An Object-Oriented Framework for Constructing Availability Management Services: System Architecture and Application Development

Xiang Hua Qin

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science
in Electrical and Computer Engineering
at Concordia University
Montreal, Quebec, Canada

July 2004

Abstract

An Object-Oriented Framework for Constructing Availability Management Services:
System Architecture and Application Development

Xiang Hua Qin

With the ever increasing dependence on the distributed computing services, service availability is becoming an important factor in building today's distributed application. To achieve fault-tolerance and high availability in modern computing infrastructures, providing redundancy in hardware, software and information is a well established approach to provide reliable services. The idea behind one such redundancy management scheme namely *server clustering* is to implement any application service that must be available to users despite hardware or software failure by a group of redundant software servers which run on distinct hosts and maintain redundant information about the global service state. To accomplish high availability of the service, the software system that facilitates implementation of application services must provide for dependability attributes through a set of primitive dependable distributed protocols in order for the service to continue functioning despite some number and types of failures.

We describe an object-oriented framework for designing and implementing availability management services. We present a library of object-oriented implementation of a suite of dependable distributed protocols, and show how these protocols can be composed together to build an application that provides highly available services to users. Specifically, we introduce our proposed system architecture called JAMS (Java-based Availability Management System) which integrates the availability management service and the online FDIR (Fault diagnosis, isolation and reconfiguration) service on a distributed heterogeneous platform. The JAMS architecture is described in a way that reflects the design and development phases, spanning from requirement analysis to module implementation. A case study of a banking service is presented to illustrate how JAMS can be used to build a dependable distributed application in a modular manner ensuring that the application service remains continuously available to the user despite the presence of failures, maintenance and growth.

Dedicated to my beloved dad and mom......................

Acknowledgments

I would like to thank my academic supervisor, Dr. Purnendu Sinha, for all his patience, support and guidance throughout my graduate studies. His supervision is proved to be invaluable in helping me complete what at times seemed to be an insurmountable task.

Most of all, I would like acknowledge my parents, sisters and brother for their support and understanding throughout my academic endeavors. Without their love and encouragement, I could not have completed this thesis.

I am grateful to Mr. Alfred Dupuis, Mr. Guo Ping Chen and Mr. Ashish Tiwari for their help in editing this thesis. I extend my sincere thanks to Mr. Serguei Mokhov for his sharp, clear-cut explanation on how to get things done with Java. Finally, I would like to thank all my colleagues in the research group for their valuable discussions during this research.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce basic concepts and terminologies with regard to JAMS (Java-based Availability Management System). JAMS is implemented by integrating the Availability Management Service and Fault Management Service namely FDIR (Fault Detection, Isolation and Recovery) in a distributed system. After highlighting the increasing importance of such a system like JAMS to build an application service in distributed system, we describe the properties of Availability Management Service and FDIR, respectively, and then discuss the features of JAMS by comparing it with related work. Object-Oriented Programming and Java technology are introduced to help the reader better understand this work. We conclude this chapter with contributions of the thesis.

1.1 Fault Tolerance in Distributed Systems

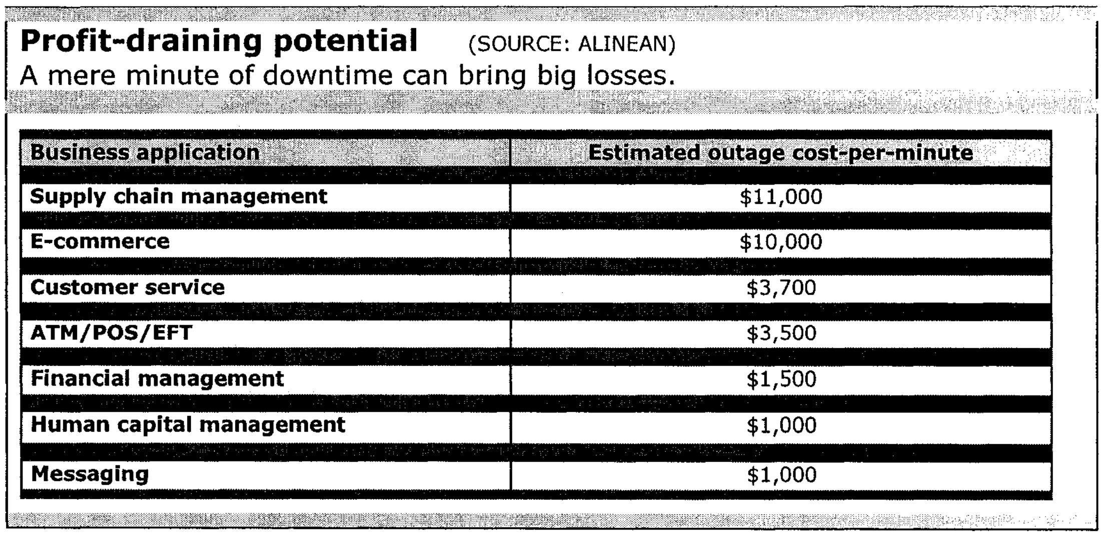
Distributed Systems are computer networks that consist of clients and servers connected in such a way that one system can potentially communicate with another. Data is not located in one server, but in many servers. These servers might be at LAN or WAN. At present, the Internet and web technology have greatly expanded the concept of distributed systems. Now, web is typically defined as "massively distributed collection of systems" [25].

Web application servers provide universal access to any client with a browser. They tend to grow up in order to meet the needs of industrial network applications and E-business. Unlimited communication and information exchange prevail, as they become compatible with different types of computing platforms and operating systems.

A distributed environment has many distinct characteristics, one of which is providing distributed data protection from local disasters. Data may be replicated to other systems to provide fault tolerance, and a distributed system may be reconstructed from replicated data so that it continues functioning despites the failure of some components.

With the ever increasing dependence on the computing services, service availability is becoming a matter of paramount importance in building today's distributed application. Distributed systems are much more complex in the sense that they have a large number of nodes that may fail independently and many activities that run concurrently. In any business, downtime or unavailability of services may cost an excessive amount of money in addition to operation cost. A conservative estimation from Gartner [23] pegs the hourly cost of downtime for computer networks at $42,000. For large e-commerce dependent sites, outages are more costly. EBay's 22-hour crash in June

1999, for instance, cost the company more than $5 million in returned auction fees. The estimated costs for a variety of network applications are outlined in Table 1 [23]:



Profit-draining potential (SOURCE: ALINEAN)
A mere minute of downtime can bring big losses.

Business application	Estimated outage cost-per-minute
Supply chain management	$11,000
E-commerce	$10,000
Customer service	$3,700
ATM/POS/EFT	$3,500
Financial management	$1,500
Human capital management	$1,000
Messaging	$1,000

Table 1: Estimated costs for a variety of network applications

Fault tolerance and high availability is about keeping systems up and running 24 hours a day, 7 days a week, or at least keeping systems up and running with a reasonable amount of performance. Fault-tolerant features in early network operating systems included mirrored disks, with both disks reading and writing the same information. If one disk failed, the other kept running in what is called "failover" mode. Of course, nowadays fault-tolerant systems must provide more than just disk failover. In modern computing infrastructures, different ways to achieve fault tolerance and high availability are the following:

- Redundancy in hardware (e.g.: disk mirroring and duplexing, redundant communication links)
- Redundancy in software (e.g.: exception handling, multiple-version software)

- Redundancy in information (e.g.: server clustering, distributed computing, duplicate data centers)

Therefore, redundancy management protocols are being designed and developed to handle specific tasks having abilities to correctly detect, diagnose, and recover from errors. The idea behind one such redundancy management scheme namely "server clustering" is to implement the application service that is required to be always available by a group of redundant software servers which run on distinct hosts and maintain redundant information about the global service state. The clustered servers work as a central hub for an application service, like air traffic control, banking, and real-time control, and so on. In such a critical system, the system availability concept is an important aspect that needs to be considered early in the system's design phase. To accomplish high availability of the service, the software system that facilitates implementation of application service must provide for dependability attributes through a set of primitive dependable distributed protocols in order for the service to continue functioning despite some number and type of failures.

1.2 Java-based Availability Management System

Towards the objective discussed above, a system architecture called JAMS (Java-based Availability Management System) is designed for constructing a fault-tolerant application software by using the redundancy management protocols. JAMS is an object-oriented fault-tolerant system, which integrates the Availability Management Service and the online FDIR (Fault Detection, Isolation and Recovery) service on a distributed platform. JAMS is aimed at helping an application programmer to build application services which can remain continuously available to users despite the presence of failures, maintenance and growth.

1.2.1 Availability Management Service

Availability management service is being developed for the purpose of keeping the critical services of a distributed system continuously available to users despite the presence of system failures, maintenance and growth [14].

In present systems, human system operators take responsibility of reconfiguring a system after failures or removal of nodes for maintenance. They tend to have comparatively slow reaction times, and this may result in an excessive interval time for critical services to become available again. System operators are likely to make mistakes while repairing, even worse this may lead to further failures, causing further unavailability. Therefore it is crucial to develop Availability Management Service to conduct automatic reconfiguration in the presence of failures and maximize the availability of the critical services.

Availability Management Service is focused on realizing automatically the availability specified for critical services offered by a distributed system to end users. A *primary* server provides the current service while a *backup* maintains past service states. Suppose a group of servers provide a critical service *S* and its *primary* server fails for some reasons. In this case, the Availability Management Service first promotes the *S* *backup* to the *primary* and then starts another *S* *backup*, rather than just cold-start another *primary* for *S*. This process will greatly reduce the duration the service is unavailable to users, since it is much faster to promote the *backup* to the *primary* by software than to cold-start a new *primary* by human. The detailed requirement and description of this service is discussed in Section 2.2.5.

The efficiency and usefulness of such highly available system can be improved by integrating FDIR service which provides a set of tools to support distributed systems dependability aspects, such as fault detection, group membership, atomic broadcast, checkpointing, etc. For instance, availability management service takes action of promoting after fault detection detects a fault in *primary* server.

1.2.2 FDIR service

FDIR is a fault management service that employs a set of distributed protocols to achieve desired dependability attributes in the presence of faults through a process of fault detection, fault isolation and resource recovery/reconfiguration.

The main goal of FDIR is to preciously diagnose faults, effectively prevent the faults from creeping into a system, and timely recover system by rolling back to a moment at which the service state is assured to be fault-free, thereby regaining operational status. This capability leads to reduction in diagnostic time or downtime, and therefore increases system availability. The following requirements are essential to the overall correctness of the protocols in FDIR:

- All of N redundant units perform identical operation with identical inputs. N must be at least 2m+1, where m is the maximum number of faulty units.
- The clocks of correct nodes are approximately synchronized with a maximum allowable deviation.
- The broadcast of message between nodes must be atomic and reliable. All correct nodes deliver a message at same synchronized time.
- The diagnosis does not require administered testing. The determination of a faulty

node is done collectively by other nodes in the system. Every faulty node is identified by all other non-faulty nodes.

- All nodes have an identical view of group membership at any time.
- Either all or none of the nodes perform recovery, and the nodes only roll back to a consistent system state.

Through these identified requirements, it can be inferred that the following building block protocols are essentially needed. The detailed discussion is given in Section 2.2.

- Clock synchronization ensures that the clocks of active nodes are synchronized within certain known constant deviation at any point in real time.
- Atomic broadcast is used to achieve the atomicity, order and bounded communication of message exchange, thus to provide consistent information to multiple nodes in the distributed systems.
- Voting function is used to compare the output in order to make a decision while multiple system entities execute the same task with the identical inputs.
- Online fault diagnosis is responsible to identify the faulty node. It is achieved through constantly monitoring, message exchange among redundant systems and determination of the faulty node by the majority of votes.
- Membership protocol identifies all functioning servers that cooperate to provide the service and organizes them into a group that exists over time.
- Check pointing is commonly used technique for recovering system from failure. It establishes periodically consistent checkpoints. In case of failure, it rolls back the system state to a previous error-free checkpoint.

1.2.3 Object-Oriented Paradigm and Java

In object oriented programming, a model is created for a real world system. Classes are programmer-defined types that model the components of the system.

A class is a prototype that defines the variables and the methods common to all objects of a certain kind. An object is a software bundle of related variables and methods. Using the prototype provided by a class, the programmer can create a number of objects, each of which is called an instance of the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. Software objects interact with one another by exchanging messages. A class can inherit state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

Java technology is both a programming language and a platform. Java language is an object-oriented programming language developed by Sun Microsystems. It has become one of the most popular network programming languages due to a number of features including simplicity, object oriented, distributed, robust, secure, architecture neutral, multithreaded, etc. The Java platform differs from most other platforms in that it is a software-only platform that runs on top of other hardware-based platforms. The *Java Virtual Machine* (JVM) is the base of the Java platform and can be implanted onto various hardware-based platforms. In other words, it means that as long as a computer has a JVM, the same program written in the Java programming language can run on different operating systems like Windows, or Solaris.

1.2.4 Features of JAMS

JAMS is an object-oriented fault-tolerant system that can be characterized by the following key features:

- *Fully Object-Oriented.* The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. Every component in JAMS is objectified without much regard to whether it is the user interface or the protocol abstraction.
- *Modularization.* The objectified components are organized into modules according to their functionalities. Each module is independent of each other. This modularization enables optimal code reuse.
- *Portable.* JAMS is designed to support applications that will be deployed into heterogeneous network environments. Pure Java programming exempts JAMS from the platform dependencies.
- *Autonomous.* JAMS is running continuously to monitor the system. When the primary server fails, the backup is promoted automatically. The availability of the service is achieved without human intervention.
- *High performance.* JAMS achieves superior performance by adopting a scheme in which multiple threads concurrently execute. Each module runs as a prioritized thread. The fault detection module and availability management module have high priority while others run as low priority threads, ensuring a high probability that resources are available when required, leading to higher performance.
- *Self fault-tolerant.* The protocols chosen to implement JAMS are all elaborately concocted to tolerate faults. Thus, they enable JAMS to make accurate judgment.

1.3 Related Work

By making use of redundant software server group, many other prototypes of fault-tolerant distributed systems have been built, often by emphasizing different aspects of fault tolerance, communication, distribution, action atomicity, persistence etc. Some of the examples of successful dependable distributed systems are: TANDEM [1], IBM XRF [1], CONSUL [28], HOROUS [28], ARJUNA [21] and BAST [17].

In TANDEM, the operating system implements the pair-management algorithms and uses duplex resources to mask hardware-resource failure from users. The IBM XRF system provides continuous database service by using a group of two database servers running on two distinct high-end processors connected by point-to-point local area network. The duplication of hardware resources needed by software servers makes the TANDEM and IBM XRF single-fault tolerant. Systems like HORUS and CONSUL provide reliable distributed protocols, but in their protocols, "group" is the elemental distributed addressing facility. ARJUNA and BAST are two of best known examples that have adopted object-oriented approach. ARJUNA is implemented in C++, which deals with failures by providing high level application-building tools based on transaction, while BAST mainly concerns with composition of distributed protocols such as fault detection and consensus. In contrast to system like ARJUNA on one hand, or HORUS on the other hand, BAST is neither a transaction-oriented nor a group-oriented system.

Compared to JAMS, BAST is the closest among all systems that we have come across. In both systems, protocols are basic structuring components and the protocols/algorithms are objectified as separate objects. BAST adopted a strictly vertically-layered approach in which the protocol dependencies are modeled as layer interactions. There are several drawbacks with this approach. First, protocol classes are unlikely ready-to-use. Protocol programmer has to build a new protocol class with desired semantics by deriving them from all the necessary protocol classes. Furthermore, the objects of derived protocol classes are quite difficult to be manipulated at runtime, since there are concurrently executing more than one protocol at the same time in an object.

In contrast to BAST, our system chooses different set of protocols to build a framework. Furthermore, all requirements of FDIR and availability management have been considered and are fully implemented in JAMS. The remarkable difference with other systems, especially with BAST, is that it adopts different design pattern to manage distributed protocol interactions. Protocols are all assembled in one horizontal layer, so called *protocol layer*. The interaction between protocol objects at run time is achieved through inner-class design and pipe mechanism. Each protocol is designed as a module-class to communicate with other modules through pipes. Communications between objects within one module are achieved by inner classes. Based on such a design, protocol classes are independent with each other and would be easily implemented without considering their synchronization. It enables optimal code reuse, for example, if we want to use a new fault detection protocol, we simply need to replace current online diagnosis protocol class with a new one, leaving all other module-classes unchanged since they have no relationship between each other in the design phase. Finally, additional protocols can be added in JAMS without considering the intricacies of protocol interactions.

1.4 Contributions

The specific contributions of the thesis include our objectives:

- To identify building block protocols, highlight and address issues involved in block interactions and inter-dependencies within the class of redundancy management protocols.
- To structure the complexity involved in block interactions and propose an object-oriented solution for JAMS by developing classes for individual building block protocols and subsequently integrating them.
- To implement JAMS for supporting most application services across different platforms.
- To evaluate the robustness of JAMS by applying it to an application service.

Each of the above stated objectives has been met successfully, and we will elaborate on these in subsequent chapters.

1.5 Organization

In the next chapter, we present the building blocks pertaining to JAMS. Their inter-dependency is also discussed in this chapter. Chapter 3 introduces our proposed object-oriented design of JAMS and describes the dynamic behavior of protocol modules. In chapter 4, we present a case study of a banking service to illustrate how we can use JAMS to actively replicate a server object in some distributed applications, in order to make it fault-tolerant and highly available. Chapter 5 concludes the thesis with discussions and future research directions.

