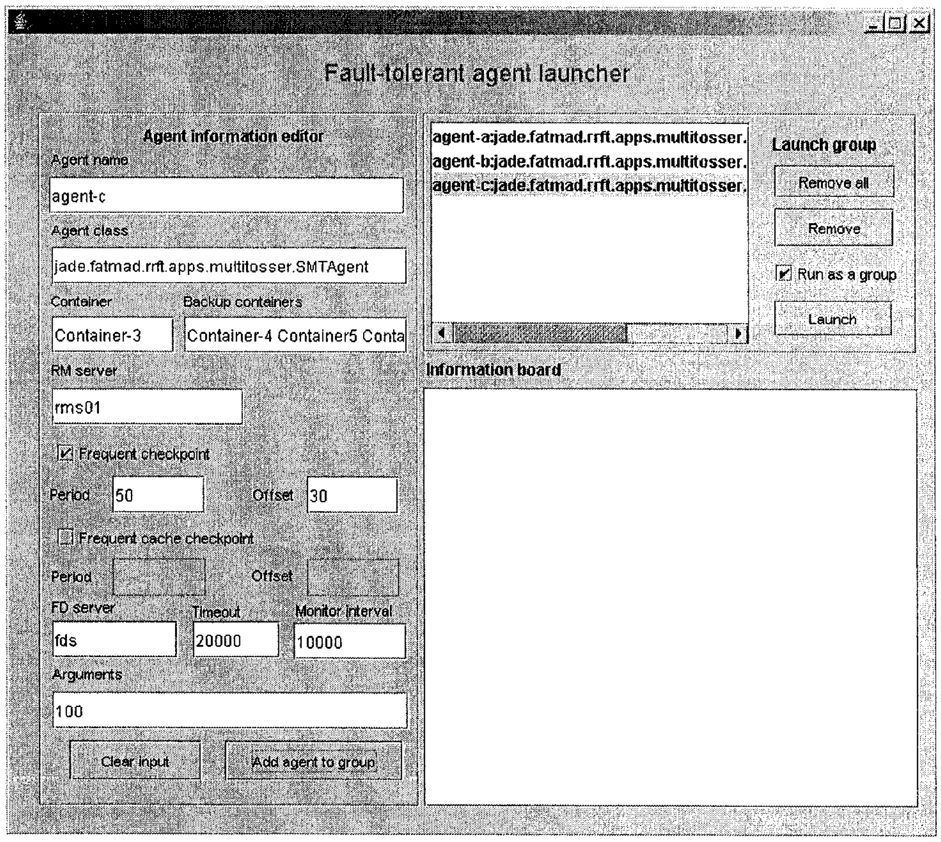
3) Launch *FT application agents*

Since some parameters are required to be specified in deployment phase, we developed a deployment tool, using which those parameters can be specified before loading agents into Jade platform.

The following are the deploying parameters that are required to load *FT application agents*:

- Agent name: the ID of the agent to be loaded;
- Agent class: the Java class that defines the agent to be loaded;

- Container: the agent container that will accommodate the agent to be loaded;
- Backup containers: the backup agent containers that could be used during recovery, especially when a container crash occurs;
- RM(repository manager) server: the repository agent that is selected to serve the agent to be loaded;
- Periodic checkpointing option: whether periodic checkpointing should be applied; and the frequency parameters;
- FD (failure detector) server: the *FD monitor agent* that is selected to serve the agent; and the monitor parameters;
- Arguments: the application arguments for the agent to be loaded;
- Group information: which set of agents should be launched as a group.



Agent name: agent-c
Agent class: jade.fatmad.rrft.apps.multitosser.SMTAgent
Container: Container-3
Backup containers: Container-4 Container5 Conta
RM server: rms01
[✓] Frequent checkpoint
Period: 50 Offset: 30
[ ] Frequent cache checkpoint
Period: Offset:
FD server: fds Timeout: 20000 Monitor interval: 10000
Arguments: 100

agent-a:jade.fatmad.rrft.apps.multitosser.
agent-b:jade.fatmad.rrft.apps.multitosser.
agent-c:jade.fatmad.rrft.apps.multitosser.

