5-2000

Enhancing Partitionable Group Membership Service in Asynchronous Distrib Systems

Marwa Mohamed Mansour
The American University in Cairo AUC


ENHANCING PARTITIONABLE
GROUP MEMBERSHIP
SERVICE IN ASYNCHRONOUS
DISTRIBUTED SYSTEMS
MARWA MOHAMED MANSOUR
2000

The American University in Cairo

School of Sciences and Engineering

Enhancing Partitionable Group Membership Service in Asynchronous Distributed Systems

Computer Science Department

In partial fulfillment of the requirements for the degree of ***Master of Science***

by

Marwa Mohamed Mansour

B.Sc in Computer Science

Under the supervision of

Dr. Ahmed Sameh

May 2000

Thesis
2000/66

The American University in Cairo

Enhancing Partitionable Group Membership Service in Asynchronous Distributed Systems

A Thesis Submitted by
Marwa Mohamed Mansour

to Department of Computer Science
May 2000

In partial fulfillment of the requirements for
The degree of ***Master of Science***

has been approved by

Dr. Ahmed Sameh
Thesis Committee Chair / Adviser
Affiliation AUC

Dr. [redacted]
Thesis Committee Reader / examiner
Affiliation Professor, Compu

Dr. [redacted]
Thesis Committee Reader / examiner
Affiliation The American University

Dr. [redacted]
Thesis Committee Reader / examiner
Affiliation Chair of Systems & Computer Engineering Dept. Fac. of Eng. Ain Sha Uni

[redacted] May 28, 2000
Department Chair/ Program Director — Date

[redacted] June 3, 2000
Dean — Date

Dedication

I would like to dedicate this thesis to my parents for all the support, help and encouragement they provided me with. They were always patient with me throughout my years of education. I love them from the bottom of my heart and pray to God to bless them and protect them always and forever.

Acknowledgment

I would like to acknowledge first of all my supervisor Dr. Ahmed Sameh for all the help and support he provided me with. I would like to thank him very much for believing in me that I could do such work. A special thank to Alberto Montresor, University of Bologna, Italy, for his patience to answer all my questions, for his continuous help, and for providing me with the Jgroup Toolkit that I based all my work on. As of my friends, I would like to thank all my beloved friends whom without their help and support I wouldn't be what I am now. Thank you all !

Abstract

As the extent of distributed systems grows, group communication has proven to be a useful paradigm for developing reliable services in asynchronous distributed systems. The most important aspect in the group communication paradigm is the group membership maintenance. The major role of group membership maintenance is to track all membership changes that occur among processes and transforms these changes into a set of agreed upon views that represents the set of members participating in the current membership change.

In this thesis, we further discuss and enhance the design and implementation of a novel project that supports a group membership mechanism. This mechanism does not only handle process failures but also responds to network partitioning and re-merging, by allowing the installation of multiple concurrent views. Two new algorithms are proposed to enhance the membership service layer of the *Jgroup Reliable Distributed Object Model*. Theoretical and experimental results are obtained, showing that both algorithms have achieved their main goals, which are enhancing the membership service in Jgroup and still conforming to Partitionable Group Membership Service characteristics.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

The notion of Group Communication has proven to be a crucial and useful paradigm in the development of a reliable and highly available asynchronous distributed system applications [KSMD98]. This paradigm has four main mechanisms that are extremely vital to achieve a reliable group communication service: a group membership service, a multicasting mechanism for handling the multicasting of messages among the group members, a failure suspicion service for the system to be fault-tolerant and finally a message delivery semantic mechanism for handling message deliveries. The most important aspect in the group communication paradigm is the group membership maintenance.

The main role of the group membership service is to manage and maintain processes inside the group. It tracks all membership changes that occur among processes and transforms these changes into what is called a *view* that is considered the output of the membership service. A *view* consists of a list of group members who participate in the current membership change, and a unique identifier.

A lot of work has been done in the area of Group Communication, specifically on the group membership service. It must be noted that communication in distributed asynchronous systems can never be reliable without considering issues like process failures and network partitioning. Those aspects must be dealt with as normal events that must be confronted. Group partitioning is a major research subject in which communication is disabled among processes that are partitioned from the rest of the group. Earlier work supported primary partition model, where it considers its only survivor, the partition that has the maximum number of processes, whereas the rest of

the members in other minor partitions are considered as faulty processes. The group membership service in this model has a weak type membership, which requires only a single view that is agreed upon by all members of the group at any execution time. However, it has been found that the primary partition model is not suitable to support "partition-aware" applications in modern large-scale asynchronous distributed systems. Hence, attempts started to move towards systems that could support multiple group partitioning. The membership service now does not only support changes in the process level only, like simple join/leave requests, but in communication between processes in the same partition as well as in changes among processes in multiple partitions. Multiple concurrent views can now exist since processes in different partitions could have different perceptions to the membership changes.