Chapter 2

Background of Dependable Protocols

As mentioned in Chapter 1, our system is designed to build a reliable distributed system that is capable of correctly detecting, diagnosing, and recovering from errors, and is based on a number of redundancy management protocols as building blocks to perform each of these specific tasks. In this chapter, we identify these building blocks, and outline a framework which incorporates these basic building blocks in JAMS. We start with introducing the system model and fault model. Next we describe a primitive building block of synchronization protocol, and then subsequently discuss other building blocks in FDIR. The availability management service is as well discussed by highlighting its key functions and features. Finally, we highlight the block interactions and inter-dependencies and outline a framework which encompasses these building blocks for JAMS considering their interactions at design phase as well as at run time.

2.1 System Model and Fault Model

2.1.1 System Model

We consider a system consisting of distributed nodes linked by a synchronous communication network which enables any two active nodes to communicate within a known, bounded time, given that there is no network partition. For simplicity, we assume that the nodes are uniprocessor. The proposed approach is applicable to multiprocessor as well.

The synchronous communication network provides a datagram service and a message diffusion service that allows both unicasting and multicasting. When an active node issues a message and broadcasts it to the network, all active nodes will eventually receive it. Communication services are implemented by the underlying network communication protocols. TCP/IP has become the de facto standard network communication protocol, and has been used in JAMS as well.

The dispersal of information is assumed to be symmetric, not asymmetric. The comparison of these two dispersion mechanism is shown in Figure 2.1. When a node transmits information to the network, symmetric dispersal indicates all receivers obtain the same information while receivers get different information with asymmetric dispersal.

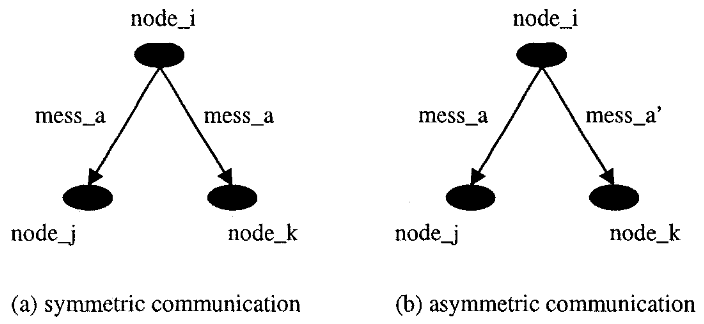


Figure 2.1 Symmetric and Asymmetric Communication

To achieve fault tolerance, distributed system architecture incorporates redundant processing components. The following definition of terms stated here is based on [1]. A computing *service* specifies a collection of operations whose execution can be triggered by inputs from service users. The operations defined for a specific *service* are performed by a *server* for that *service.* The behaviors of a *service* can only be carried out by a number of redundant *server*s in distributed systems. The set of functioning *servers* form a *group* to cooperate so as to provide the services and each server works as a *member.* If a node possesses all physical resources needed for running a certain service, it is called a potential host for that service. There is a total order on the set P of node identifiers. A service implemented by a server relies on services implemented by other servers. It is called service dependency. That is to say a service u depends on a service r if the correctness of u's behavior depends on the correctness of r's behavior. In JAMS, each protocol provides a service. For example, *Broadcast Service* that we make use of is based on the *Atomic Broadcast* protocol. The application service is also an example of service extended by a server. In JAMS, service dependencies are interpreted / referred as inter-dependencies of building block protocols.

2.1.2 Fault Model

It is presumed that all the communication links are non-faulty and that nodes are the only potentially fault units. This fault model is termed as *PP* (processor- processor) [7]. It is assumed that the datagram service could suffer omission or performance failures. Failures in the underlying network such as lost, late, duplicated, or corrupted messages are supposed to be overcome by the TCP/IP.

A node is perceived to either work correctly or become faulty which can be the case of having benign faults, value faults or crash faults. The PP model that we utilize is flexible to cover a range of fault types. This is done to incorporate a realistic system environment where faults do not match idealized fault conditions of only a single specific fault occurring over system operations. In *PP*, we classify the faults that an individual node is sufficient to detect as benign fault since the fault-effect is locally detectable. On the other hand, there are situations that require multiple nodes to exchange their syndrome information with each other in order to provide accurate diagnosis. The value faults and crash faults are classified as globally detectable faults. A value fault occurs when the server gives the incorrect output in response to inputs. The node is said to suffer the crash failure if the server omits to produce output to subsequent inputs till it restarts.

2.2 Building Block Protocols

For each protocol, we give a brief discussion by highlighting its features and requirements. For details we refer the reader to [4, 6, 7, 11, 14, 26, 27].

2.2.1 Clock Synchronization

Synchronization is a fundamental issue in distributed system. The clock of servers are synchronized with some known constant maximum deviation at any point in real time and such synchronized clocks run within a linear envelope of real time [4]. A processor is synchronous if it always performs its intended function within a known time limit.

We denote by C_p the clock of processor p and use $C_p(t)$ to denote p's local clock time at real time t. We say the processors are synchronized if the following properties are satisfied:

- Every processor p has a local C_p with known bounded rate of drift ρ respecting to real time. That is, for all real time $t \geq t'$,

$$(1+\rho)^{-1} \le \frac{C_p(t)-C_p(t')}{(t-t')} \le (1+\rho)$$

- For any correct processors p and q, and for any real time t, clocks are within a maximum deviation σ and are within a linear envelope of real time where

$$\left|C_p(t)-C_q(t')\right| \le \sigma$$

Through clock synchronization primitive, it is possible to measure message timeout, and this provides a mechanism that makes possible the implementation of many other distributed protocols, such as atomic broadcast, fault detection, and availability management

2.2.2 Atomic Broadcast

The objective of atomic broadcast is to enable the correct processor of a distributed system to attain consistent knowledge of the system state, and achieve the atomicity, order and bounded communication of message exchange, despite failures and random communication delay. It relies on the *clock synchronization* protocol. An atomic protocol is a protocol that satisfies the following properties [11]:

- *Atomicity:* If an update is initiated by a correctly functioning member at clock time T, then at $T+D$, where D is time constant, the update is either delivered to all functioning members or not delivered to any correctly functioning members.
- *Total Order*: All updates delivered by correctly functioning members are delivered in the same order at each correctly functioning member.

- *Termination*: If a correctly functioning member broadcasts an update *m* at time *T* on its clock, then *m* is delivered by all correctly functioning members at time *T+D* on their clock.

With these properties, atomic broadcast can be used to implement the abstraction of synchronous replicated storage: a distributed, resilient storage that display, at any clock time, the same contents at every correct physical processor and that requires *D* time units to complete replicated updates. This satisfies the need of membership and the availability management primitives.

2.2.3 Membership

Membership protocol identifies all correctly functioning servers that cooperate to provide the service and organizes them into a group that exists over time. It provides all correctly functioning members with consistent views of the membership, which can shrink with crashes or grow with joins, and guarantees bounded failure detection and join delay [6]. A membership service is required to satisfy the following properties:

- *Stability of local views*: After a processor joins a group as a member, it stays joined in that group until a failure is detected or a processor start occurs.
- *Agreement on history*: let processors *p* and *q* be correct throughout a certain time interval. If during that interval, *p* and *q* are joined to a common group *G1* and after leaving *G1*, *p* joins group *G2* and *q* joins group *G2'*, then *G2=G2'*.
- *Agreement on group membership*: if two correct members *p* and *q* are joined to the same group, then the two processors have the same view of membership of that group: *members(p)=members(q).*
- *Bounded join delay*: there exists a time constant *J* such that, if a processor *j*

starts at time T and j stays correct until time $T+J$, then by time $T+J$, the processor j joins a group that is also joined by each other processor that was correct throughout $[T, T+J]$.

- *Bounded failure detection delays*: there exists a time constant D such that, if a processor f joined to a group G fails at time T, then each member of G that stays correct throughout $[T, T+D]$ joins a group G' by $T+D$ such that f is not a member of G'.

The above requirements imply that two services are fundamental to implement this membership protocol: clock synchronization and atomic broadcast. For example, the total order on the delivery of group atomic broadcast messages enable all correct members see the same changes of the membership in the same order.

Next, we describe a simple "periodic broadcast membership" protocol. A membership server j that starts at time S invites the other servers to form a new group by broadcasting a "new-group" message time-stamped S. In response to a "*new-group*" message, each server broadcasts a "present" message that contains its identifier and indicates its willingness to join a new group by time $V=S+\Delta$, where Δ denotes the bound of atomic broadcast delay. We call the set of servers at time V *the membership as of view time* V, denoted by *MEMBERS (V)*. With this method, all nodes, which had left the group earlier and later joined *MEMBERS (V)* again, will have the same view of membership at time $C=V+\Delta$. This task of identifying membership is scheduled periodically by all members of *MEMBERS (V)*. At an interval of π, each server broadcasts a "*present*" message and re-computes the membership at membership-confirmation time $V+ k\pi + \Delta$, where k is some integer and $\pi > \Delta$ in order to let server know the membership as of view time *V* before it checks next "*currentness*". Thus, the membership is sampled with a sequence of snapshots taken at V, $V + \pi$, $V + 2\pi$, etc. The join of new member at time *W* leads to cancellation of previously scheduled task of identifying membership, and rescheduling of a new task with new check time $O=W+ k\pi$. The following pseudo code [6] gives an overview on this membership protocol.

```
task Membership;
var group: Time; members: set of P initially{ };
broadcast("new-group", myclock+ Δ);
cycle
  when receive("new-group", V)
  do
     if (myclock > V) then abort fi;
      cancel(Broadcast);
      broadcast("present", V, myid)
      schedule(Broadcast, V+π)
  od;
endcycle;

task broadcast (V:Time)
if (myclock> V) then abort fi;
broadcast("present",V, myid);
schedule(Broadcast, V+π)
```

By this membership protocol, the crash failure of server *f* can be easily detected from the formation of a new group from which *f* is excluded. Suppose server *f* is a member of *MEMBERS (V)* and fails at time $V+\eta$, where $\eta < \pi$. By time $V+\pi$, all members of *MEMBERS (V)* except the failed server *f* broadcast a "present" message. At membership confirmation time $V+\pi+\Delta$, assured by the atomicity and termination properties of atomic broadcast, all surviving members of *MEMBERS (V)* detect that *f* is not in *MEMBERS (V+ π)* and they join a new group of *MEMBERS (V+ π).*

Due to a node join or crash, the change of membership is sent to *availability management* service that we will discuss in Section 2.2.6.

2.2.4 Online Fault Diagnosis

Fault diagnosis is responsible to identify the faulty node so as to restrict the influence of faults in the system operation and to support the fault isolation and reconfiguration processes of FDIR. A variety of approaches have been proposed for system diagnosis. The online diagnosis approach that we are going to discuss, unlike the existing fault detection techniques, does not require administered testing but is achieved through constant monitoring and exchange of message among redundant system functions. Through this assimilation of messages, it then accurately determines the faulty node by the majority voter which guarantees that, given a majority of non-faulty inputs, the error will be masked [8]. This online diagnosis approach is itself tolerant to faults in the diagnostic process.

As discussed in our fault model, all faults are classified as 1) local-classification of locally detectable fault-effects at node level, and 2) global-classification based on the nodes exchanging of their local-classification to make a global judgment at the system level. Hence, the diagnosis process is characterized with two phases [7]:

- Phase1: Local diagnosis syndrome formulation based on a node's local perception of other nodes' fault status. This is established by a node's self analysis of incoming message traffic from other nodes.
- Phase2: Global diagnosis syndrome formulation through exchange of local diagnosis syndrome information with all other system nodes.

The detail of algorithm is illustrated as following, where *n* represents frame number, and V_i is the result of node *i*'s voting process on the inputs received over frame *n*.

Online Diagnosis Algorithm [7]

```
Round($n$), $n$>0
    1) Each node $i$ monitors all received messages over frame n, executes frame n
       of the workload, and arrives at a voted value $V_i$.
    2) Each node sends $V_i$ to all other nodes.
    3) Each nodes $i$ compares incoming messages to its own voted value $V_i$:
        • If the value from $j$ does not match, is missing, or there is an
          accusation from the last frame of $i$ against $j$, $i$ records that $j$ is BAD.
        • Otherwise, $i$ records that $j$ is GOOD.
    4) Each node $i$ sends its report on each other processor to all processors.
    5) Each node $i$ collects all votes from other node $j$:
        • If the majority of votes are BAD, then node $i$ declares $j$ is faulty.
          Furthermore, node $i$ records an accusation against any processor $k$
          that voted $j$ as GOOD.
        • If the majority of votes are GOOD, then $i$ records an accusation
          against any node k that voted $j$ as BAD.
```

The online diagnosis algorithm is required to satisfy two specific properties [9]:

- Correctness: every node diagnosed to be faulty by a non-faulty node is indeed faulty, or we can say if a good "*i*" declares "*j*" faulty, "*j*" is indeed faulty.
- Completeness: every faulty node is identified, or we can say if "*j*" is faulty, then all good processors diagnose "*j*" as faulty.

Both requirements guarantee the online diagnosis algorithm has sufficient capability to support the fault isolation and reconfiguration process.

2.2.5 Checkpointing

Checkpointing is a common technique used in conjunction with recovery method to speed up the recovery time. It establishes consistent recovery points periodically. In case of failure, it rolls back the system state to a previous error-free checkpoint and continues from that point.

As our system is not intended to design as a transaction-oriented system, we don't adopt any transactional protocol. To accommodate this point, we develop a simple *synchronous checkpointing* protocol which depends on *clock synchronization* and *atomic broadcast* protocols. It can be simply described as follows:

- There are two types of checkpoints [26]. Permanent checkpoint is a local checkpoint at a node, and part of a consistent global checkpoint. Tentative checkpoint refers to the temporary checkpoint that is made to be a permanent checkpoint at the successfully termination of a checkpointing cycle.
- Every Ω period, each node takes a tentative checkpoint which records its current state and makes the last tentative checkpoint to be permanent. The permanent checkpoint is written to a stable storage. This process refers to a checkpointing cycle.
- When a *primary* updates its state, it broadcasts an update *m* to all nodes at time *T*.
- When a node receives the update *m* from *primary*, it updates its current state with *m*. By the properties of atomic broadcast, at time $T+\Delta$, all nodes will have the same state on the primary.

- Upon a fault being detected at *primary*, all nodes recover which results in canceling the current checkpointing cycle, rolling back the current state to the permanent checkpoint and rescheduling checkpointing with period Ω.

This Checkpointing algorithm requires two basic properties:

- Correctness: Only the error-free tentative checkpoint can be made to be permanent checkpoint. Nodes roll back only to their permanent checkpoints.
- All or nothing: Either all or none of the nodes takes permanent checkpoints.

2.2.6 Availability Management

Availability management (AM) service is a distributed system service that enables the critical service of a distributed system to remain continuously available to users despite arbitrary numbers of concurrent node removals, node joins, node crashes and/or node failures.

Availability management service depends directly on the *atomic broadcast*, *membership*, and *fault diagnosis* blocks. It will be notified by *membership* of any node join, leave or crash; It will notified by *atomic broadcast* when *fault diagnosis* detects any faulty node. AM service is also based on the following assumptions [14]:

- *Loose synchronization* among the servers for the critical service *S*: a *primary* maintains the current service state, while one or more *backup*s maintain past service states. The consistency of state information between loosely synchronized servers can be maintained by periodic checkpoints.
- *Location transparent:* we assume a *directory naming service* that keeps track of resources and location information, and a request/reply transport service ensures that server migrations are transparent to users.

```
Const P: set of all nodes of the system
var N: Set of P init { };  //set of active nodes or hosts
var on: Boolean init false; // on=true when $S$ is started
var primary: node init $\perp$ ; //points to node hosting primary for $S$, $\perp$ : undefined value
var backup: node init $\perp$ ; //points the node hosting backup for $S$
```

```
start-service() $\equiv$
    on $\leftarrow$ true;
    start-servers(N);

start-servers(A: set of P) $\equiv$
    start-primary(A);
    if primary $\neq \perp$
    start-backup(A-{primary})
    fi;

stop-service() $\equiv$
    on $\leftarrow$ false;
    if backup $\neq \perp$ ;
    stop server for $S$ on backup;
    backup $\leftarrow \perp$ ;
    fi;
    if primary $\neq \perp$
    stop server for $S$ on primary;
    primary $\leftarrow \perp$;
    fi;

faulty-node(n) $\equiv$
      N $\leftarrow$ N-{n};
      if myid=n then exit();
      if primary=n
      if myid=backup
      broadcast a "recovery" command;
      sleep(50); //wait checkpoint recovery
      fi;
      primary $\leftarrow \perp$;
      promote-backup(N);
       fi;
       if backup=n
       backup $\leftarrow \perp$;
       start-backup(N-{primary});
       fi;

add-hosts(h) $\equiv$
    $N \leftarrow N \cup \{h\}$

start-primary(A: set of P) $\equiv$
    if primary $= \perp$ and A $\neq$ { }
    primary $\leftarrow$ select-host(A);
    start primary server for $S$ on primary;
    fi;

start-backup(A: set of P) $\equiv$
    if backup $= \perp$ and A $\neq$ { }
     backup $\leftarrow$ select-host(A);
     start backup server for $S$ on backup
     fi;

promote-backup(A: set of P) $\equiv$
      if backup $\neq \perp$ and A $\neq$ { }
      promote backup server for $S$ on backup
      primary $\leftarrow$ backup;
      backup $\leftarrow \perp$;
      start-backup(A-{primary});
      fi;

crash-node(n) $\equiv$
       N $\leftarrow$ N-{n};
        if primary=n
        primary $\leftarrow \perp$;
        promote-backup(N);
        fi;
        if backup=n
         backup $\leftarrow \perp$;
         start-backup(N-{primary});
         fi;
```

Figure 2.2: State transitions of Availability Management Service [14]

- *Automatic reboot:* nodes can automatically reboot after a crash or a failure. It ensures that the number of servers for the critical service will not shrink greatly in a certain time period, thus to maintain availability autonomously even there is no human interference for a long time.