[✓] Run as a group

Figure 8-3 FT agent deployment tool: FT Agent Launcher

Figure 8-3 shows the interface of our deployment tool: Fault-tolerant agent launcher.

Chapter 9 Evaluation

In this chapter, we detail our evaluation of FATMAD.

9.1 Usability evaluation

In order to evaluate the usability of FATMAD, we invited three students to develop a rollback-recovery protocol as well as a test agent application using FATMAD. They built up a *protocol extension* based on the staggered protocol [Vaidya99], and a test application to test the protocol. By reviewing the test results, we verified that the protocol and the application were correctly implemented.

By talking to the developers, we received the following feed back:

- FATMAD provides many useful services and greatly reduced their development work:
    - Many services have been provided, such as checkpointing, data service, message intercepting service, monitor service, failure triggering service, etc.
    - Some protocol-specific behaviour can be implemented by customizing existing services in FATMAD, such as message logging and recovery execution, which are very handy and easy to apply.
- Using FATMAD, developers can focus on protocol design instead of spending time in building some fundamental services. The implemented *protocol extension* classes are more like design specification, in which they specify protocol behaviours and decisions. This leads to a much-reduced implementation with only six small classes, which are easy to examine and debug.
- Using FATMAD, it's easy to transform a normal application into a fault tolerant one. It is also easy to replace a protocol with another one on the same application.

- The deployment of *FATMAD runtime* and fault-tolerant agents is quite flexible. Developers can customize the deployment plan to satisfy their performance requirement.

9.2 Performance evaluation

Performance is another factor in evaluating FATMAD. By evaluating performance, we can deduce how much overhead is incurred to add a fault tolerance feature using FATMAD.

During error free execution of an *FT application agent*, the checkpoint/logging could impose additional overhead on an application agent during the following functions:

- Event logging
  - Message events are intercepted and logged into stable place
- Checkpointing
  - Agent object serialization and checkpoint delivery
- Protocol coordination
  - This is protocol dependent
- Notifying monitor
  - Notify the monitor agent of "alive" messages
- Protocol initialization
  - Registering to relevant *FATMAD runtime* services

However, these overhead factors do not fully affect an agent's performance if the agent is idle. For example, taking a checkpoint when an agent is waiting for a message may not incur visible overhead.

The previous overhead can be further refined and categorized as the following:

- Object serialization
- Data transmission
- Data storing

- Synchronization

Certainly computer speed, memory size, hard drive speed, I/O throughput, and network speed are important hardware factors that affect FATMAD performance.

Performance test

In order to implement a meaningful performance test, we designed a test application to evaluate FATMAD overhead. The test application performs frequent computational job combined with messaging actions and a log-based protocol. As a result, most of the above overhead factors could be involved, except for protocol coordination, which is protocol dependent.

The test application ran in a LAN environment with a group of machines running Windows2000/XP operating systems. The following are the configurations:

PC1:

CPU: Athlon 800MHz

Memory: 512 MB

Hard drive: 30G 5400 RPM

Operating system: Windows 2000

Software: Sun JDK 1.4.2, Modified Jade 3.01b plus FATMAD

PC2:

CPU: Celeron 2.4GHz

Memory: 256 MB

Hard drive: 40G 7200RPM

Operating system: Windows XP

Software: Sun JDK 1.4.2, Modified Jade 3.01b plus FATMAD

PC3:

CPU: Pentimum4 2.6GHz

Memory: 1GB

Hard drive: 80G 7200RPM

Operating system: Windows XP

Software: Sun JDK 1.4.2, Modified Jade 3.01b plus FATMAD

Network: 10 Base T Eithernet

We arranged the application agents to run on two machines, and FATMAD components to run on a separated machine.

By providing different parameters, we wanted to see how checkpointing and logging impose visible overheads to overall execution of an agent application. The parameters include event logging frequency, checkpoint frequency and checkpoint size.

Table 9-1 illustrates four groups of testing results categorized by the different parameters. All the four groups take minimum checkpoint, which is one, and have different message event frequencies. For each group, we first ran the application without applying fault tolerant protocol, and then we ran the same application embedded with log-based protocol.