In order for systems to support a reliable partitionable group membership service (PGMS) that could handle partitioning and re-merging of processes, a formal specification of partitionable group membership service must be stated carefully. Much theoretical and experimental work done in this area to define a set of formal requirements for the membership service have proven to be unsatisfactory, which made the partitionable group membership service still an open problem. Specifications of PGMS must be strong enough to preclude useless or trivial solutions as well as being weak enough to be implementable in asynchronous systems [DBM97]. The most recent work done in this area, is in the computer science department at the University of Bologna, Italy [BDM99]. This work aimed at building a set of formal specification of PGMS avoiding flaws and trivial solutions found in prior works. Their most recent project developed, is the ***Jgroup Reliable Distributed Object Model*** [Mont99]. This model is still an evolving prototype that is

designed and implemented to become an extension to Java distributed object model where all objects interact via Remote Method Invocation (RMI), in addition to its support to the new formal specifications of the Partitionable Group Membership Service (PGMS).

1.1 Problem Description

The problem tackled in this thesis is how to enhance the performance of the partitionable group membership service implemented in Jgroup model. How to make PGMS service reliable enough to execute in less execution time with less number of messages exchanged among group members. By which, views can be installed faster and yet still abiding to the formal specifications stated for the PGMS service.

1.2 Solution Highlights

In this thesis, we present two new algorithms implemented in Jgroup model to enhance on the existing group membership service. In Jgroup, the membership algorithm terminates in four rounds with servers exchanging five types of messages in two different states. The first proposed algorithm is designed to show that Jgroup algorithm, though still executing the membership changes in four rounds, can still perform efficiently but in less execution time, by avoiding the sending and receiving of messages that might be ignored at certain situations when the network becomes stable. Performance results show that the first algorithm has successfully reached its aim and Jgroup has executed the different membership changes that occur in the group in less execution time than the original algorithm.

The second proposed algorithm, on the other hand, divides Jgroup membership service algorithm into two parts, the fast agreement algorithm and the slow agreement algorithm. The fast agreement algorithm allows termination and

installation of a final agreed view among members of the group in a maximum of two rounds only. Still, making sure of synchronization among servers to avoid the delivery of obsolete views and still conforming to the PGMS specifications. If temporary inconsistencies occur between views perceived by processes in the same group, due to lack of symmetry or network instability, the algorithm has to run more rounds among the servers. Hence, the slow agreement algorithm is executed where the algorithm run in a maximum of four rounds as in the original algorithm ensuring final stability. Again, the performance of the Jgroup algorithm has been enhanced using the second proposed algorithm and the time needed to respond to a membership change has been reduced since the fast agreement algorithm is used in most cases when the network stabilizes.

1.3 Thesis Objectives

This thesis makes the following contributions:

- **The Jgroup Reliable Distributed Object Model:** A thorough study of Jgroup will be presented in chapter 4, to have a closer look at the Partitionable Group Membership Service, PGMS, developed in Jgroup along with its major properties. The main structure of the algorithm and the algorithm specifications and implementation is further discussed. Finally, this chapter concludes with the major strengths and weaknesses found.
- **A Client-Server Oriented Group Membership Algorithm in WANs:** This model was especially designed for WANs. The reason why I studied this system is because of its idea of decreasing the number of message exchange during the membership service. Although the algorithm is not as powerful as Jgroup, but if this idea was developed in Jgroup, the performance of the PGMS service will be further

enhanced. The Client-Server model specifications, structure and algorithm is discussed in chapter 5. The chapter concludes with a comparison table of Jgroup and the Client-Server models, specifying the major strengths and weaknesses of both models.

- **Enhanced Jgroup Algorithm I:** My contribution in this algorithm is to prove that the performance of Jgroup can be enhanced. Though the same execution rounds will be used as in the original algorithm, the number of message exchanged will be decreased depending on the type of network changes that occur during execution. The new membership service will be found to execute in faster execution time than the original Jgroup. A description and a pseudo code of this algorithm are presented in chapter 6.
- **Enhanced Jgroup Algorithm II:** From the *Client-Server* model discussed earlier, another novel algorithm for *Jgroup* is designed and discussed in chapter 7. This algorithm is able to combine the strengths found in *Jgroup* and the *Client-Server* model avoiding the major weaknesses in both. Coming up with a PGMS service model that executes in maximum of two rounds in most cases and in four rounds in cases where major network instability exist.
- **Proof of Correctness of the Enhanced Jgroup Algorithm II:** Appendix A presents a thorough proof for the second algorithm, *algorithm II*. This proof is divided into two major parts. First, I prove that the algorithm terminates in all cases. In addition, the new algorithm still conforms to the PGMS properties stated in the original Jgroup.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