To ensure that the availability management service is itself available as long as at least one node is active, a team of availability management servers is hosted by all nodes of the system. For simplicity, we assume availability management service is responsible for only one critical service *S*, and all active nodes are hosts of service *S*. Various constants, variables being used and operations of availability management service are depicted in Figure 2.2. The operation *faulty-node(n)* is an additional operation made over the original protocol in view of the case that a faulty node have been detected by online diagnosis protocol.

2.3 Inter-dependencies of Building Block Protocols

Having identified the necessary building blocks and pointed out the role of each building block in achieving the overall objectives of JAMS, we highlight issues involved in block interactions and inter-dependencies which we have to manage not only during the design phase (between the protocol classes) but also at run time (between the protocol objects). An important policy in composing protocols is to establish the consistency of specifications across various constituent blocks by demonstrating that the requirements of these building blocks are non-conflicting [15]. According to this policy, we elaborate the inter-dependencies of these building blocks in JAMS framework, shown in Figure 2.3.

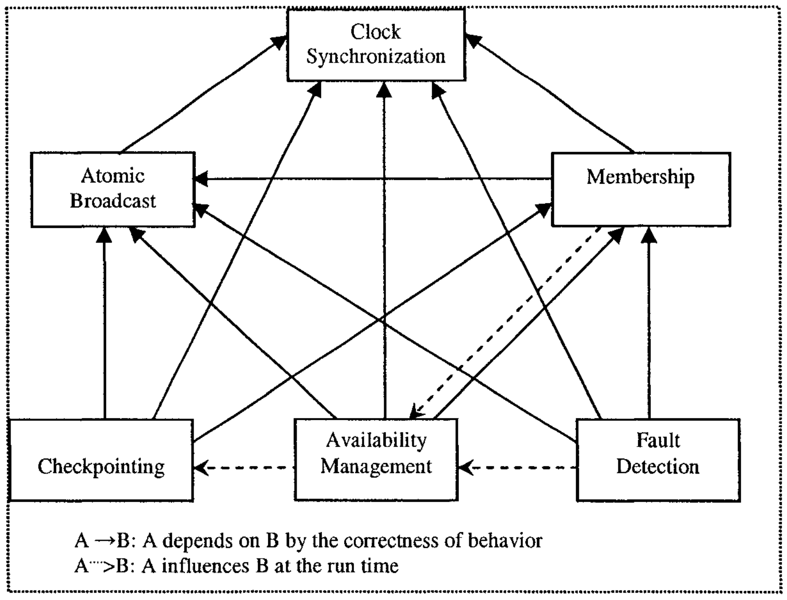


Figure 2.3: Inter-dependencies of Building Blocks

Based on the interaction of the building-block protocols in this figure, we identify the exact conditions that a particular block imposes on other blocks. This helps ascertain the exact requirements that need to be observed across the blocks. As we have already identified the specific attributes of each block, we summarize, in Table 2.1, the conditions and requirements for each block-interaction in Table 2.1.

We can address building blocks dependencies and identify the governing condition by analyzing the dependency paths as well as influence paths as depicted in Figure 2.3 along with the reference to Table 2. For example, in a dependency path of *AM* (*Availability Management*) →*Membership*→ *Broadcast* →*Clock Synchronization*, *AM* depends on *Membership* directly, and the *Membership* relies on the *Atomic Broadcast* to get the membership agreement. The influence of *Atomic Broadcast* on the *Membership* is that the underlying system model must be synchronized. This constraint is satisfied by the properties of *Clock Synchronization.* As an example, the influence path *Fault Detection* → *AM* → *Checkpointing* shows their run-time interaction when a fault is detected. The presupposition of *AM* asking *Checkpointing* for recovery is that a benign or value fault has been accurately diagnosed by the *Fault Detection.*

Block Interactions	**Primary Attributes**
Sync.→ Broadcast Sync.→ Membership Sync.→ Fault Detection Sync.→ Checkpointing Sync.→ AM Sync.→ FDIR	synchronized clocks bounded drift rate bounded clock skew
Broadcast → Membership Broadcast → Fault Detection Broadcast → Checkpointing Broadcast → AM Broadcast → FDIR	synchronous system model synchronized clocks bounded delay
Membership → Fault Detection Membership → Checkpointing Membership → AM Membership → FDIR Membership --> AM	unpartitioned network synchronous system model synchronized clocks bounded communication atomicity and ordering crash fault
FaultDetection --> AM FaultDetection → FDIR	unpartitioned network synchronous system model synchronized clocks bounded communication group Communication benign fault and value fault
AM --> Checkpointing AM → FDIR	unpartitioned network synchronous system synchronized clocks bounded communication atomicity and ordering group Communication
Checkpointing → FDIR	unpartitioned network synchronous system synchronized clocks bounded communication atomicity and ordering coordinated checkpoints group Communication

Table 2: Block Interactions and Primary Attributes

In this chapter, we have identified the building blocks that are used to implement JAMS and their dependencies. It is to note that the relationships between those distributed protocols at run-time are quite complex. In the next chapter, we address the complexities in protocol interactions in our object-oriented design of JAMS framework.

Chapter 3

Objected-Oriented Design of JAMS

As discussed in Chapter 2, composition of building-block protocols is a challenging task since the protocols are interacting with one another not only during the design phase but also at run time. With this viewpoint, we now introduce the object-oriented design of JAMS in detail. We first present the overview of JAMS system architecture, and describe how the framework integrates the building blocks as modules in this architecture. Then we give an in-depth explanation of the proposed modular design. Finally, we illustrate the dynamic behavior of each module through a set of sequence diagrams.

3.1 System Architecture

Since the relationships between distributed protocols are quite complex, the first step is to structure this complexity. Figure 3.1 presents an overview of object-oriented system architecture design of JAMS. The JAMS architecture is structured in two layers: Communication layer and Protocol layer. The communication Layer is constituted by three communication modules. This layer is responsible for sending messages to a multicast socket, receiving messages from network, and delivering them to the corresponding protocol module in protocol layer. The protocol layer consists of five protocol modules. Each protocol module includes a protocol class that is implemented with the specification of basic building block protocols identified earlier.

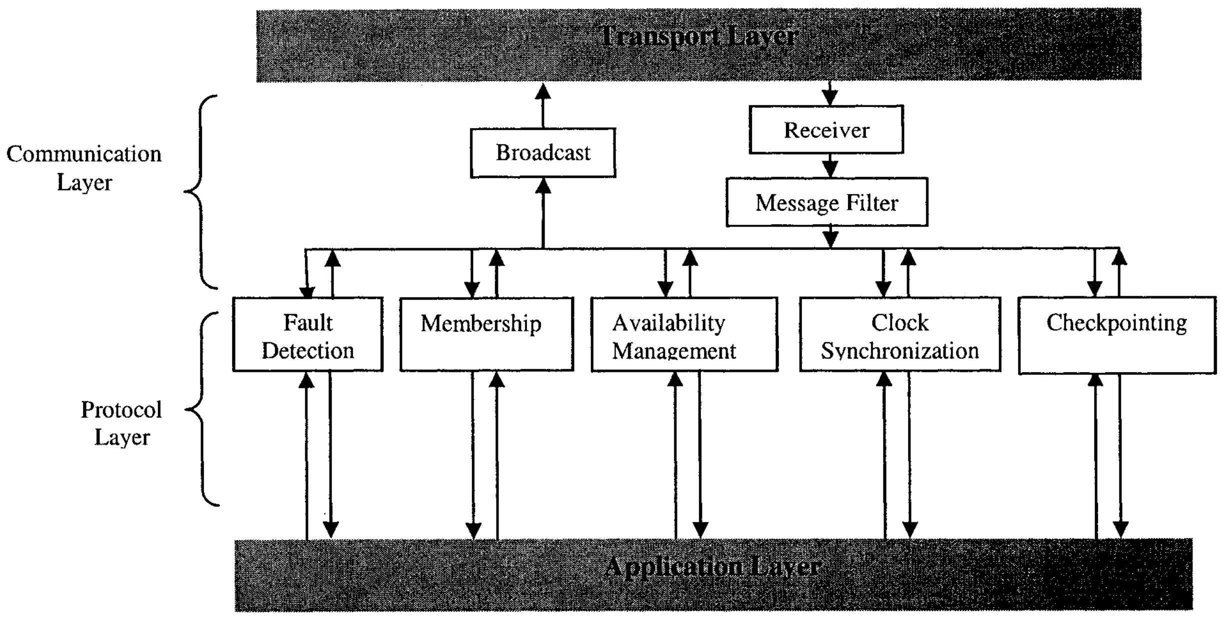


Figure 3.1: Overview of the JAMS framework

In each module, there is a main class named *module class*. In each protocol-module class, the related protocol is implemented as its inner class named *protocol-class*. We use the name of the related protocol to refer its protocol module, module class, protocol-class and corresponding objects. For example, for fault detection module - *FDmodule*, inside its module class - *FaultDetection*, the object *fdPtclObject* of protocol class *FaultDetectionProtocol* is capable of executing the fault detection protocol. An object of *FaultDetection* is expressed as *fdObject*. The three communication modules are *BCmodule*, *RCVmodule* and *MFmodule*.

3.2 In-depth View of Design

As mentioned, JAMS is fully object-oriented. Every component in JAMS is objectified without any concerns of it being a user interface or the protocol abstraction. A user interface is implemented in order to illustrate and test JAMS. Due to the environment restriction, the *CSmodule* is not implemented. We assume network administrator has configured the system to be synchronized. In the subsequent sections, we discuss the detailed design and the implementation for those components to collaborate together.

3.2.1 Design Patterns – Modular Design

Before discussing the system architecture, we first highlight issues related to protocol interactions. There are several approaches to protocol compositions. Most of them have applied inheritance as main tool to compose protocols, such as BAST, Arjuna. In our opinion, inheritance alone is not sufficient because it does not offer enough flexibility. For example, as seen in Figure 2.2 which depicts the inter-dependencies of building blocks, *AMmodule* depends on *CSmodule*, *BCmodule, MEMmodule*, and influences *CKPTmodule*. Except the *CSmodule* (since the clock can be maintained as global variable by the system), *AMmodule* must inherit three other classes in order to be able to run on its own. This type of inheritance is hard to be implemented, and typically does not allow code reuse, for example, if we use another *membership* protocol to replace current one, all other protocols on which *membership* is dependent have to be changed. Also, Java does not support multiple inheritance. In our design, protocol interaction has been dealt with by message passing mechanisms. The communication layer thus plays an important role of message exchange in our system.

In Figure 3.2, class diagram of main modules shows eight main classes involved in the system. We have designed the *BCmodule* class – *Broadcast* as an abstract class. All protocol-classes are subclasses of *Broadcast*. By inheritance, all protocol-classes can use the multicast service which is implemented in *Broadcast*. We will discuss this inheritance in Section 3.3.1. The *RCVmodule* and *MFmodule* can be viewed as one single module which is simply responsible for receiving datagram from the network and passing the data to the object of protocol-module class as required. The protocol-module classes are all extended from class *Thread*. A protocol-class is built in as an inner class of the module class. The module-class threads runs only for receiving messages from the communication layer. With such a design, an object of a protocol-class is capable of manipulating the data which is received by its module-class object and multicasting the message to its peers. In JAMS, the objects of communication-module classes and protocol-module classes work concurrently, and the objects of the protocol-classes also run at the same time. Accordingly, all objects in the system perform task independently and the interactions between them are easily handled. This independent design enables each module to be easily implemented and/or extended in future.

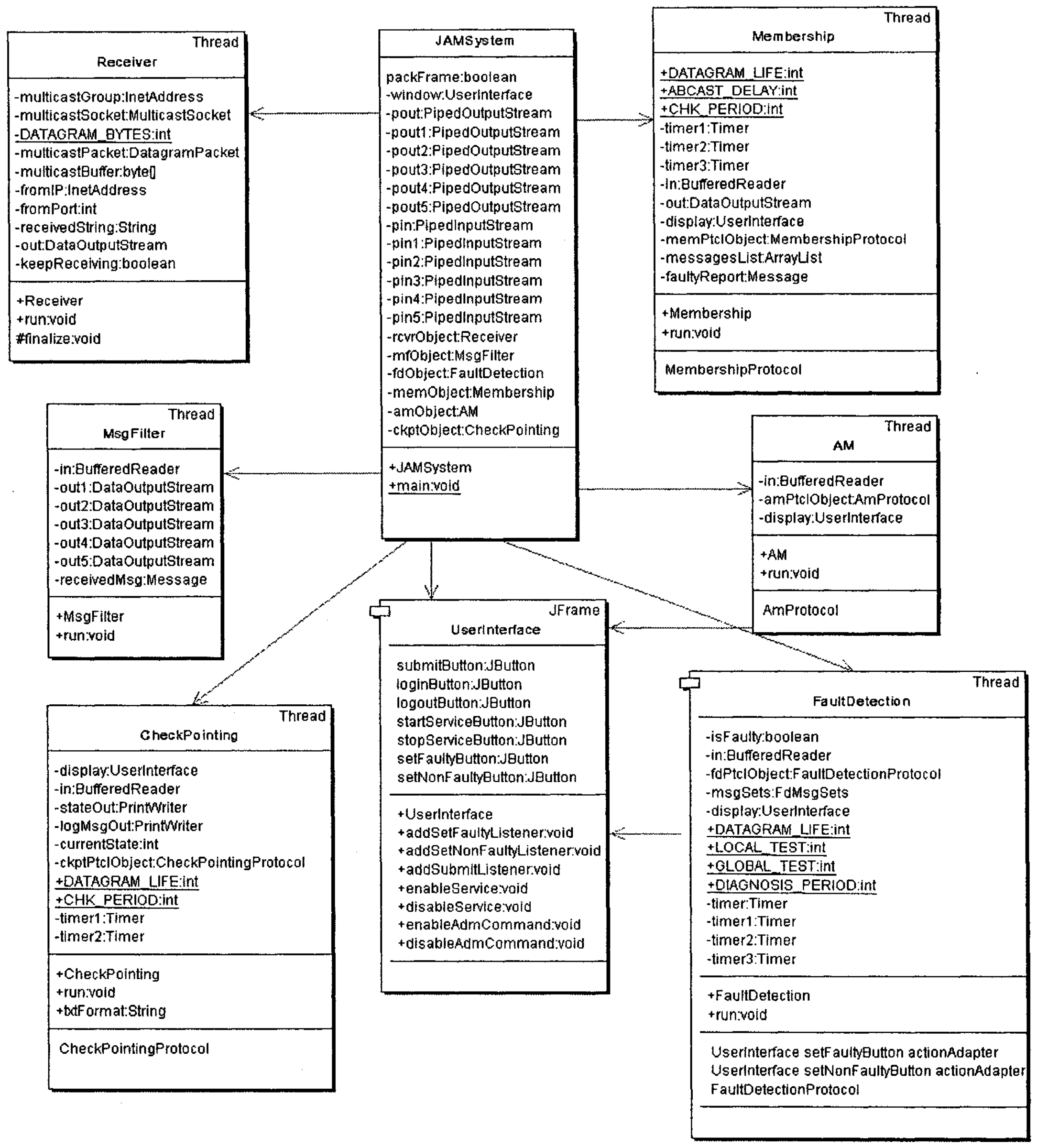


Figure 3.2: Classes diagram of main modules of JAMS

3.2.2 Protocol Module

Protocol module is defined as a set of classes that collaborate together to perform the desired operations as specified in different protocols' requirements. There are four implemented protocol-module classes in JAMS. Each module corresponds to a specific protocol-module class, and each protocol-module class includes one *protocol-class* and other supporting-protocol classes. The protocol-class is defined as the inner class of its module class, and practically it implements the behavior of the related protocol. An *inner class* is a nested class whose instance exists within an instance of its enclosing class and has direct access to the members of its enclosing instance.

The inner class design is a key feature of JAMS. The reasons of using inner class are as follows:

1. As a member of its enclosing class, an inner class has a special privilege [24]. It has unlimited access to its enclosing class's members, even if they are declared privates. For example, *fdPtclObject* is free to use all variables and methods of *fdObject*. This high access priority greatly facilitates programming of the protocol algorithm.
2. Inner class is hidden to other classes in the same package. This is a security mechanism which protects encapsulation of protocol-classes. In JAMS, by nesting protocol-class within a module-class, only the module-class can create the object of protocol-class.
3. It is easy to code the event-driven program by using inner class. There are two kinds of inner classes used in our system to implement event-driven task: one is being extended from class *TimerTask* which handles an event of time-elapse,

and the other one implements an *actionListener* interface to handle an event from *UserInterface.*

Each protocol-module class embodies a set of instances of classes. Their relationship is shown in Figure 3.3 with an example of the class diagram of *FDmodule.* A brief explanation is as follows:

FaultDetection is the main class of *FDmodule.* It creates all instances of other classes in a module. *FaultDetectionProtocol* is an inner class of *FaultDetection*, and its instance acts as a *fdPtclObject* that executes online diagnosis algorithm which we presented in Section 2.3.2. Being inherited from Broadcast, *FaultDetectionProtocol* can multi-broadcast *Message* to the group. Upon receiving *Message* by reading *InputStream, FaultDetection* sets timer to schedule a *TimerTask* which invokes some operations in *FaultDetectionProtocol*, and collects the diagnosis information in *FdMsgSets* in which a set of *Syndromes* reflects the local test information and a set of *Reports* reflects the global test information. *FaultDetectionProtocol* tests the faulty node based on the information in *FdMsgSets.* Through *UserInterface*, a user can see the results of protocol execution as well as log of events. Additionally, it allows tester to inject artificial fault to simulate node fault.