In group 1, each *FT application agent* takes 1 initial checkpoint and logs 1.088 event(s) per second. The average overhead is 2.9% of the additional computation time.

In group 2, 3, and 4, we adjust the parameters so that message event frequencies become 2.471 events/second, 4.365 events/second, and 9.561 events/second respectively. The overheads incurred on relevant *FT application agent* are increased to 6.05%, 7.40%, and 19.29% respectively. The diagram in Figure 9-1 illustrates that the more frequently the events are logged the more overhead is generated.

Under such a runtime environment, if a fault tolerant protocol embedded in an application agent incurs event logging action below 5 events per second, then the overhead should be acceptable in general. Here we assume that the general message size is less than 20 KB. Certainly the larger the size of the message is, the more overhead the logging incurs.

Since a checkpoint-based protocol usually requires much less messages to be logged than a log-based protocol, we predict that a checkpoint based protocol will be more scalable than a log-based protocol in terms of number of message events occurring at each agent.

	events /process	average execution time (s, Without FT)	average event frequency (events/s)	average execution time (s, with FT)	average overhead
Group 1	50	45.946	1.088	47.278	2.90%
Group 2	100	40.463	2.471	42.911	6.05%
Group 3	300	68.723	4.365	73.811	7.40%
Group 4	500	52.295	9.561	62.385	19.29%

Table 9-1

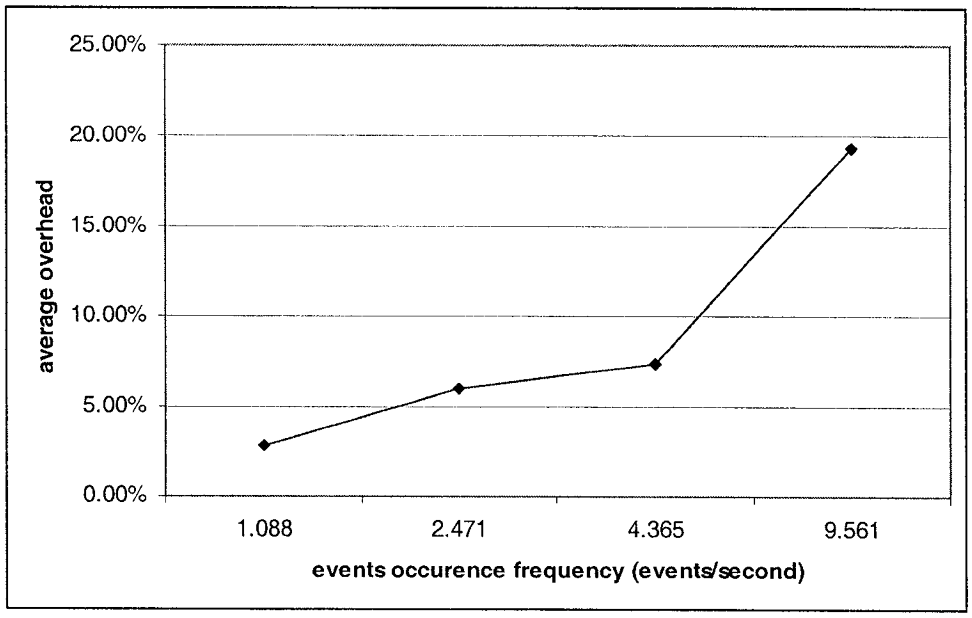


Figure 9-1 FATMAD performance chart - I

Table 9-2, 9-3, and 9-4 illustrate test cases focusing on how checkpointing imposes overhead on application agents. In the collection of test cases shown in table 9-2, we fixed the event frequency to 1.088 events per second (same as group 1 in above test case), and then adjusted checkpointing frequency to observe the average overhead. Obviously, overhead is directly proportional to the frequency of checkpointing. The test cases shown in table 9-3, and 9-4 did the same test except that the checkpoint sizes are different in these test cases, i.e., 12 KB, 108KB, and 1MB.

checkpoints /process	average execution time (s)	average checkpointing period (s)	average overhead
1	47.278	47.278	2.90%
3	47.335	15.778	3.02%
6	47.595	7.933	3.59%
9	47.908	5.323	4.27%
19	49.571	2.609	7.89%