- **Chapter 2** presents previous research on group communication protocols, semantics and mechanisms.
- **Chapter 3** presents different formal specifications of Partitionable Group Membership Service (PGMS) in systems that support group partitioning.
- **Chapter 4** presents the ***Jgroup Reliable Distributed Object Model*** structure, algorithm and its membership specifications.
- **Chapter 5** presents ***the Client-Server Oriented Group Membership Algorithm in WANs.***
- **Chapter 6** details the first enhanced Jgroup algorithm, ***AlgorithmI.*** It includes the algorithm description and performance results.
- **Chapter 7** presents the second enhanced Jgroup algorithm, ***AlgorithmII***. It details the algorithm specifications along with its performance results.
- **Chapter 8** concludes the thesis.
- **Appendix A** proves the correctness of *AlgorithmII* that it still conforms to the partitionable group membership specifications mentioned in Jgroup.
- **Appendix B** presents the code implementation details.

Chapter 2

2 The Group Communication Paradigm

2.1 Why Asynchronous Distributed Systems?

Many practical real-world distributed applications could easily be considered more realistic, if they are not based on assumptions like clock synchronization or maximum message delays. Such applications could neither have an exact synchronization of local clocks nor have setting of a global time. Hence, an asynchronous distributed system model could easily fit those types of applications.

Asynchronous distributed system model places no bounds on communication delays, such that, neither the relative speed of processes nor message delays could be bounded, due to several factors such as allowing for message non-blocking, transient failures or load variation on the communication environment.

However, an arbitrary drift in local clock rates could occur which may lead to a process lack of response. This leads to a major drawback in the asynchronous model since there would exist an impossibility to determine whether a communication failure or a process crash is the main cause of such a problem. Hence, a failure detection technique should exist to suspect and handle such failures.

2.2 Communication Paradigms in Distributed Systems

In distributed systems, communication paradigms could take different forms:

2.2.1 One-to-One Communication

This type of communication is based on the client-server model. A single server serves a single client, local to its machine, via a simple message passing, or remote from its machine, via RPC (Remote Procedure Calls.)

2.2.2 One-to-Many Communication

A single server could serve multiple clients using replication. Replicating data helps in improving system performance by ensuring that information is highly available in cases of failures or crashes.

2.2.3 Many-to-Many Communication

This type of communication introduces the group communication paradigm where a number of servers/processes form a group where they all co-operate to perform a certain required task for a set of clients. The group communication paradigm allows applications to become highly available and totally reliable [Maf95]. This makes the system behavior predictable despite of failures or processes joining or leaving the system dynamically.

2.3 Group Communication Components

In order to understand the notion of Group Communication paradigm, we must understand its main mechanisms. Four main mechanisms are extremely vital to achieve a reliable group communication service. It should support a *Group Membership* mechanism to detect changes among processes during operation, a *Multicasting* mechanism for sending messages to group members, a *Failure Suspicion*

mechanism for the system to become fault-tolerant by handling group splitting and re-merging, and finally, a mechanism for *Message Delivery Semantic.*

2.3.1 What is a Group?

A group consists of a collection of distributed members in which each member communicates with other members in the group sharing some global state and cooperating towards achieving a common goal. Processes that exist outside the group, view the group as a single unit or process.