All other protocol modules have the same inner class design as *FDmodule,* while having their own supporting classes accommodating their own protocol. The detailed discussion about each module is given in Section 3.3.

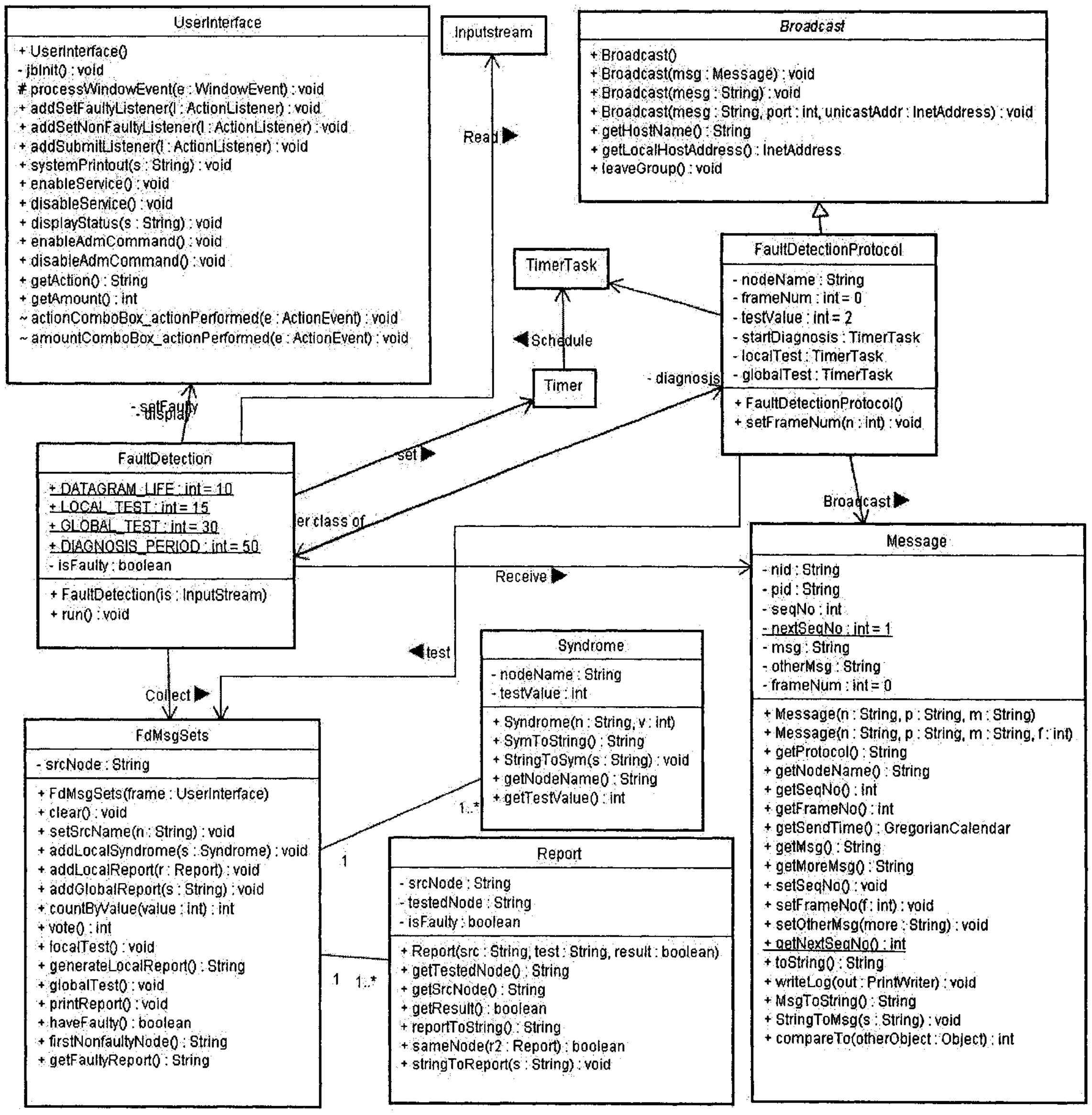

UserInterface
+ UserInterface()
- jbInit() : void
# processWindowEvent(e : WindowEvent) : void
+ addSetFaultyListener(l : ActionListener) : void
+ addSetNonFaultyListener(l : ActionListener) : void
+ addSubmitListener(l : ActionListener) : void
+ systemPrintout(s : String) : void
+ enableService() : void
+ disableService() : void
+ displayStatus(s : String) : void
+ enableAdmCommand() : void
+ disableAdmCommand() : void
+ getAction() : String
+ getAmount() : int
~ actionComboBox_actionPerformed(e : ActionEvent) : void
~ amountComboBox_actionPerformed(e : ActionEvent) : void
Inputstream
Read
Broadcast
+ Broadcast()
+ Broadcast(msg : Message) : void
+ Broadcast(mesg : String) : void
+ Broadcast(mesg : String, port : int, unicastAddr : InetAddress) : void
+ getHostName() : String
+ getLocalHostAddress() : InetAddress
+ leaveGroup() : void
TimerTask
Schedule
Timer
FaultDetectionProtocol
- nodeName : String
- frameNum : int = 0
- testValue : int = 2
- startDiagnosis : TimerTask
- localTest : TimerTask
- globalTest : TimerTask
+ FaultDetectionProtocol()
+ setFrameNum(n : int) : void
- diagnosis
- setFaulty
- display
FaultDetection
+ DATAGRAM_LIFE : int = 10
+ LOCAL_TEST : int = 15
+ GLOBAL_TEST : int = 30
+ DIAGNOSIS_PERIOD : int = 50
- isFaulty : boolean
+ FaultDetection(is : InputStream)
+ run() : void
set
er class of
Broadcast
Receive
Message
- nid : String
- pid : String
- seqNo : int
- nextSeqNo : int = 1
- msg : String
- otherMsg : String
- frameNum : int = 0
+ Message(n : String, p : String, m : String)
+ Message(n : String, p : String, m : String, f : int)
+ getProtocol() : String
+ getNodeName() : String
+ getSeqNo() : int
+ getFrameNo() : int
+ getSendTime() : GregorianCalendar
+ getMsg() : String
+ getMoreMsg() : String
+ setSeqNo() : void
+ setFrameNo(f : int) : void
+ setOtherMsg(more : String) : void
+ getNextSeqNo() : int
+ toString() : String
+ writeLog(out : PrintWriter) : void
+ MsgToString() : String
+ StringToMsg(s : String) : void
+ compareTo(otherObject : Object) : int
test
Collect
Syndrome
- nodeName : String
- testValue : int
+ Syndrome(n : String, v : int)
+ SymToString() : String
+ StringToSym(s : String) : void
+ getNodeName() : String
+ getTestValue() : int
FdMsgSets
- srcNode : String
+ FdMsgSets(frame : UserInterface)
+ clear() : void
+ setSrcName(n : String) : void
+ addLocalSyndrome(s : Syndrome) : void
+ addLocalReport(r : Report) : void
+ addGlobalReport(s : String) : void
+ countByValue(value : int) : int
+ vote() : int
+ localTest() : void
+ generateLocalReport() : String
+ globalTest() : void
+ printReport() : void
+ haveFaulty() : boolean
+ firstNonfaultyNode() : String
+ getFaultyReport() : String
1..*
1
1
1..*
Report
- srcNode : String
- testedNode : String
- isFaulty : boolean
+ Report(src : String, test : String, result : boolean)
+ getTestedNode() : String
+ getSrcNode() : String
+ getResult() : boolean
+ reportToString() : String
+ sameNode(r2 : Report) : boolean
+ stringToReport(s : String) : void

Figure 3.3: Class Diagram of Fault Detection Module

3.2.3 Communication

Communication mechanism is a key aspect of JAMS framework. It enables interaction between the running objects which can be a network node in the group, or any instances of classes running in one node. Communications between objects within a module can be easily achieved by an inner class. Communications between network nodes and between the modules are mainly discussed here. Figure 3.4 gives an overview of JAMS communications. Communications between network nodes are implemented by a multicast socket, and communications between the objects of module-classes are achieved by setting up pipes. The messages they exchange with one another are of the type of class *Message* which has an attribute to identify which protocol module the *Message* belongs to.

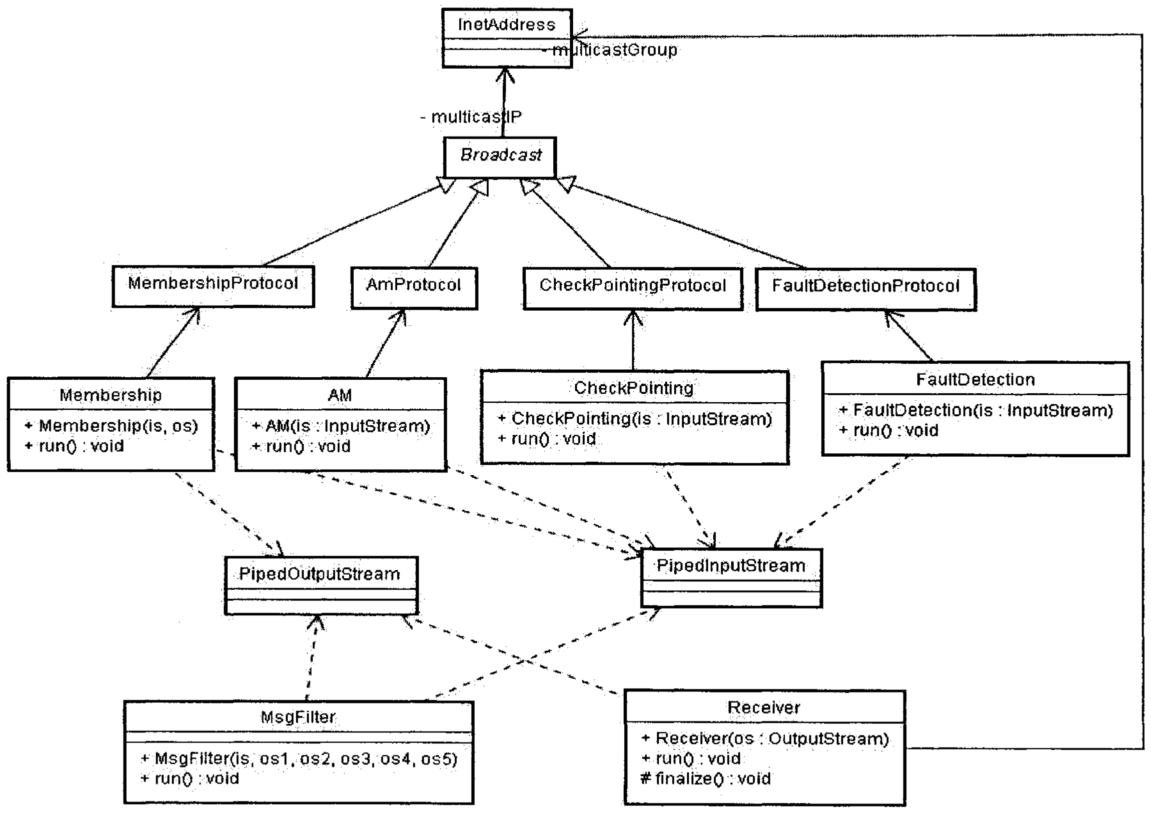


Figure 3.4: Communication in JAMS

Communications between Network Nodes

Relying on the transport layer, *BCmodule*, and *RCVmodule* provide group communication services. As shown in Figure 5, this service implements one-to-many multicast communications that enables any member to broadcast, at any time, a message *m* to all active members in the group over the distributed network.

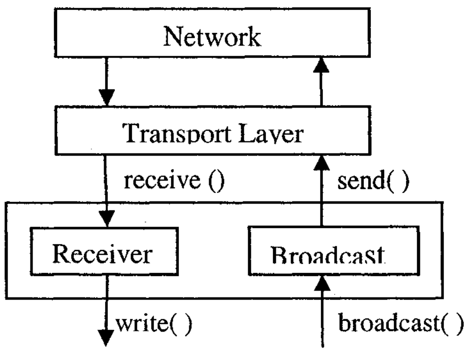


Figure 3.5: Group Communication Service

When JAMS starts up in a node, this node joins the multicast group by executing the following code in *BCmodule.*

```
//create a multicast socket with known IP and port.
multicastGroup = InetAddress.getByName("225.2.36.6");
multicastSocket = new MulticastSocket(1111);
multicastSocket.joinGroup(multicastGroup);
```

All members in this group have a multicast socket with the same IP address. Each node's *ReceiverObject* is running as a thread, listening to this multicast socket. Therefore any messages sent to this socket will be received by all of members in this multicast group.

BCmodule class is designed as an abstract class *Broadcast.* All protocol-classes are inherited from *Broadcast.* Whenever the communication between network nodes is necessary, an object of protocol-class broadcasts a message by calling function *broadcast( )* which is implemented in class *Broadcast*. Any message sent to this multicast socket will be received by the *rcvObject* of each member of the group, and then written to *mfObject*. Finally, the message will be delivered to the corresponding protocol module by *mfObject*. Next section will discuss how it is delivered properly.

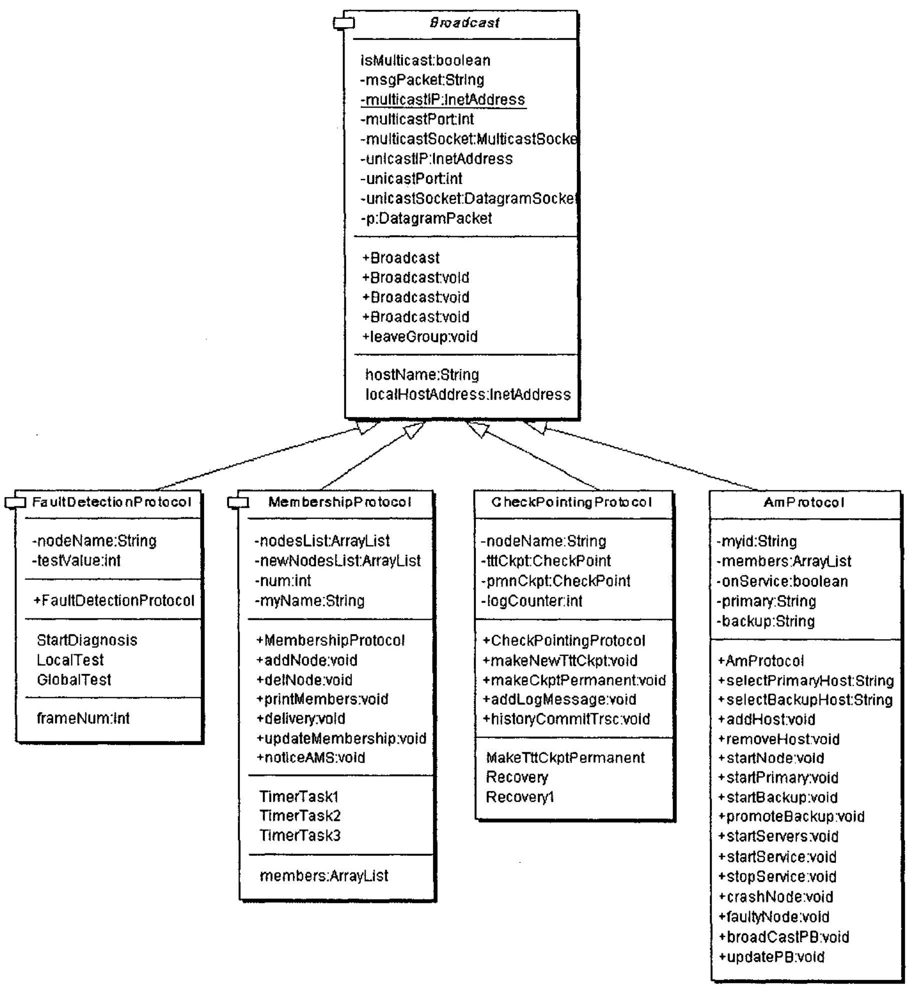


Figure 3.6: Inheritance in JAMS

Communication between Modules

Communication between modules is referred as the communication between threads at run time since each module-class object starts as a thread. Pipes are used to communicate between those module-class threads in JAMS. It enables threads to communicate in a portable way. By using pipes, a bunch of threads can be connected together, and thus synchronization between them is being taken care of. Figure 4 shows how pipes are used to connect threads in JAMS.