Table 9-2 Test cases with average event interval = 1.088 events/s and checkpoint size = 12 KB

checkpoints /process	average execution time (s)	average checkpointing period (s)	average overhead
1	47.773	47.77	3.98%
3	48.294	16.10	5.11%
5	48.421	9.68	5.39%
8	48.61	6.08	5.80%
22	51.573	2.34	12.25%

Table 9-3 Test cases with average event interval = 1.088 events/s and checkpoint size = 108 KB

checkpoints /process	average execution time (s)	average checkpointing period (s)	average overhead
1	53.427	53.43	16.28%
3	57.137	19.05	24.36%
5	59.375	11.88	29.23%
7	62.254	8.89	35.49%
15	78.978	5.27	71.89%

Table 9-4 Test cases with average event interval = 1.088 events/s and checkpoint size = 1013 KB

The diagram in Figure 9-2 illustrates how checkpoint size and checkpoint frequency affect average overhead on an application agent.

From the test results we can conclude:

- Overhead is directly proportional to checkpoint frequency.
- The more frequent it takes checkpoints the more overhead generated
- The bigger the checkpoint size the more overhead is generated. As we observed, most agents have a checkpoint size of less than 50 KB.
- With a small size of agent checkpoint, increasing checkpoint frequency leads to slower performance degradation than larger sizes of agent checkpoints.
- Since this application is a computationally oriented, we predict that non-computationally oriented applications would have smaller performance degradation.