2.3.2 Types of Group Members

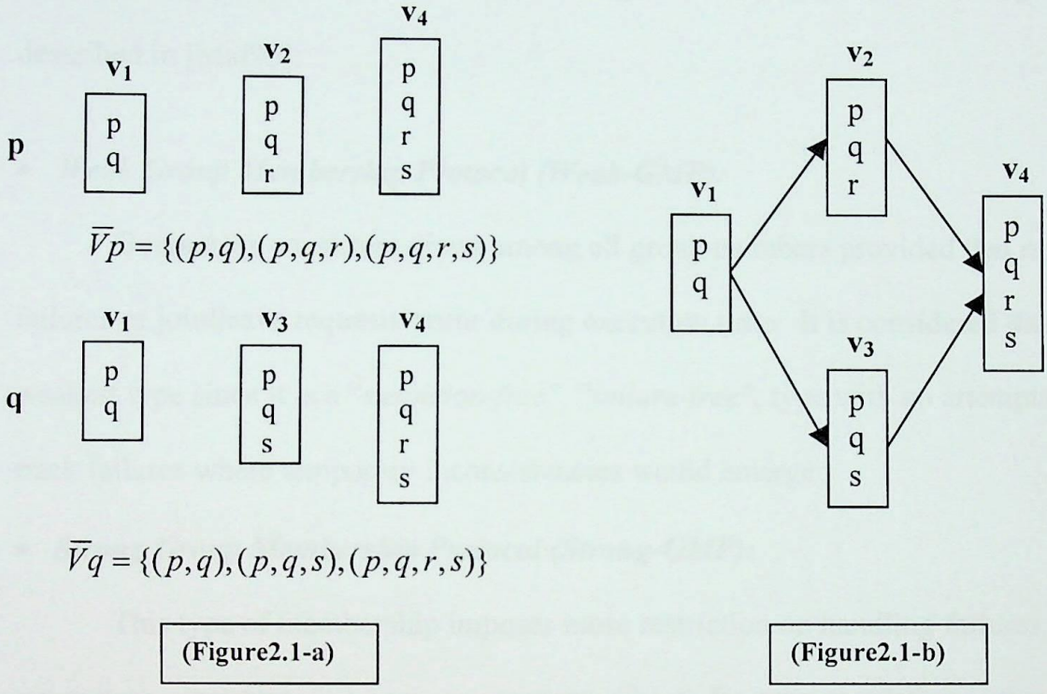
Members of the group are either processes or objects that dynamically leave or join the group. Processes that join the group are considered operational; i.e., they have not left the group. On the other hand, processes may dynamically leave the group, become unreachable due to failures, or are expelled from the group via the group communication paradigm.

2.3.3 What is Group Membership?

"*Group Membership manages the formation and maintenance of processes inside the group"* [RSB93]. It tracks all process changes, i.e., changes in the process itself when dynamic join/leave requests occur or changes among mutually reachable processes communicating within the group when a process or link failure occur. The group membership service transforms these changes into views that are agreed upon by all members of the group.

2.3.3.1 What is a View?

A view is defined as, "*A snap-shot of the group membership at a certain point in the execution of an application.*" [Maf95] Multiple views might exist in each process, which comprises a view list consisting of the process changes during an execution. Consider the following example:



(Figure2.1-a) (Figure2.1-b)

Figure 2.1: Group Membership Changes & Installation of Views

Figure 2.1 shows a change in a group behavior during a certain execution time. The group contains two processes, *p* & *q* reachable from each other. So *p* & *q* both install the same view $v_1^p = v_1^q = \{(p,q)\}$. Suppose after some time, processes, *r* & *s* join the group where *r* becomes reachable from *p* and *s* is reachable from *q*. Hence, *p* installs its new view change as $v_2 = \{(p,q,r)\}$, and *q* installs $v_3 = \{(p,q,s)\}$. Since the four processes belong to the same group, they should all become mutually reachable from each other. Hence, the group membership service will install a final agreed view, $v_4 = \{(p,q,r,s)\}$, consisting of all processes *p*, *q*, *r* and *s*. Both diagrams in

Figure 2.1 describe the same example. **Figure2.1-a** represents the changes that occur in each process, with a list, $\overline{V}$, of all views the process has installed. As of **Figure2.1-b**, it shows the whole group action and how the group membership service installs the different views.

2.3.3.2 Types of Group Membership

In recent years, different kinds of Group Membership protocols emerged, as described in [Maf95]:

- ***Weak Group Membership Protocol (Weak-GMP):***

  Guarantees consistent views among all group members provided that no failures or join/leave requests occur during execution time. It is considered the weakest type since it is a "*suspicion-free*", "*failure-free*", type with no attempts to track failures where temporary inconsistencies would emerge.

- ***Strong Group Membership Protocol (Strong-GMP):***

  This type of membership imposes more restriction on handling failures, and it guarantees consistent views among group members. Its main restriction is that it allows view changes among all processes to be seen in the same order, which is not always the case. This would affect the specification of *Strong-GMP*, which will further be investigated in chapter3.

- ***Hybrid Group Membership Protocol (Hybrid-GMP):***

  This protocol is a combination between the weak and the strong types of group membership, where it allows for a more relaxed level of consistency among group members. This will be discussed in more depth in chapter 3.

2.3.4 Multicasting & Ordering of Events

Processes inside the group receive messages via a single multicast communication operation. Multicast techniques have several types along with various ordering semantics. To name a few:

2.3.4.1 Types of Multicasting

- ***FIFO Multicast***

This multicast technique has the weakest ordering semantic, where it considers a First-In-First-Out multicast. It ensures that if a process multicasts a certain message m_1 before m_2, m_2 will not be received by any process unless m_1 is delivered first.