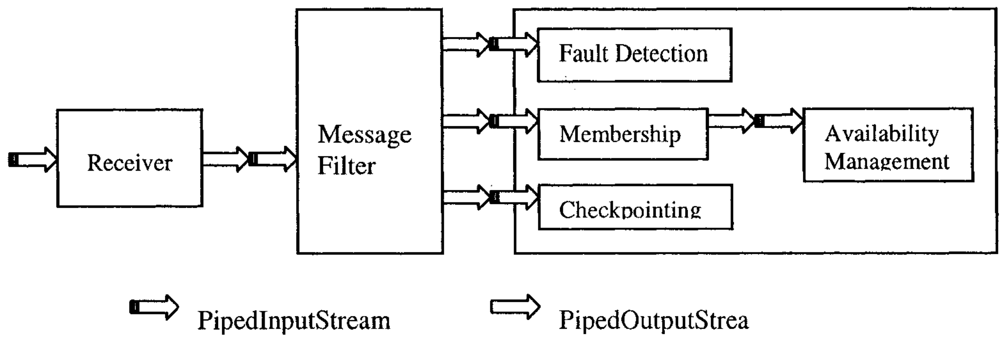


Figure 3.7: Communication between modules

The main objective to use pipe for communication between modules is to keep every thread simple at all time. For example, consider the communication channel from *RCVmodule* to *FDmodule.* The *rcvObject* is only responsible to listening for the incoming message all the time and transferring the received message to a piped-output stream connected with *mfObject.* Upon reading a message from the piped-input stream, the *mfObject* will simply state which protocol-module that the message belongs to, and then pass it to that specific module. The *fdObject* just needs to read the message from the piped-input stream without caring for the source of that message. Since *AMmodule* depends on *MEMmodule* directly, we set up a pipe between them, and thus whenever the *memObject* detects a node crash, it notifies *amObject* right away, without broadcasting the message to network.

The following fragment shows how to set up pipes and connect them with threads in order to construct a communication channel from *RCVmodule*, through *MFmodule*, *MEMmodule*, to *AMmodule*.

```
/*set up pipes*/
PipedOutputStream pout = new PipedOutputStream();
PipedInputStream pin = new PipedInputStream(pout);
PipedOutputStream pout1 = new PipedOutputStream();
PipedInputStream pin1 = new PipedInputStream(pout1);
PipedOutputStream pout2 = new PipedOutputStream();
PipedInputStream pin2 = new PipedInputStream(pout1);

/* construct threads */
Receiver receiver=new Receiver(pout);
MsgFilter filter = new MsgFilter(pin, pout1);
Membership mem = new Membership(pin1,pout2);
AM am=new AM(pin2);

/* start threads */
receiver.start();
filter.start();
mem.start();
am.start();
```

3.3 Details in Protocol Modules

This section describes the dynamic behavior among objects in each module with the help of sequence diagrams. A sequence diagram, on the horizontal axis, shows the life time of the object that it represents, while on the vertical axis, shows the sequence of creation or invocation of these objects. Sequence diagram clearly depicts the sequence of events, and allows the designer to specify the sequence of messages sent between objects in collaboration over time. It is an excellent tool for depicting concurrent operations in each module of JAMS.

3.3.1 Fault Detection Module

The sequence diagrams of *FDmodule* are shown in Figures 3.8, 3.9 and 3.10. The first diagram shown in Figure 3.8 captures the course of events that take place when JAMS is started. Class JAMSystem defines *main( )* method which is the entry to the system. Upon calling *main(),* object *jamsObject* is created. The objects of module-classes are created in succession, such as *mfObject* and *fdObject.* Creation of module-classes - *AM, Membership, CheckPointing* is omitted in this diagram. Objects in each module are produced subsequently. When an instance of *FaultDetectionProtocol*, *fdPtclObject,* is created, it broadcasts a "join" message to the *group*.

The second diagram shown in Figure 3.9 captures what happens when the *FaultDetection* thread receives a "join" message from other peers. The process of passing message through *MulticastSocket*, *Receiver, MsgFilter*, data *inputStream* and *outputStream* is not shown in the diagram. There are three *Timers* (*t1*, *t2*, *t3*) and three associated *TimerTask* (*startDiagnosis*, *localTest*, *globalTest*) involved in the process.

Timer *t1* schedules *startDiagnosis* – a start of a fault diagnosis algorithm. The *localTest* and *globalTest* of diagnosis algorithm are scheduled by *t2* and *t3* respectively. *fd* is continuously listening to inputStream. When a "join" message arrives, *t2*, *t3* are canceled and t1 reschedules *startDiagnosis*. The elapsed time of *A1* invokes *fdPtclObject* to run *startDiagnosis*. In this way, diagnoses of the group are synchronized. By calling *startDiagnosis, fdPtclObject* broadcasts a "diagnosis" message which includes its test value, and schedules a *localTest* and a *globalTest* at the same time in each node. All "diagnosis" messages are sent to the *group*. As noted in diagram, *A1* is a constant which represents datagram timeout in the network, and *A2* is a constant indicating diagnosis period.

The third sequence diagram in Figure 3.10 shows a complete diagnosis cycle in the case of a group of three nodes. Upon receiving a "diagnosis" message, *msgSets* converts the message to a syndrome and adds the syndrome to its syndrome list. The event of elapsed time *B* invokes *fdPtclObject* to run *locatTest* which executes *vote()* on the syndrome, generates a *localReport* to record which node is *GOOD* or *BAD*, and broadcasts this "report" to the *group*. Upon receiving this "report" message, *fd* saves it in msgSets' local report list. The event of passing time *C* invokes *fdPtclObject* to run *globalTest.* The final diagnosis result is reported as a "faultyReport" message and broadcasted to the group by a non-faulty node. This diagnosis cycle is repeated every *A2* period, and restarts if it receives a "join" message.

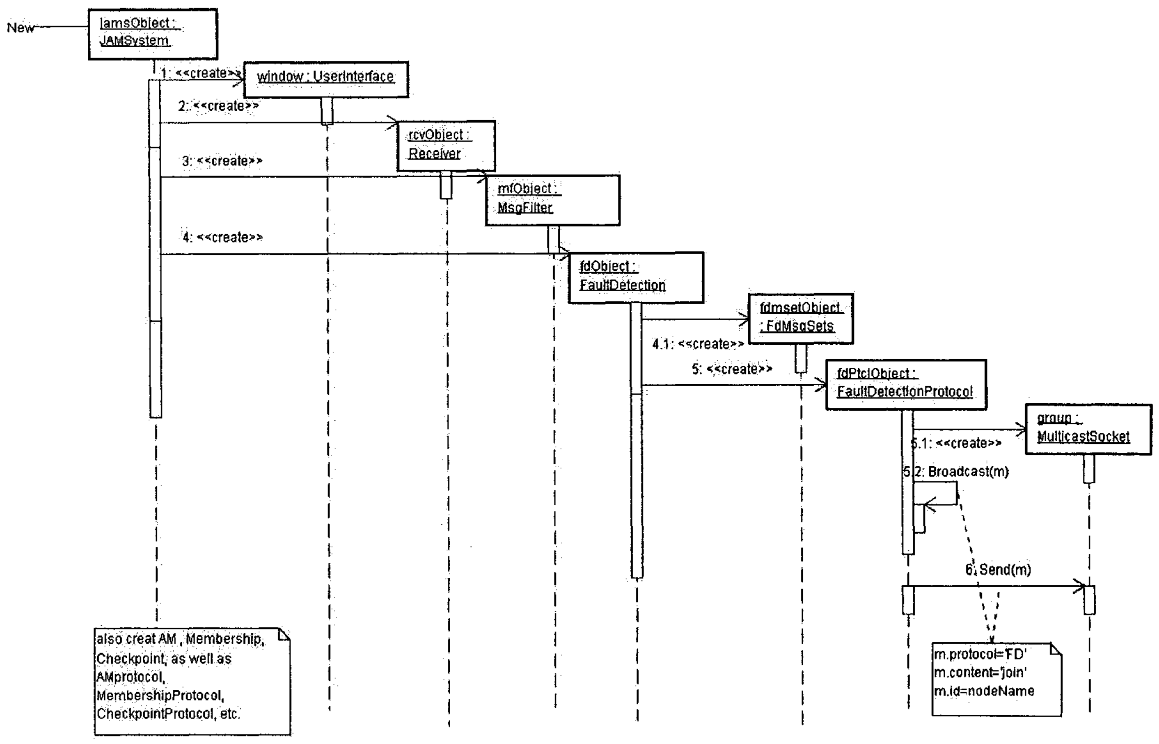


Figure 3.8: Sequence Diagram #1 of Fault Detection Module

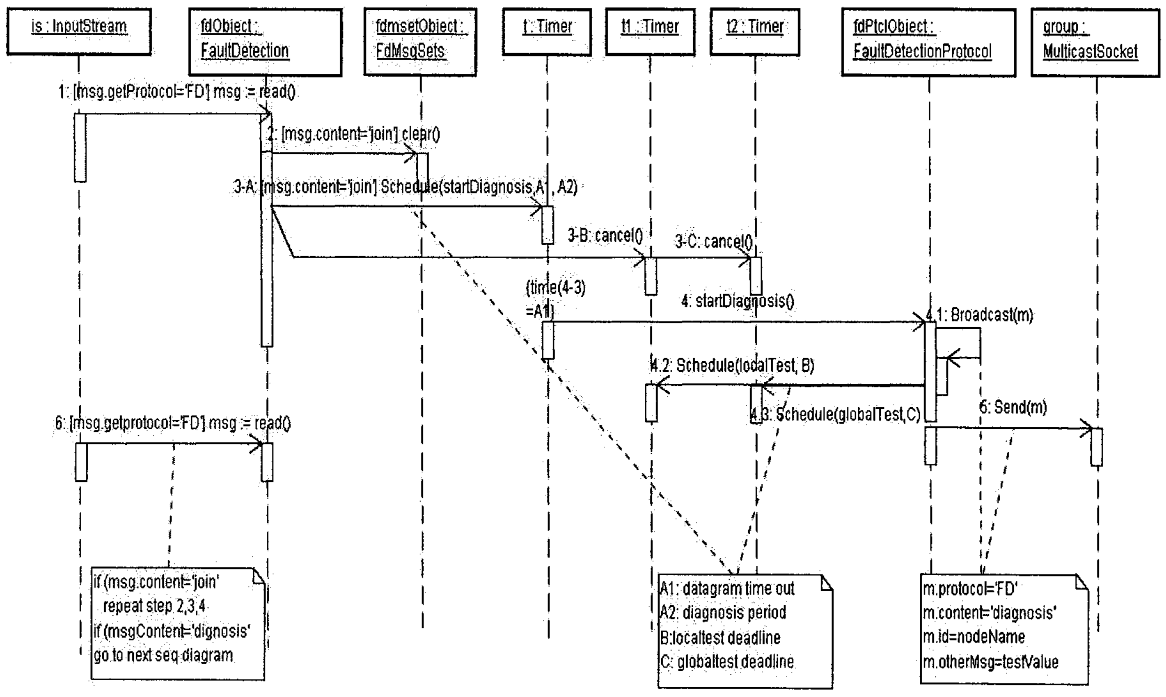

is : InputStream
fdObject : FaultDetection
fdmsetObject : FdMsgSets
t : Timer
t1 : Timer
t2 : Timer
fdPtclObject : FaultDetectionProtocol
group : MulticastSocket
1: [msg.getProtocol='FD'] msg := read()
2: [msg.content='join'] clear()
3-A: [msg.content='join'] Schedule(startDiagnosis,A1, A2)
3-B: cancel()
3-C: cancel()
{time(4-3) =A1}
4: startDiagnosis()
4.1: Broadcast(m)
4.2: Schedule(localTest, B)
4.3: Schedule(globalTest,C)
6: Send(m)
6: [msg.getprotocol='FD'] msg := read()
if (msg.content='join' repeat step 2,3,4
if (msgContent='dignosis' go to next seq diagram
A1: datagram time out
A2: diagnosis period
B:localtest deadline
C: globaltest deadline
m.protocol='FD'
m.content='diagnosis'
m.id=nodeName
m.otherMsg=testValue

Figure 3.9: Sequence Diagram #2 of Fault Detection Module

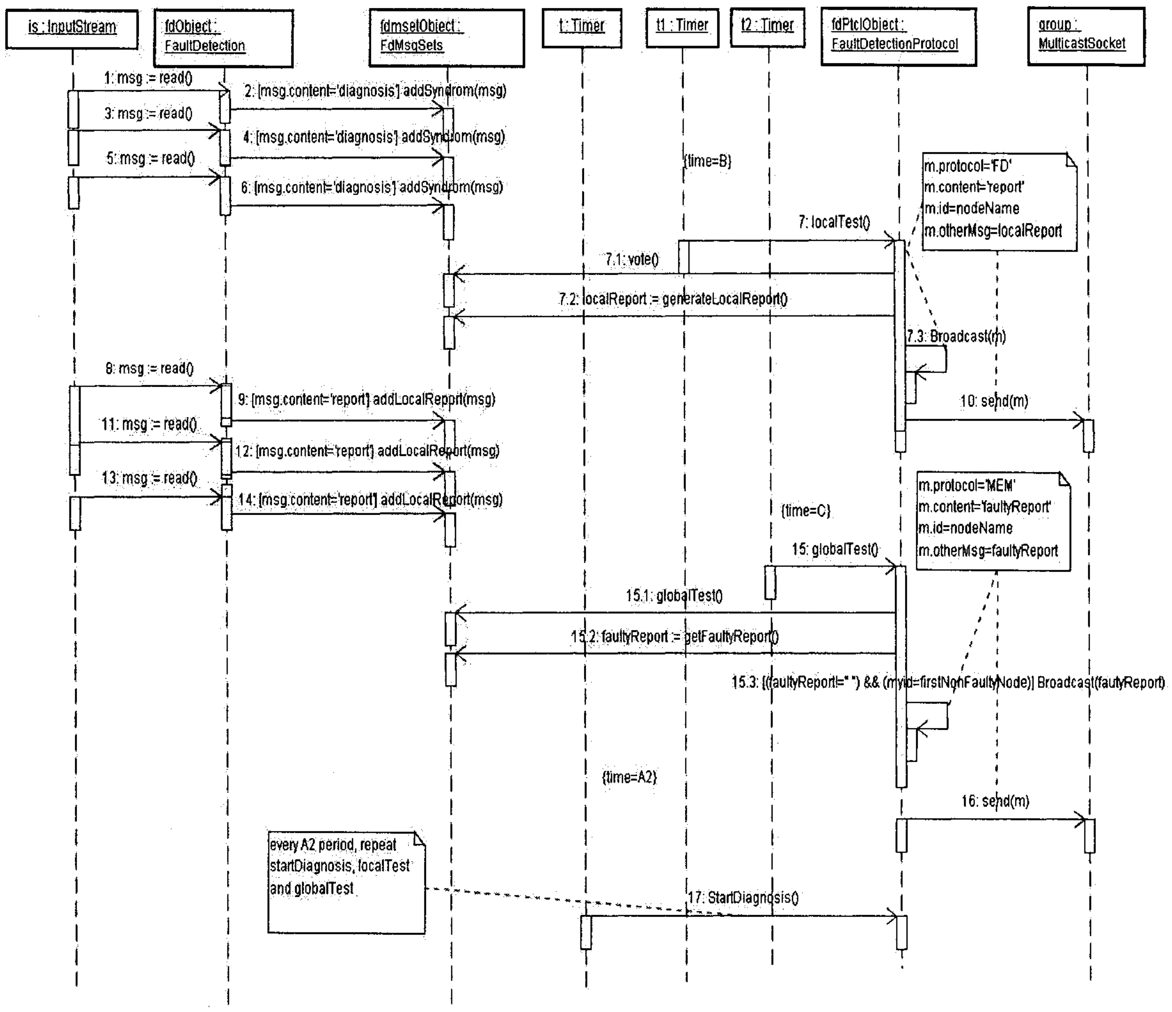

is : InputStream
fdObject : FaultDetection
fdmselObject : FdMsgSets
t : Timer
t1 : Timer
t2 : Timer
fdPtclObject : FaultDetectionProtocol
group : MulticastSocket
1: msg := read()
2: [msg.content='diagnosis'] addSyndrom(msg)
3: msg := read()
4: [msg.content='diagnosis'] addSyndrom(msg)
5: msg := read()
6: [msg.content='diagnosis'] addSyndrom(msg)
{time=B}
m.protocol='FD'
m.content='report'
m.id=nodeName
m.otherMsg=localReport
7: localTest()
7.1: vote()
7.2: localReport := generateLocalReport()
7.3: Broadcast(m)
8: msg := read()
9: [msg.content='report'] addLocalReport(msg)
10: send(m)
11: msg := read()
12: [msg.content='report'] addLocalReport(msg)
13: msg := read()
14: [msg.content='report'] addLocalReport(msg)
{time=C}
m.protocol='MEM'
m.content='faultyReport'
m.id=nodeName
m.otherMsg=faultyReport
15: globalTest()
15.1: globalTest()
15.2: faultyReport := getFaultyReport()
15.3: [(faultyReport!=" ") && (myid=firstNonFaultyNode)] Broadcast(fautyReport)
{time=A2}
16: send(m)
every A2 period, repeat startDiagnosis, localTest and globalTest
17: StartDiagnosis()

Figure 3.10: Sequence Diagram #3 of Fault Detection Module

3.3.2 Membership Module

The sequence of actions in *MEMmodule* is depicted in Figure 3.11, 3.12 and 3.13. The first diagram shown in Figure 3.11 captures what happens when *memObject* receives a "join" message from other peers. Figure 3.12 captures the activities and transitions during a membership checking period. The third sequence diagram in Figure 3.13 captures the invocations which take place when the *memObject* receives a *"faultyReport"* message from the group. There are three *Timer*s (*t1*, *t2*, *t3*) and three associated *TimerTask*s (*TimerTask1*, *TimerTask2*, *TimerTask3*) involved in *MEMmodule*.

TimerTask1 is scheduled to broadcast a "present" message for membership checking purpose. *TimerTask2* updates the membership according to the received "present" messages, and informs *AMmodule* of the membership update. In *TimerTask3* the "*faultyReport*" message is passed to *AMmodule*. This leads to the removal of the faulty node from the group. From *FDmodule* sequence diagrams, it is evident that receiving "join" message always leads to canceling of the current algorithm cycle and rescheduling of the next algorithm cycle.