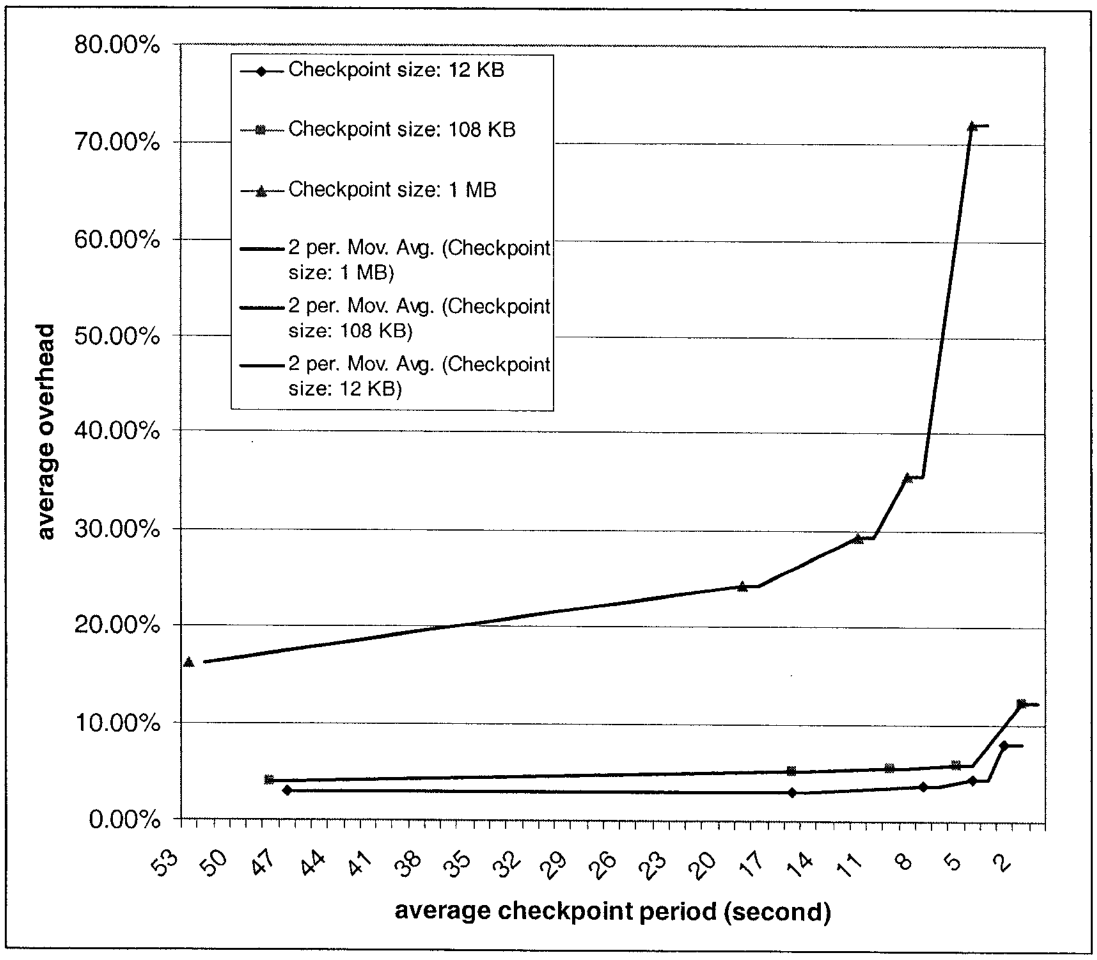


Figure 9-2 FATMAD performance chart - II

Chapter 10 Conclusions

In this thesis, we have presented a fault-tolerant multi-agent development framework (FATMAD). Based on checkpoint/recovery techniques, FATMAD is aimed at providing application level fault tolerance development support to both application developers and fault tolerance protocol developers. Using FATMAD, application developers can easily build up fault tolerant applications with minimum implementation efforts and minimum knowledge on fault tolerance design except for necessary APIs. FATMAD provides a protocol library that allows application developers to select a protocol suitable to their applications. Using FATMAD, protocol designers can design new protocols so that FATMAD can be enriched. FATMAD provides many useful services that make protocol developers' work easy. In addition, FATMAD provides flexibility on deployment of *FATMAD runtime* to allow application developers to easily adapt to their needs.

However, there are still many valuable features that should be further developed to enhance FATMAD. We list them in the following:

- FATMAD API should be further enriched to support various protocols;
- FATMAD needs a rich protocol set and hence more protocols should be developed;
- FATMAD itself should be designed to be fault tolerant so that it can handle failures that cause FATMAD service crash.
- Failure detection that goes beyond crash failure, such as safety failure, should be included;
- Protocol test-bed issue needs to be explored further;
- More usability experiments need to be conducted to improve the ease of use of FATMAD.

Bibliography

[Adina98] Adina Magda Florea. Introduction to Multi-Agent Systems. International Summer School on Multi-Agent Systems, Bucharest, 1998

[Agentcities] Agentcities, http://www.agentcities.net/

[Agentlink] Agent Link http://www.agentlink.org/

[AGJ04] M. M. Akon, D. Goswami, and S. Jyoti. Routing in Telecommunication Network with Controlled Ant Population, in Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, Nevada, USA, January 2004.

[AGKSW01] Josef Altmann, Franz Gruber, Ludwig Klug, Wolfgang Stockner, and Edgar Weippl. Using Mobile Agents in Real World: A Survey and Evaluation of Agent Platforms. in Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the 5th International Conference on Autonomous Agents, Montreal, Canada, May 28-31, 2001, ACM Press 2001.

[AlMa98] L. Alvisi, and K. Marzullo. Message logging: pessimistic, optimistic, causal and optimal, IEEE Trans. Software Eng. 24(2) (1998) 149-159.

[ArLa98] Y. Aridor, and D.B. Lange. Agent design patterns: elements of agent application design, in Proceedings of the Agents'98, Minneapolis, Minnesota, May 1998, pp. 108-115.

[BePR99] F. Bellifemine, A. Poggi, and G. Rimassa. JADE --- A FIPA-Compliant Agent Framework, in Proceedings of the PAAM'99, London, UK, 1999. The Practical Application Company Ltd, pp. 97-108.
(See http://sharon.cselt.it/projects/jade/ for latest information)

[BMO01] B. Bauer, J.P. Müller, and J. Odell. Agent UML: a formalism for specifying multiagent interaction, in: P. Ciancarini, M. Wooldridge, (Eds.), Proc. Agent-Oriented Software Engineering, May 2001, Springer-Verlag, Berlin, 2001, pp. 91-103.