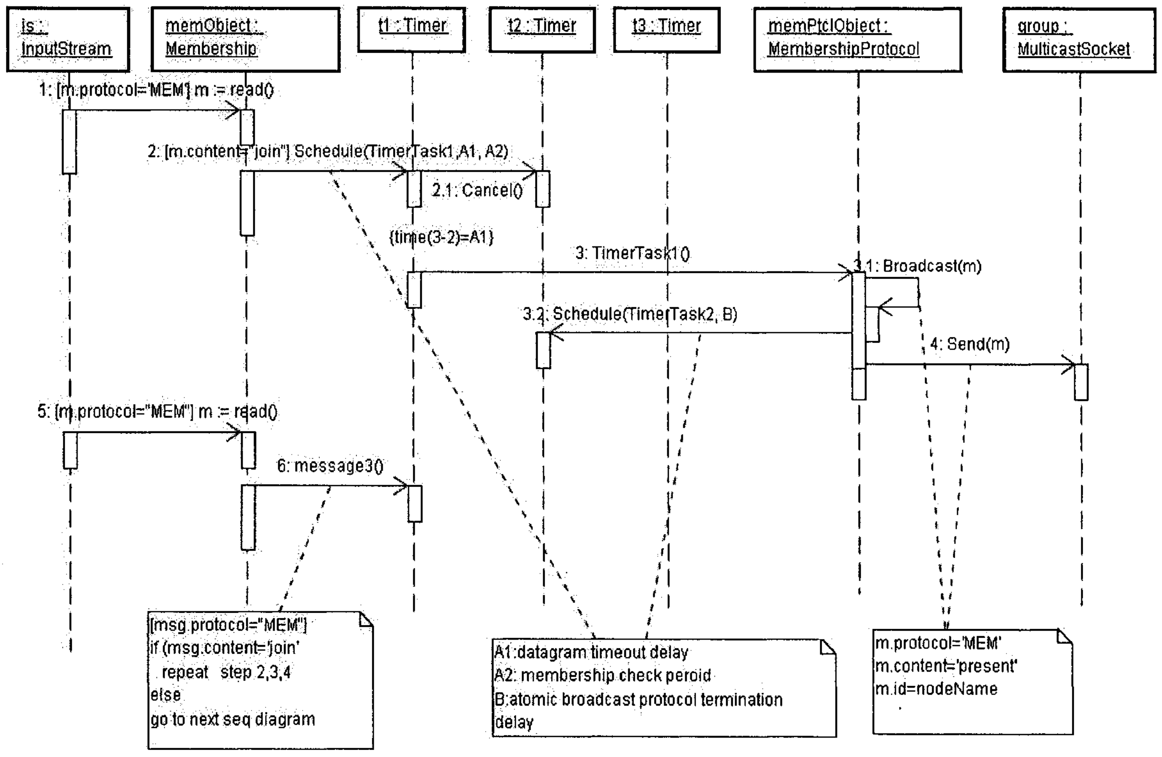


Figure 3.11: Sequence Diagram #1 of Membership Module

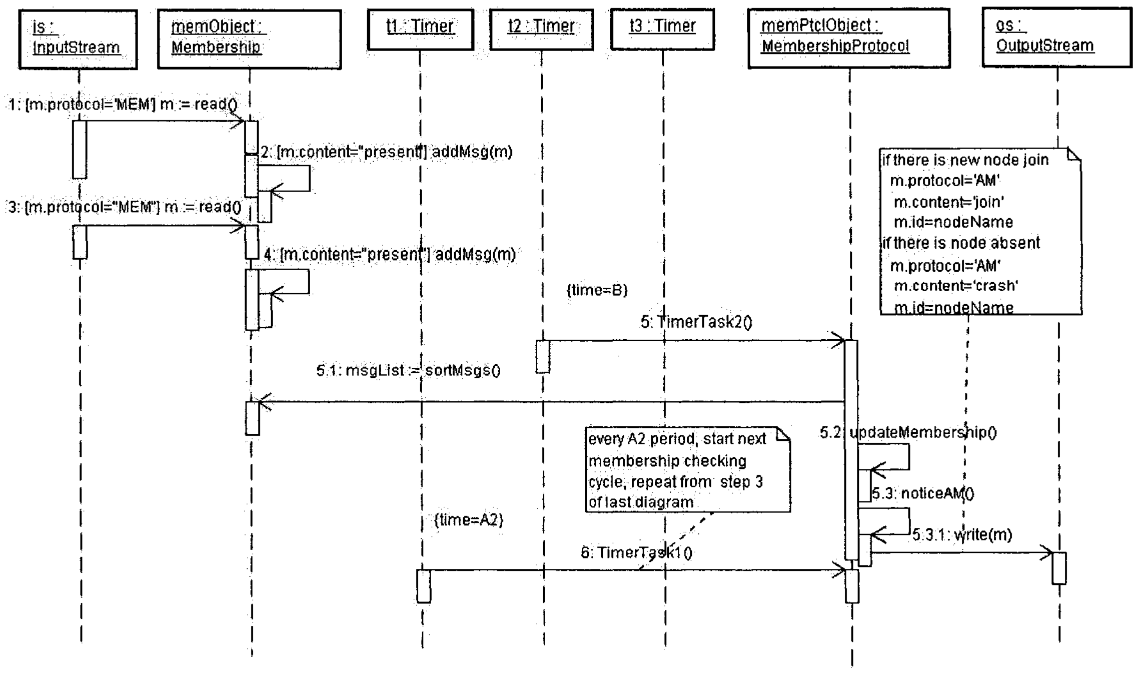

is : InputStream
memObject : Membership
t1 : Timer
t2 : Timer
t3 : Timer
memPtclObject : MembershipProtocol
os : OutputStream
1: [m.protocol='MEM'] m := read()
2: [m.content="present"] addMsg(m)
3: [m.protocol="MEM"] m := read()
4: [m.content="present"] addMsg(m)
if there is new node join m.protocol='AM' m.content='join' m.id=nodeName if there is node absent m.protocol='AM' m.content='crash' m.id=nodeName
{time=B}
5: TimerTask2()
5.1: msgList := sortMsgs()
every A2 period, start next membership checking cycle, repeat from step 3 of last diagram
5.2: updateMembership()
5.3: noticeAM()
5.3.1: write(m)
{time=A2}
6: TimerTask1()

Figure 3.12: Sequence Diagram #2 of Membership Module

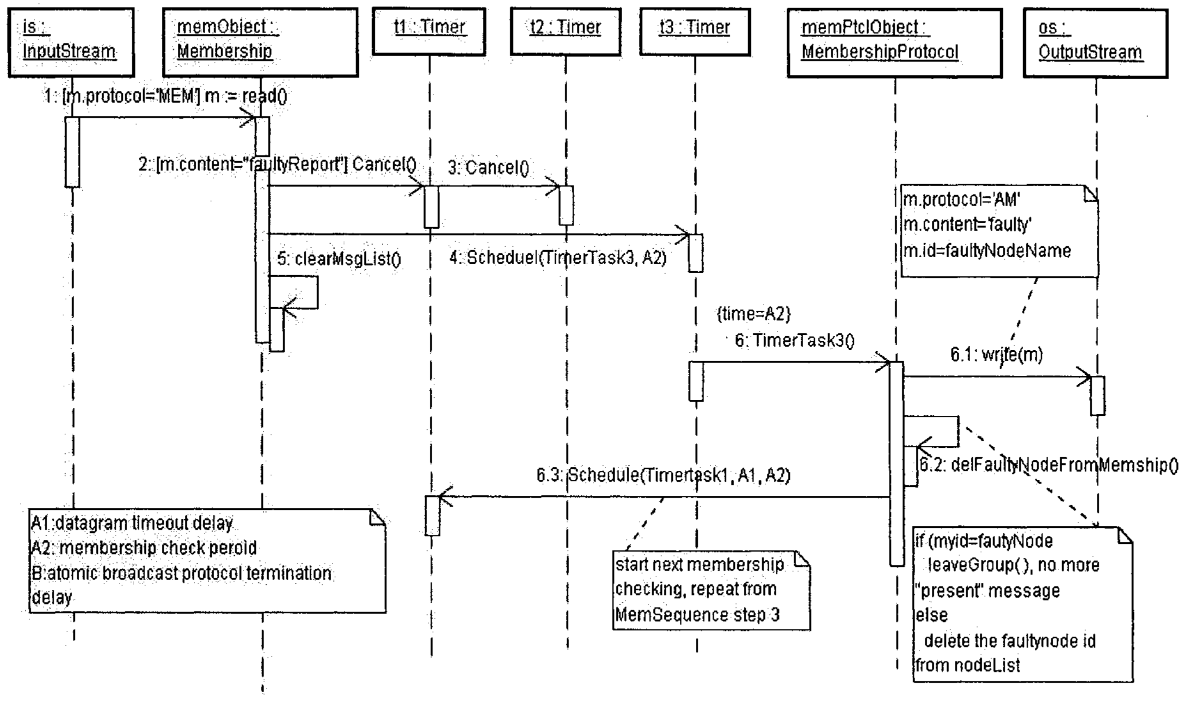

is : InputStream
memObject : Membership
t1 : Timer
t2 : Timer
t3 : Timer
memPtclObject : MembershipProtocol
os : OutputStream
1: [m.protocol='MEM'] m := read()
2: [m.content="faultyReport"] Cancel()
3: Cancel()
m.protocol='AM' m.content='faulty' m.id=faultyNodeName
5: clearMsgList()
4: Scheduel(TimerTask3, A2)
{time=A2}
6: TimerTask3()
6.1: write(m)
6.2: delFaultyNodeFromMemship()
6.3: Schedule(Timertask1, A1, A2)
A1: datagram timeout delay
A2: membership check peroid
B: atomic broadcast protocol termination delay
start next membership checking, repeat from MemSequence step 3
if (myid=fautyNode leaveGroup(), no more "present" message else delete the faultynode id from nodeList

Figure 3.13: Sequence Diagram #3 of Membership Module

3.3.3 Availability Management Module

The sequence of actions in *AMmodule* is depicted in Figures 3.14, 3.15 and 3.16. Referring the pseudo code of Availability Management Service in Section 2.2.5 helps better understand the sequence. Figure 3.14 captures what happens when a*mObject* receives a "join" or a "crash" notice from *memObject.* As shown in Figure 3.14, function *add-host()* and *crash-Node()* will be invoked by receiving "join" and "crash" message respectively. "p&b" message is broadcasted in order to indicate the newly joined node which is current *primary* and which is current *backup.* The second diagram shown in Figure 3.15 captures the course of events when an administrator presses "start service" button on *userInterface,* where a action listener is on watch. The third sequence diagram in Figure 3.16 captures the invocation that occurs when *memObject* receives a "p&b" or a "faulty" message from the group. Upon receiving "p&b" message, the node updates its *primary* and *backup.* Function *faulty-Node()* is invoked by receiving a "faulty" message.

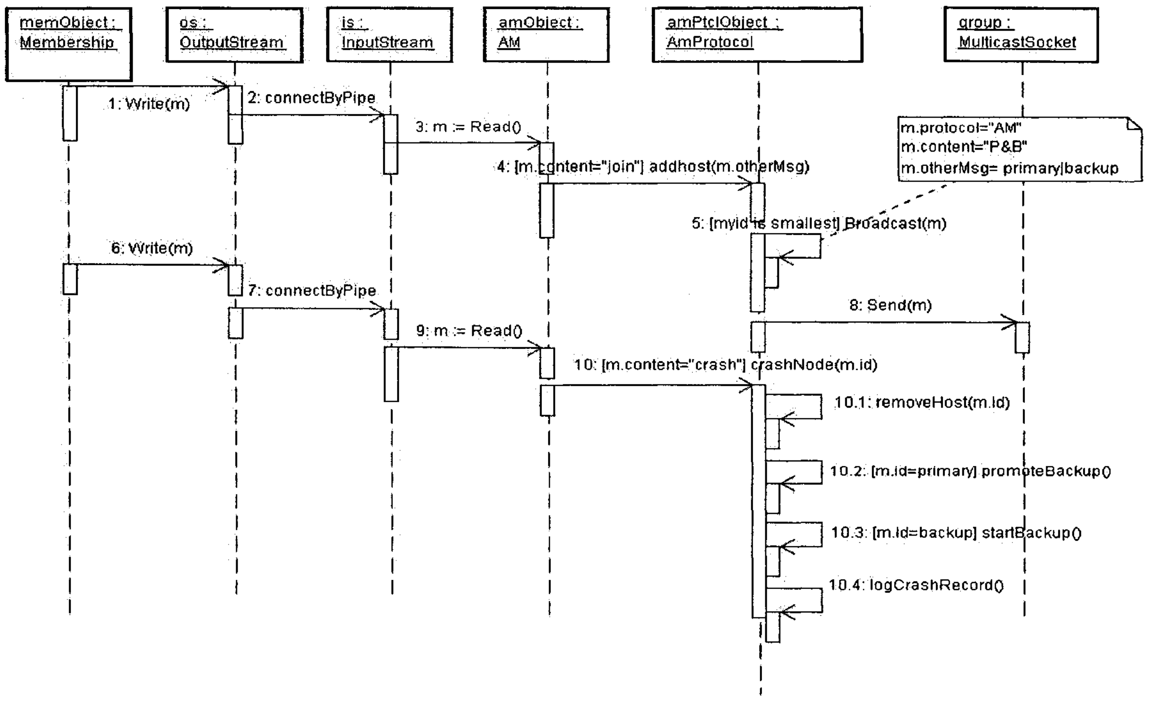


Figure 3.14: Sequence Diagram #1 of Availability Management Module

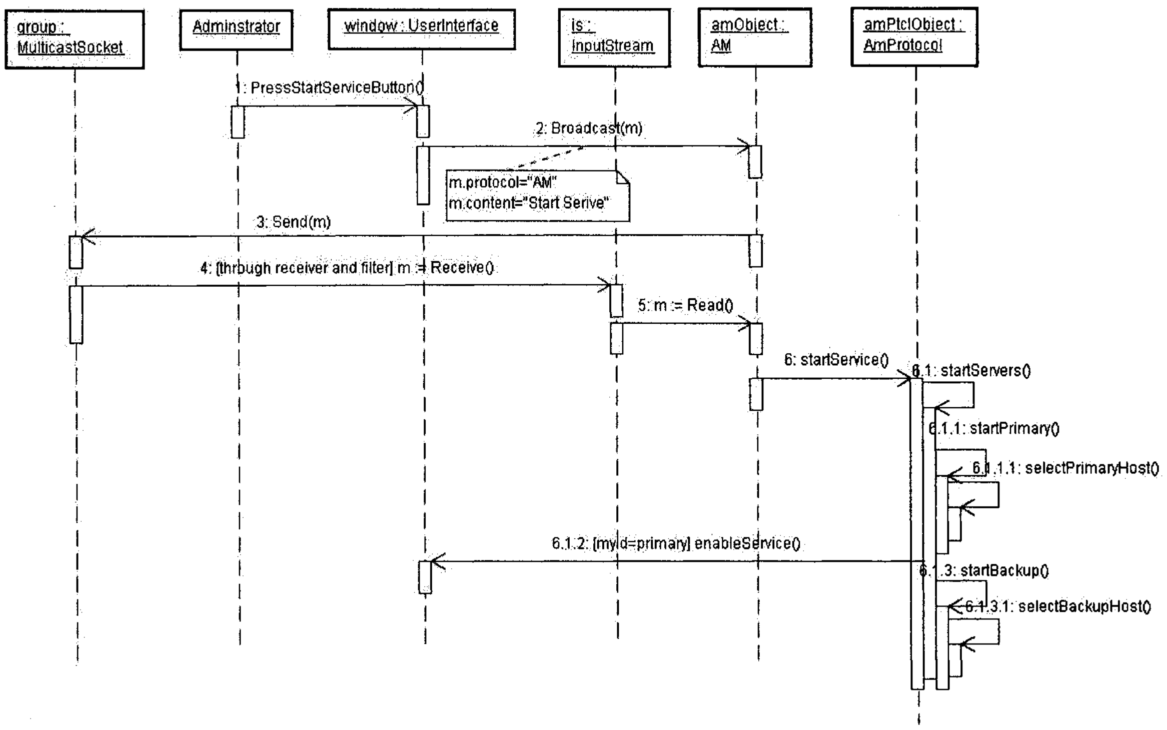


Figure 3.15: Sequence Diagram #2 of Availability Management Module

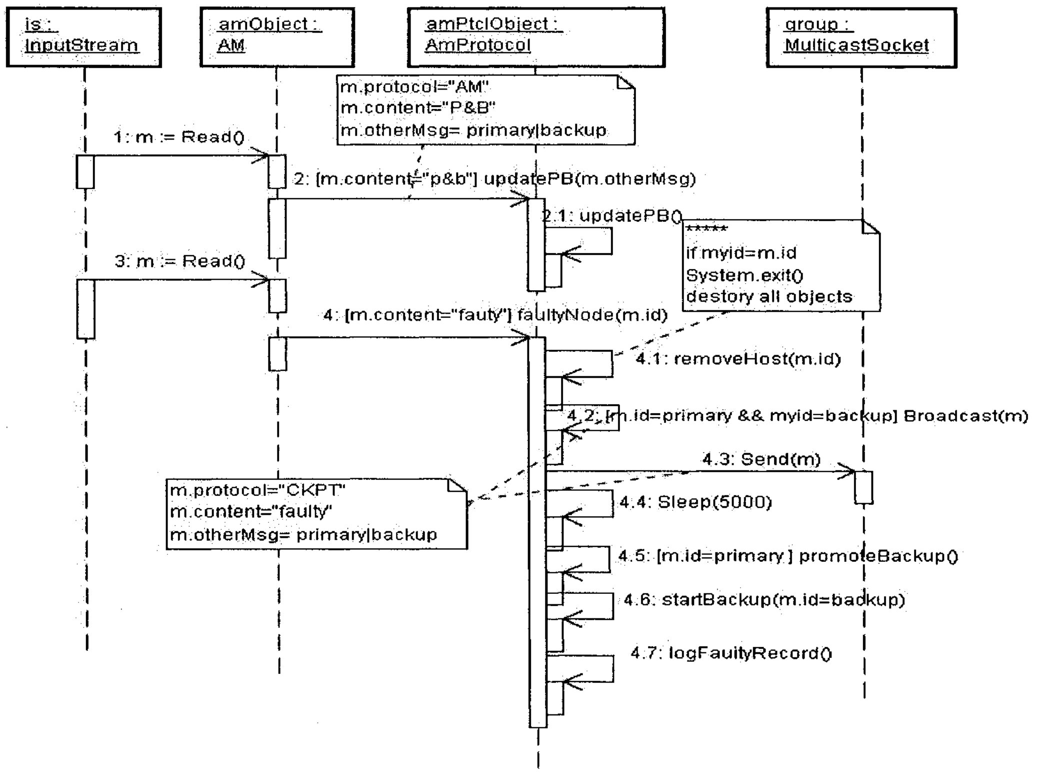
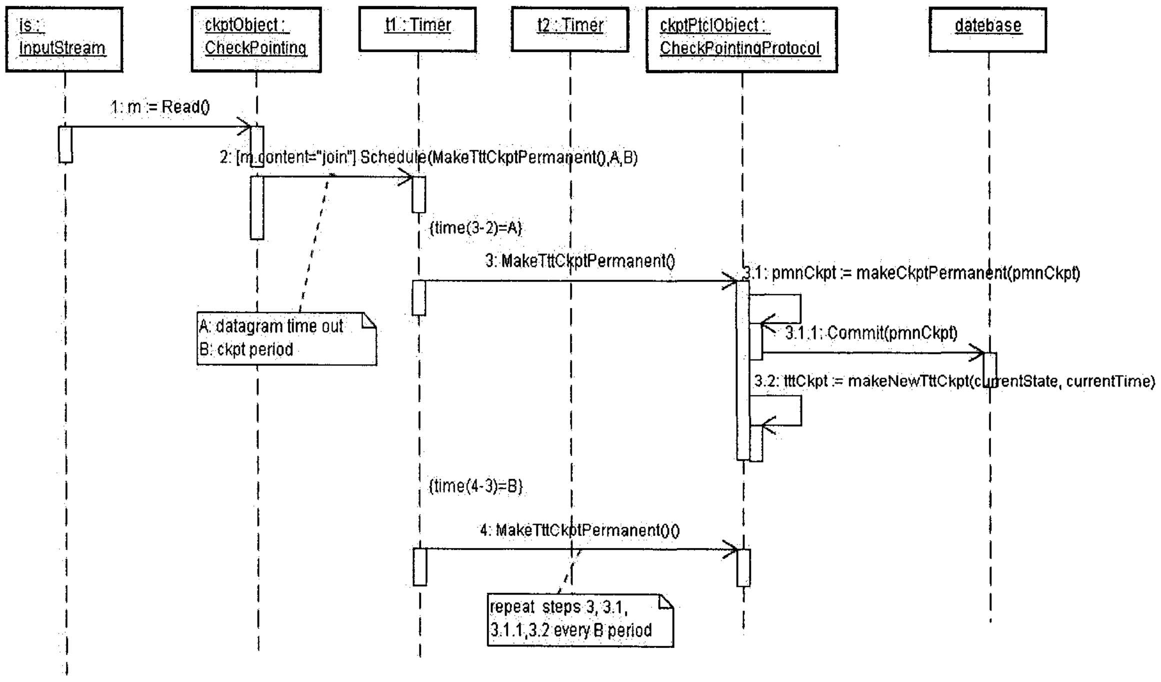


Figure 3.16: Sequence Diagram #3 of Availability Management Module

3.3.4 Checkpointing Module

The sequence of actions in *CKPTmodule* is depicted in Figures 3.17 and 3.18. Figure 3.17 captures what happens when ckpt*Object* receives a "join" message from other peers. The current checkpointing period is cancelled and a new checkpointing cycle is rescheduled. This diagram also demonstrates a complete process in a checkpointing period. The second diagram shown in Figure 3.18 captures the invocation of *Recovery ()* that occurs when *ckptObject* receives a "recovery" message from the *group.*



is : InputStream
ckptObject : CheckPointing
t1 : Timer
t2 : Timer
ckptPtclObject : CheckPointingProtocol
datebase
1: m := Read()
2: [m.content="join"] Schedule(MakeTttCkptPermanent(),A,B)
{time(3-2)=A}
3: MakeTttCkptPermanent()
3.1: pmnCkpt := makeCkptPermanent(pmnCkpt)
3.1.1: Commit(pmnCkpt)
3.2: tttCkpt := makeNewTttCkpt(currentState, currentTime)
A: datagram time out
B: ckpt period
{time(4-3)=B}
4: MakeTttCkptPermanent()()
repeat steps 3, 3.1, 3.1.1,3.2 every B period

Figure 3.17: Sequence Diagram #1 of Checkpointing Module

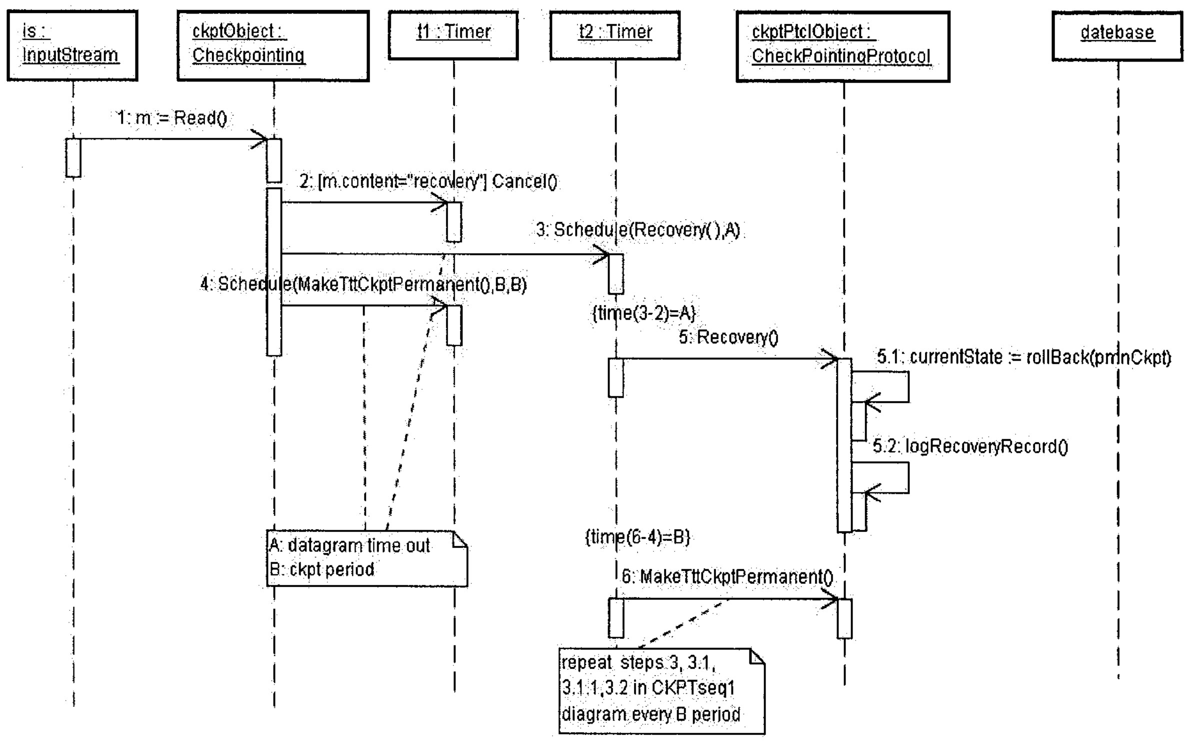

is : InputStream
ckptObject : Checkpointing
t1 : Timer
t2 : Timer
ckptPtclObject : CheckPointingProtocol
datebase
1: m := Read()
2: [m.content="recovery"] Cancel()
3: Schedule(Recovery( ),A)
4: Schedule(MakeTttCkptPermanent(),B,B)
{time(3-2)=A}
5: Recovery()
5.1: currentState := rollBack(prmnCkpt)
5.2: logRecoveryRecord()
A: datagram time out
B: ckpt period
{time(6-4)=B}
6: MakeTttCkptPermanent()
repeat steps 3, 3.1, 3.1.1,3.2 in CKPTseq1 diagram every B period

Figure 3.18: Sequence Diagram #2 of Checkpointing Module

After having described JAMS architecture, we next present a case study of banking service where we have utilized JAMS framework to develop such a highly available application in a modular way.

Chapter 4

Case Study of Applying JAMS to an Application

A fully operational JAMS has been implemented based on the modular design discussed in Chapter 3. This chapter discusses how the application programmer can use JAMS to build a fault-tolerant distributed application in a modular manner. We apply JAMS's implementation to an application service namely banking service to see how it ensures that application service remain continuously available to users despite occurrence of failures, maintenance and growth. Our presentation sacrifices description of details involved in a realistic application development for the purpose of making the underlying concepts understandable.

4.1 Building the Application

The application developer is responsible to develop a reliable distributed application by using ready-to-use protocol object libraries of JAMS. Figure 4.1 reflects where an application developer is going to place the desired application in the distributed platform provided by JAMS. The application service is designed as self-contained coded module, and located in the protocol layer in JAMS.

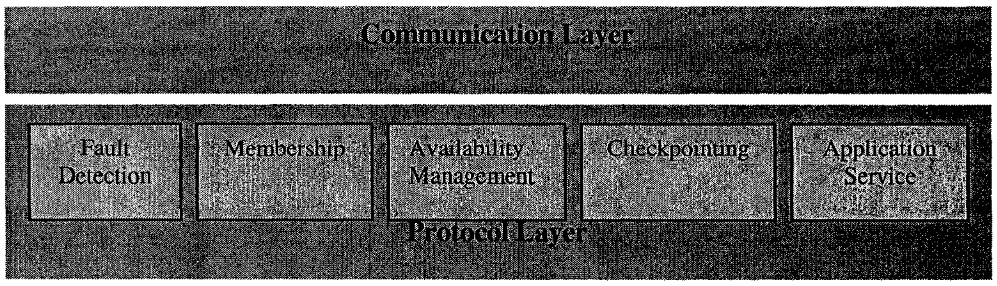


Figure 4.1: Building Application as a module in JAMS

A simple *banking* service has been developed for the purpose of testing. Two types of transactions are available to a client: *deposit* and *withdraw*. A data file is created to simulate a bank's *database*. Only the total balance of a bank account is saved in the data file. We name this bank application as *JAMS_BANK*.

4.2 Achieving Active Replication of JAMS_BANK

Considering a situation that *JAMS_BANK* objects are running in a network of three host nodes, we want the *banking* service to be available even when some host nodes crash or become faulty. In this network, *banking* service objects (*banking servers*) and JAMS objects are replicated on all three host nodes. Objects which are instances of a same protocol class on different nodes have exactly the same information. Only the *primary* of banking servers provides *banking* service to users. The *banking* service is a replicated object and it is manipulated as a group of replica banking servers $\{a, b, c\}$. Whenever *primary* server crashes or becomes faulty, the *backup* server is promoted to become the primary, and subsequently, continues to provide *banking* service without any prolonged disruption.

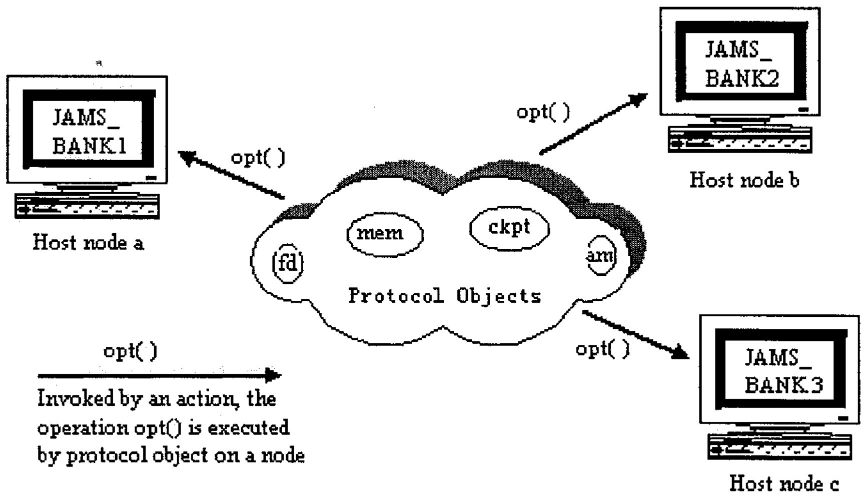


Figure 4.2: Active Replication with Protocol Objects

This high availability is ensured by the running protocol objects on all three nodes. Figure 4.2 depicts the active replication of protocol objects on all three nodes. Three kinds of actions can be initiated by three concurrent users: the human administrator, the adverse environment and the system-timer. The actions that an administrator can initiate include “start service”, “stop service”, “add host”, “remove host”, and the actions of an adverse environment include the “crash node” and “fault node”. The system-timer executes periodically the “increment current time” action on a node’s clock governing the rounds of the protocols’ execution. Any one action of these three kinds triggers an operation of the replicated protocol objects on all host nodes. For example, the elapsed time of a period of fault detection process invokes the execution of *startDiagnosis( )* on each *fdPtclObject* of all three nodes; identifying a fault node triggers the invocation of *faulty_Node( )* of *amPtclObject* on all three nodes. It ensures that the states of the replicate objects remain identical on each node.

The relationship between the banking server and other protocol module-class objects inside a node is illustrated in Figure 4.3. Through the interface, *AM* administrator can start or stop *bank* service; client can access the *bank* service; tester can set a node faulty over JAMS_BANK system; the protocols' execution information and the banking result can be displayed. *AM* module could know of any node join or crash/leave from *Membership* module and that of a node being declared faulty from *FaultDetection* module. The service state is checkpointed by *Checkpointing* protocol periodically. Upon receiving a *faultReport* message that a faulty node is detected on *primary* of the banking service, *AM* module sends a "recovery" command to *Checkpointing* module. Upon receiving "recovery" message, *Checkpointing* module discards current tentative checkpoint, and rolls back *banking* service state to a previous permanent state whose banking balance was already committed into the date file.

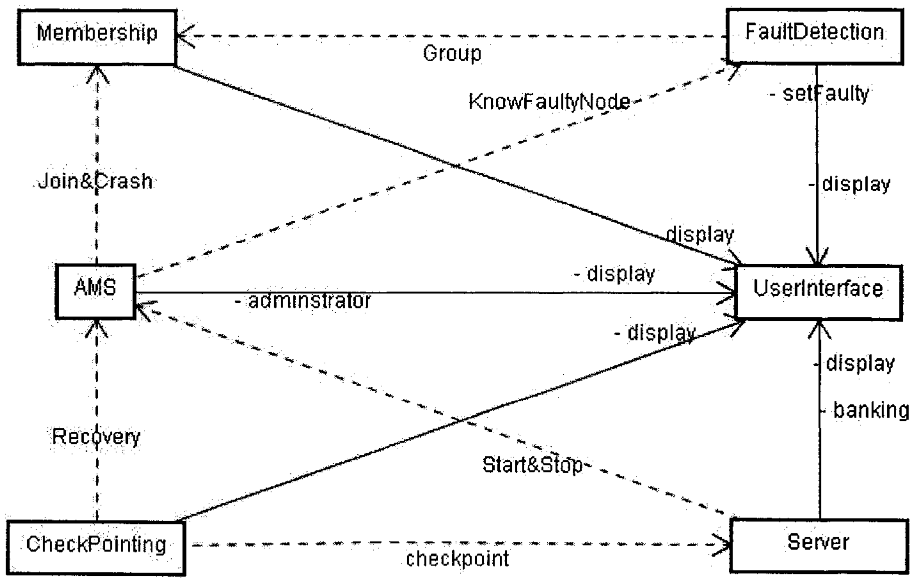
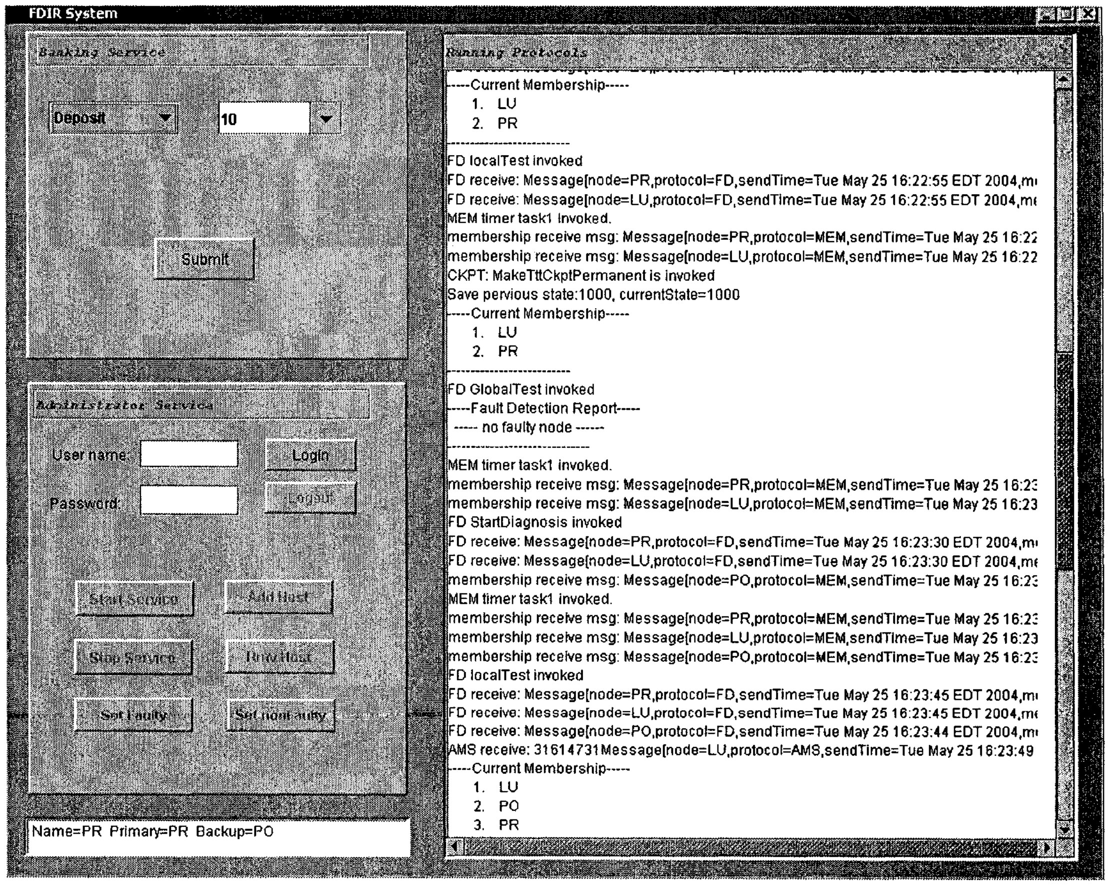