[CaSi98] G. Cao, and M. Singhal. On coordinated checkpointing in distributed systems, IEEE Trans. Parallel and Distributed Systems, 9(12) (1998) 1213-1225.

[ChanLamp85] M. Chandy, and L. Lamport. Distributed snapshots: determining global states of distributed systems, ACM Trans. Computing Systems, 3(1) (1985) 63-75.

[CM-MAS] Carnegie Mellon University. Multi-Agent Systems
http://www-2.cs.cmu.edu/~softagents/multi.html

[CoLe98] J.C. Collis, and L.C. Lee. Building electronic market-places with the ZEUS tool-kit, in Proceedings of the AMET-98 workshop, Autonomous Agents'98, pp. 17-32.

[CriJah91] F. Cristian, and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In Proceedings, Tehth Symposium on Reliable Distributed systems, 12-20, 1991.

[DeLoach99] Scott A. DeLoach Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems. In Proceedings of Agent Oriented Information Systems, pages 45–57, 1999.

[DiWo03] Ian Dickinson, and Michael Wooldridge. Web technologies: Towards practical reasoning agents for the semantic web, in Proceedings of the second international joint conference on Autonomous agents and multiagent systems table of contents, Melbourne, Australia, Pages: 827 - 834, 2003.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys, Vol.34, No.3 September 2002, pp. 375-408.

[Elnozahy93]	E.N. Elnozahy Manetho: Fault Tolerance in Distributed Systems using Rollback-Recovery and Process Replication, Ph. D. Thesis, Rice University, Department of Computer Science, 1993.
[ElnZwa92]	E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing, In Proceedings, Eleventh Symposium on Reliable Distributed systems, 39-47, 1992.
[FiLa97]	T. Finin, and Y. Labrou. KQML as an agent communication language, in Software Agents. Bradshaw, J.M.(ED.), MIT Press, Cambridge, MA, 1997.
[FIPA]	Agent Communication Language, FIPA 97 Specification, http://www.fipa.org/.
[GiMP02]	Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1 table of contents, Bologna, Italy, Pages: 35 - 36, 2002.
[GOF94]	Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software
[GuSc96]	R. Guerraoui, and A. Schiper. Fault-tolerance by replication in distributed systems, in Reliable Software Technologies - Ada-Europe'96, LNCS 1088, pp. 38-57. Springer-Verlag, June 1996.
[HaMu01]	M. Hannebauer, and S. Muller. Distributed constraint optimization for medical appointment scheduling, in Proceedings of the Agents 2001, Montreal, Quebec, Canada, May 2001, pp. 139-140.
[Jade]	Jade Agent Development Framework, http://jade.tilab.com/
[Jalote94]	P. Jalote. Fault Tolerance in Distributed Systems. PTR Prentice Hall
[JohFoo88]	R. Johnson and B. Foote, Designing reusable classes, Journal of Object-Oriented Programming, vol. 1, pp. 22-35, June 1988.
[Johnson97]	Ralph E. Johnson Frameworks = (Components + Patterns), Communications of The ACM, October 1997/ Vol. 40, No.10
[JSMM03]	Thomas Juan, Leon Sterling, Maurizio Martelli, and Viviana Mascardi. Software engineering: Customizing AOSE methodologies by reusing AOSE features, Proceedings of the second international joint conference on Autonomous agents and multiagent systems table of contents, Melbourne, Australia, Pages: 113 - 120, 2003.
[JuPS02]	Thomas Juan, Adrian Pearce, and Leon Sterling. ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1 table of contents, Bologna, Italy, Pages: 3 - 10, 2002.
[Kendall00]	E.A. Kendall. Agent software engineering with role modeling, in: P. Ciancarini, M. Wooldridge, (Eds.), in: Proc. the First International Workshop (AOSE-2000), Springer-Verlag, Berlin, Germany, Jan. 2000, pp. 163-170.
[Kendall98]	E.A. Kendall. Agent roles and aspects, in: S. Demeyer, J. Bosch, (Eds.), Proc. ECOOP Workshops, Springer-Verlag, LNCS 1543 (1998) 440.
[KiGR96]	D. Kinny, M. Georgeff, and A. Rao. A Methodology and Modeling Technique for Systems of BDI Agents, In W. Van de Velde and J.W. Perram, editors, Agents Breaking Away: Proceedings of the 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (LNAI 1038), pp 56-71. Springer Verlag, 1996.
[KMMK+03]	Salim Khan, Ravi Makkena, Foster McGeary, Keith Decker, William Gillis, and Carl Schmidt. Simulation: A multi-agent system for the quantitative simulation of biological networks, in Proceedings of the second international joint conference on Autonomous agents and multiagent systems table of contents, Melbourne, Australia, Pages: 385 - 392, 2003
[KoTo87]	R. Koo, and S. Toueg. Checkpointing and rollback recovery for distributed systems, IEEE Trans. Soft. Eng., 13(1) (1987) 23-31.

[KSTV+01] Yiannis Kouroupis, Yiannis Stavroulas, Katerina Tsiara, Theodora Varvarigou, Aarno Lehtola, Kuldar Taveter, Victor A. Villagra, Jorge E. Lopez de Vergara, and Christophe Duhem. Technology Review and Selection. Project Deliverable D51 in the project of Multilingual Knowledge Based European Electronic Marketplace.

[LaiYang87] T.H. Lai, and T. H. Yang. On distributed snapshots, Information Processing Letters 25, 153-158,

[Lamport78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System, Communication of the ACM 21(7), 558-565, 1978.

[Lind00] J¨urgen Lind Issues in Agent-Oriented Software Engineering The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000.

[Liu01] J. Liu. Autonomous Agents and Multi-Agent Systems: An Introduction, World Scientific, Singapore, 2001.

[LiWG04] H.F. Li, Z. Wei, and D. Goswami. Quasi-atomic recovery for distributed agents, under journal revision.

[MaDN02] Philippe Massonet, Yves Deville, and Cédric Nève. From AOSE methodology to agent implementation, In proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1 table of contents, Bologna, Italy, Pages: 27 - 34, 2002.

[MaFu01] Yasuo Matsumoto, and Satoru Fujita An Auction Agent for Bidding on Combinations of Items in Proceedings of the fifth international conference on Autonomous agents table of contents, Montreal, Quebec, Canada, Pages: 552 - 559, 2001.

[ManSing99] D. Manivannan, and Mukesh Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. IEEE Transactions on Parallel and Distributed Systems. 10-7 July 1999.

[MBBC+98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF The OMG Mobile Agent System Interoperability Facility, Proceedings of the International Workshop on Mobile Agents (MA'98), Stuttgart, September 1998. It also appeared as Personal Technologies, Springer Verlag, (1998), 2:117-129.

[MiGa01] N. Mittal, and V.K. Garg. On Detecting Global Predicates in Distributed Computations, in: Proc. IEEE ICDCS, Phoenix, May 2001, pp. 3 - 10.

[MoIs02] Antonio Moreno, and David Isern A First Step Towards Providing Health-Care Agent-Based Services to Mobile Users in Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, Bologna, Italy. Pages: 589 - 590, 2002.

[NeiTou87] G. Neiger, and S. Toueg. Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems(preliminary version), Proceedings of the Sixth ACM Annual Symposium on Principles of Distributed Computing, Schneider, F.B., ed., Vancouver, BC, Canada, 281-293, 1987.

[OdPB00] J. Odell, H.V.D. Paranak, and B. Bauer. Extending UML for agents, in Proc. AOIS Workshop at AAAI 2000, Mar. 2000, Austin, TX, USA, pp. 3-17.

[OMG-MASIF] OMG MASIF, OMG TC Document ORBOS/97-10-05.

[PaWi02] Lin Padgham, and Michael Winikoff. Prometheus: a methodology for developing intelligent agents, In proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1 table of contents, Bologna, Italy, Pages: 37 - 38, 2002.

[PiGa00] A. Pivk, and M. Gams. Intelligent Agents in E-Commerce. Electrotechnical Review, 67(5)(2000) 251-260.

[PPG00] H. Pals, S. Petri, and C. Grewe. FANTOMAS: Fault Tolerance for Mobile Agents in Clusters, J. Rolim, et al., (Eds.), Parallel and Distributed

Processing – Proc. 15th IPDPS 2000 Workshops, Cancun, Mexico, May 2000, Springer-Verlag, LNCS 1800 (2000) 1236-1247.