Figure 4.3: Relationship between the server and other objects in JAMS

Figure 4.4 shows the layout of the implemented user interface. The right region displays the protocol execution results, such as the fault report, membership update, etc. On left top corner, there is the banking service access area. The banking service is only accessible on the *primary* of banking servers. The node name, the current banking *primary* and *backup* are shown on the bottom status bar. The administration panel is between the status bar and the banking area. After administrator logs in, six command buttons are enabled for administrator. They are Start Service, Stop Service, Add Host, Remove Host, Set Faulty and Set *Non_faulty*.



```
-----Current Membership-----
    1.  LU
    2.  PR
------------------------
FD localTest invoked
FD receive: Message[node=PR,protocol=FD,sendTime=Tue May 25 16:22:55 EDT 2004,m
FD receive: Message[node=LU,protocol=FD,sendTime=Tue May 25 16:22:55 EDT 2004,m
MEM timer task1 invoked.
membership receive msg: Message[node=PR,protocol=MEM,sendTime=Tue May 25 16:22
membership receive msg: Message[node=LU,protocol=MEM,sendTime=Tue May 25 16:22
CKPT: MakeTttCkptPermanent is invoked
Save pervious state:1000, currentState=1000
-----Current Membership-----
    1.  LU
    2.  PR
------------------------
FD GlobalTest invoked
-----Fault Detection Report-----
 ----- no faulty node ------
----------------------------
MEM timer task1 invoked.
membership receive msg: Message[node=PR,protocol=MEM,sendTime=Tue May 25 16:23
membership receive msg: Message[node=LU,protocol=MEM,sendTime=Tue May 25 16:23
FD StartDiagnosis invoked
FD receive: Message[node=PR,protocol=FD,sendTime=Tue May 25 16:23:30 EDT 2004,m
FD receive: Message[node=LU,protocol=FD,sendTime=Tue May 25 16:23:30 EDT 2004,m
membership receive msg: Message[node=PO,protocol=MEM,sendTime=Tue May 25 16:23
MEM timer task1 invoked.
membership receive msg: Message[node=PR,protocol=MEM,sendTime=Tue May 25 16:23
membership receive msg: Message[node=LU,protocol=MEM,sendTime=Tue May 25 16:23
membership receive msg: Message[node=PO,protocol=MEM,sendTime=Tue May 25 16:23
FD localTest invoked
FD receive: Message[node=PR,protocol=FD,sendTime=Tue May 25 16:23:45 EDT 2004,m
FD receive: Message[node=LU,protocol=FD,sendTime=Tue May 25 16:23:45 EDT 2004,m
FD receive: Message[node=PO,protocol=FD,sendTime=Tue May 25 16:23:44 EDT 2004,m
AMS receive: 31614731Message[node=LU,protocol=AMS,sendTime=Tue May 25 16:23:49
-----Current Membership-----
    1.  LU
    2.  PO
    3.  PR
```

Name=PR Primary=PR Backup=PO

Figure 4.4: Outlook of the user interface

4.3 Testing of JAMS_BANK

Each module or protocol class of JAMS has been individually tested. In this section, we discuss the complete system testing with a real-world application to ensure the overall robustness and correctness of all aspects of dependability supported by JAMS. Testing is undertaken in a 100M LAN during a normal workday. Network nodes hosting *JAMS_BANK* are running either Windows XP or Red Hat Linux 9. Test cases that we used to validate the overall functioning of JAMS_BANK are partially listed in Table 3, where (a, b, c) represents three host nodes in the group.

Test Cases	**Initial states**	**Actions or events**	**Expected Results**	**Pass**

Case 1	Service is off
	(*a*, *b*, *c*)
		Administrator login
		Press "Start Service"
			Service → on
			primary =*a*
			backup=*b*
			(*a*, *b*, *c*)
				√

Case 2	primary=*a*
	backup=*b*
	(*a*, *b*, *c*)
		b crash	primary=*a*
			backup=*c*
			(*a*, *b*, *c*)
				√

Case 3	primary=*a*
	backup=*b*
	(*a*, *b*, *c*)
		a crash	primary=*b*
			backup=*c*
			(*a*, *b*, *c*)
				√

Case 4	primary=*a*
	backup=*b*
	(*a*, *b*, *c*)
		Administrator login on *b*
		Press "Set faulty"
			primary=*a*
			backup=*c*
			(*a*, *c*)
				√

Case 5	primary=*a*
	backup=*b*
	(*a*, *b*, *c*)
		Administrator login on *a*
		Press "Set faulty"
			primary=*b*
			backup=*c*
			(*b*, *c*)
				√

Case 6	primary=*b*
	backup=*c*
	(*b*, *c*)
	balance=$1000
		Deposit $100	primary=*b*
			backup=*c*
			(*b*, *c*)
			balance=$1100
				√

Case 7	primary=*b*
	backup=*c*
	(*b*, *c*)
	balance=$1100
		Deposit $100
		b crash
			primary=*c*
			(*c*)
			balance=$1200
				√

Case 8	primary=*b*
	backup=*c*
	(*b*, *c*)
	balance=$1100
		Administrator login on *b*
		Press "Set faulty"
		Deposit $100
			primary=*c*
			(*c*)
			balance=$1100
				√

Case 9	primary=*c* (*c*)	Administrator login Press "Stop Service"	Service → off (*c*)	√

Table 3: Test Cases of *JAMS_BANK*

We take test case 8 as an example to explain the testing procedures.

Initial States:

1. There are initially 2 host nodes, b and c.
2. Node *b* is the *primary* of the banking service and node *c* is the *backup*.
3. The balance of the bank account is $1100.

Actions:

1. Administrator logs in on *b*, so the six command buttons are enabled.
2. Administrator presses button "Set Faulty".
3. A user deposits $100 from *b*.

Expected Results:

1. Node *c* becomes the *primary* of the banking service.
2. Node *b* leaves the group, only node *c* stays in the group.
3. There is no *backup* now.
4. The transaction "Deposit $100" is not completed. The user is prompted to repeat this transaction. The balance of the bank account remains $1100.

As the test case has passed through, we could extract the following conclusions:

1. The faulty node is diagnosed and isolated accurately by Fault Detection/Diagnosis protocol.
2. The fault on *primary* leads to promotion of the *backup*, so that the service

remains available to users.

3. Any transaction on the faulty *primary* is rolled back, so that the service states remain consistent.

After systematic testing, it has been demonstrated that *JAMS_BANK* does provide autonomous availability of banking service despite arbitrary numbers of concurrent nodes removals, crashes, joins, and faults. The banking service always remains continuously available on the *primary* to users as long as there is at least one non-faulty node in the group.

There are two log files for recording *JAMS_BANK* operations from the start of a service to its completion. The first log file, *ams.log*, is used to record the change of service status, i.e., the change in status of a node being primary and backup. The second log file, *history.log*, logs all transactions and any recovery history. Below we show these two log files which are generated by a testing sequence comprising of test cases 1,3,6,8 and 9 as shown in Table 3.

```
********************************* ams.log *********************************
Bank Service is on at Thu Mar 11 15:02:47 EST 2004 with Primary=SR  Backup=SI .
Primary SR crash, SI becomes new primary, S becoms new backup at Thu Mar 11 15:04:22 EST 2004.
Remove faulty node SI, S becomes new primary at at Thu Mar 11 15:07:32 EST 2004.
Bank Service is off at Thu Mar 11 15:11:41 EST 2004.
********************** history.log *****************************
SI: Deposit$100  at Time=Thu Mar 11 15:05:14 EST 2004, Balance->1100
SI: Deposit$100  at Time=Thu Mar 11 15:06:47 EST 2004, Balance->1200
Recovery at Time=Thu Mar 11 15:07:34 EST 2004. Balance->1100
```

More tests will be necessary to refine our analysis and further optimize JAMS. In order to support a variety of application semantics, additional transactional protocols should be added to JAMS. For example, an atomic commitment protocol is needed when there may exist two or more primaries at one time. This issue is beyond the scope of this thesis.

Chapter 5

Discussion and Conclusions

Service availability has become an increasingly important factor in building distributed applications of late. In e-business, downtime may cost millions of dollars in addition to operating expenses. JAMS aims to help the application programmer build the application services that can remain continuously available to users despite the presence of failures, maintenance and growth. By implementing the distributed environment provided by JAMS protocol modules, one can be assured that the final product, in redundancy management aspect, is a consistent, fault tolerant, maintainable system with high availability. Please note that if we add protocols from the class of transactional process in JAMS, the product will be made more robust.

In this thesis, we have presented JAMS, an object-oriented library of reliable distributed protocol classes. In order to obtain overall global properties of JAMS, we have identified the building blocks that are appropriate to provide availability management service and FDIR service, and also highlighted their inter-dependencies. It has been shown how these complex protocols are integrated in JAMS by modularization. The complexity of inter-dependencies among the protocols was easily solved by using the modular design strategy and communication mechanisms. In JAMS, each protocol was designed as a module, structurally independent with each other, thus they can be used as the basic structuring components of distributed environment. We have introduced details of JAVA implementation for each module according to its sequence diagrams. The case study of the banking service has demonstrated that, by using JAMS, the reliable distributed application can be easily developed and its availability is greatly enhanced.

Other previous research works have already tackled the problem of distributed system design. Only BAST adopted the object-oriented approach of composing protocol objects, but its design has a couple of limitations. For example, it does not offer enough flexibility to optimal code reuse. The approach of modular composition, which is implemented in the JAMS, is the main contribution to the modeling of well-structured distributed systems. The JAMS is an object-oriented fault-tolerant system that can be characterized by the following features: (a) fully object-oriented; (b) modularization; (c) portable; (d) autonomous; (e) high performance; (f) self fault-tolerant.

5.1 Contributions

This work is dedicated to the field of application service availability by integrating Availability Management with FDIR method. The main contributions are summarized to be:

- An investigation of related work, which helps to identify the problems and to provide inspiration in the pursuit of optimal design of the system architecture.
- An identification of building block for each problem within classes of redundancy management protocols and extensions/modifications to some of existing protocols.
- The specification of functionalities of the building blocks and block inter-dependencies.
- Development of an *availability management system* by using modular composition.

- A modular design of JAMS system architecture, where each module is independently implemented.
- An implementation of JAMS in an object-oriented language.
- A case study of applying JAMS to a banking service to demonstrate real-world applicability.

5.2 Future Research Directions

Over our experience in designing and developing JAMS, we envision the following research directions:

- The availability management service would be designed to serve a set of application services instead of just one.
- The PP fault model would be extended to Hybrid Fault-Effect Model [7] which considers not only the benign and value faults, but also the arbitrary and link faults.
- The *synchronization policy* would be changed to *close synchronization* instead of *loose synchronization.* That is to say there may be many primary servers acting as peers to interpret the service requests in parallel and maintaining their internal states close to each other. In this case, additional transactional protocol would be needed.
- The harmonization of task scheduling of entire system would be an attractive research topic. For example, how to decide broadcast termination time in a specific network; when fault diagnosis should be scheduled after the membership is updated and when checkpointing

should be scheduled compared to diagnosis; how long the availability management should wait for checkpointing and recovery.

Moreover, JAMS *framework* is open not only for application programmers, but also for protocol programmers. As mentioned earlier, each suggested extension would lead to development of a new protocol. However, each of these new protocols could be implemented as an independent module and be added to the JAMS without having to construct a new structure.

Bibliography

1. F. Cristian, "Understanding Fault-Tolerant Distributed System," *Comm. of the ACM*, 34(2), pp. 57-78, Feb.1991.
2. F. Cristian, H. Aghili, R. Strong, "Approximate clock synchronization despite omission and performance failures and processor joins," *16th Int. Symp. On Fault-tolerant Computing,* 1986.
3. F. Cristian: "Probabilistic clock synchronization," *Distributed Computing*, Vol. 3, pp.146-158, 1989.
4. N. Suri, M. Hugue, C. Walter, "Synchronization Issues in Real-Time Systems," *Proc. Of IEEE*, 82(1), pp. 41-54, 1994.
5. H. Kopetz, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Tr. On Computers*, Vol. C-36, No.8, 1987.
6. F. Cristian, "Reaching Agreeement on Processor Group Membership in Synchronous Distributed Systems," *Distributed Computing*, Vol. 4, pp.175-187, 1991.
7. C. Walter, P. Lincoln, N. Suri, "Formally Verified On-Line Diagnosis," *Proc. Of IEEE*, pp.684-721, 1997.
8. C.J. Walter et al., "MAFT : A Multicomputer Architecture for Fault-Tolerance in Real-Time Control System," *Proc. RTSS*, 1985.
9. K. Shin, P. Ramanathan, "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System," *Proc. FTCS-17*, pp. 55-60, 1987.
10. M. Barborak, Malek, Dahbura, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, vol. 25, pp.171-220, 1993.
11. F. Cristian, H, Aghili, R. Strong, D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *15th Int. Symp. on Fault-tolerant Computing*, 1986.
12. O.Babaoglu, P.Stephenson, R.Drumond, "Reliable Broadcasts and Communication Models: Tradeoffs and Lower Bounds," *Distributed Computing*, No. 2, pp. 177-189, 1988.
13. A. Gopal, R. Strong, S. Toueg, F. Cristian, "Early-delivery Atomic Broadcast," Proc. 9th ACM Symp. *On principle of Distributed Computing*," pp. 297-309, 1990.
14. F. Cristian, "Automatic Reconfiguration in the Presence of Failures," *Software Engineering Journal*, pp.53-60, 1993.
15. P. Sinha, N. Suri, "Modular Composition of Redundancy Management Protocols in Distributed System: An outlook on Simplifying Protocol Level Formal Specification and Verification," *Proc. Of Intl. Conf. on Distributed Computing Systems-21,* pp.253-263, 2001.
16. P. Jalote, *Fault Tolerance in Distributed Systems. Prentice Hall*, 1994.
17. B. Garbinato, R.Guerraoui, "Flexible Protocol Composition in BAST," *ICDCS-18*, pp.22-29, 1998.

18. B. Garbinato, P. Felber, R. Guerraoui, "Modeling Protocols as Objects for Structuring Reliable Distributed Systems," In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'97)*, pp. 165-171, 1997.
19. B. Garbinato, P. Felber, R. Guerraoui, "Protocol Classes for Designing Reliable Distributed Enviroments" *ECOOP, LNCS* 1098, 1996.
20. B. Garbinato, R. Guerraoui, "BAST-A Framework for Reliable Distributed Computing," *Swiss Federal Institute of Technology,* 1997.
21. S.K. Shrivastava, G.N. Dixon, G.D. Parrington, "An Overview of Arjuna Distributed Programming System," *IEEE software*, 1991.
22. P. Jalote, *Fault-Tolerance in Distributed Systems*, Prentice Hall, 1994.
23. T. Pisello and B. Quirk, "How to quantify downtime," *Network World*, 2004.
24. C.S. Horstman, G. Cornell, *Core Java 2 Advanced Features*, Prentice Hall, 2002.
25. D. Nessett, "Massively Distributed Systems," *3Com Corporation,* 1999.
26. R. Koo, S. Toueg, "Checkpointing and Rollback Recovery for Distributed Systems," *IEEE Trans. On Software Engineering,* pp. 23-31, 1987.
27. S. Mishra, L.L. Peterson, R.D. Schlichting, "Experience with Modularity in Consul," *Software Practice and Experience*, 1993.
28. R. van Renesse, K. Birman, R. Friedman, M. Hayden, D. Karr. "A Framework for Protocol Composition in Horous," *Proceedings of the ACM Symposium on PODC*, 1995.