[Pree94] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design, in M. Tokoro and R. Pareschi (eds), Springer-Verlag, proceedings of the ECOOP, Bologna, Italy: 150-162.

[Preist99] Chris Preist. Comodity Trading Using An Agent-Based Iterated Double Auction in Proceedings of the third annual conference on Autonomous Agents table of contents, Seattle, Washington, United States, Pages: 131 - 138, 1999.

[Randell75] B. Randell. System structure for software fault tolerance, IEEE Transaction Software Engineering 1, 2, 220-232, 1975.

[SaSh00] J. Saldhana, and Sol M. Shatz. UML diagrams to object petri net models: an approach for modeling and analysis, in: Proc. Intl. Conference on Software Eng. and Knowledge Eng. (SEKE), Chicago, July 2000, pp. 103-110.

[SchMat92] R. Schwarz, and F. Mattern. Detecting causal relationships in distributed computations, In search of the Holy Grail, Department of Computer Science, University of Kaiserslautern, Technical Report SFB124-15/92, Kaiserslautern, Germany, 1992.

[ScSc83] R.D. Schlichting, and F.B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans.Computer Systems, 1(3)(1983) 222-238.

[SDPW+96] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents, Distributed intelligent agents, IEEE Expert, 11(6) (1996) 36-46.

[ShSt01] Onn Shehory, and Arnon Sturm. Evaluation of modeling techniques for agent-based systems, in Proceedings of the fifth international conference on Autonomous agents table of contents, Montreal, Quebec, Canada, Pages: 624 - 631, 2001

[Silva97] L.M. Silva. Checkpointing Mechanisms for Scientific Parallel Applications, Ph.D.Thesis, University of Coimbra, Department of Computer Science, 1997

[SSS+99] L.M. Silva, P. Simões, G. Soares, P. Martins, V. Batista, C. Renato, L. Almeida, and N. Stohr. JAMES: A Platform of Mobile Agents for the Management of Telecommunication Networks, in Proc. 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA'99), Springer-Verlag LNCS 1699 (1999) 76-95.

[StroYam85] R. E. Strom, and S. Yamir. Optimistic recovery in distributed systems. ACM Trans. Comput. Syst. 3,3, 204-226, 1985.

[StYe85] R.E. Strom, and S.A. Yemini. Optimistic recovery in distributed systems, ACM Trans. Computer Systems, 3(3) (1985) 204-226.

[TaKo99] H. Tai, and K. Kosaka. The Aglets project, Comm. of the ACM, 42(3)(1999) 100-101.

[TamSeq84] Y. Tamir, and C.H. Sequin. Error recovery in multicomputers using global checkpoints, In Proceedings of the International Conference on Parallel Processing, 32-41.

[Tanenbaum02] Andrew S. Tanenbaum. Distributed Systems: Principles and Paradigms, 2002

[TongKT92] Z. Tong, R. Y. Kain, and W. T. Tsai. Rollback-recovery in distributed systems using loosely synchronized clocks. IEEE Transactions Parallel and Distributed system, 3, 2, 246-251, 1992.

[Tveit01] Amund Tveit. A survey of Agent-Oriented Software Engineering. in Proceedings of the First NTNU Computer Science Graduate Student Conference. Norwegian University of Science and Technology, May 2001.

[Vaidya99] N.H. Vaidya. Staggered consistent checkpointing, IEEE Trans. Parallel and Distributed Systems, 10(7)(1999) 694-702.

[VeCo02] Mario Verdicchio, and Marco Colombetti Commitments for agent-based supply chain management, ACM SIGecom Exchanges archive Volume 3 , Issue 1 Winter, 2002, Pages: 13 - 23

[WaPW98] T. Walsh, N. Paciorek, and D. Wong. Security and reliability in Concordia, in Proc. 31th Annual Hawaii International Conference on System Sciences (HICSS31), 7(1998) 44-53.

[WoJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems, 3, 285ñ312, 2000 © 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.