- ***Reliable Multicast***

It doesn't allow for spurious messages to be sent, and it guarantees that the message is sent to all group members provided that the sender doesn't fail.

- ***Totally Ordered Multicast***

Guarantees that messages sent in a certain order should be received with the same order by all group members.

- ***Causal Multicast***

It uses the happen before relationship described in [Maf95], it ensures that if a message depends on another message that happened before it, this message will not be received unless the one that occurred before, has already been received.

- ***Unreliable Multicast***

Multicasting could also be unreliable where messages may not be received by all processes in the group, or duplicate messages may exist.

2.3.5 Fault Tolerance

As previously noted, systems that follow the asynchronous distributed system model must devise a failure detection mechanism to handle any kind of failures that could occur, and to allow the system to become failure resilient and fault tolerant. Different kinds of failures exist in distributed systems, however, our discussion here in this thesis concentrates primarily on the types of failures that causes the system to partition.

2.3.5.1 Partition & No-Partition Assumption

It may be tempting to assume that communication in distributed asynchronous systems is totally reliable, where issues of process failures and partitions are avoided. However, as the geographic extent of distributed systems start to widen, more complex structures start to emerge where network failures and partitioning would normally occur or even imposed. Hence *"failure and Network partition should be seen as a normal not as an exceptional occurrence"* [ACT95]. In other words, services where partitioning occur should be confronted explicitly.

2.3.5.2 Types of Partition Failures

Failures that cause partitioning are of two main types:

- ***Virtual Partition***

This is a type of erroneous failure suspicion where one assumes that a failure occurred but in reality it is only an illusion. An illusion could occur at the process level stated as ***Virtual Process Failure,*** which assumes that the process has failed but actually it is only very slow. In addition, there may exist a ***Virtual Link Failure*** where a communication link is assumed faulty while it is only heavily loaded. In both cases of virtual partitioning we assume partitioning to occur.

- ***Actual Partition***

Actual partitioning indicates that a real failure exists. In other words, a process has either failed by crashing or a network link has actually failed or got disconnected. An *Actual Process Failure* is considered permanent, while an *Actual Link Failure* is only temporary since links could be re-paired and communication re-established. Hence actual link failures would cause group splitting but group members would re-join after repairs.

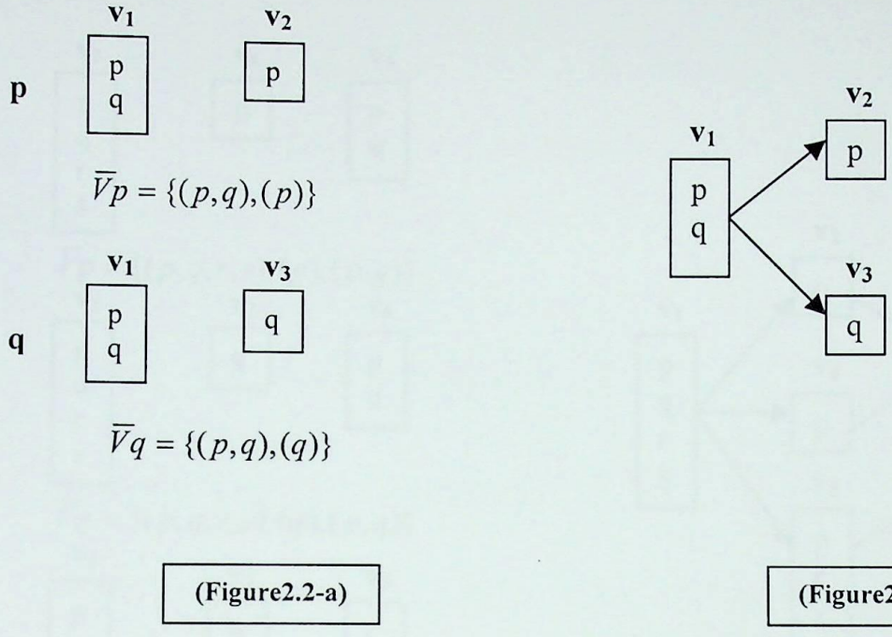
2.3.6 Group Partitioning & Re-Merging

2.3.6.1 Group Partitioning

Group splitting disables communication among processes that partitioned from the rest of the group members. Partitioning could either be a clean partitioning or a non-clean partitioning [BDGB95].

- ***Clean Partitioning***

Clean partitioning, also called *Strong-Partial*, occurs when the set of processes that have partitioned from the actual group are mutually unreachable, i.e., the intersection between them is ϕ. Hence, concurrent views are allowed to exist. Consider the following example shown in **Figure 2.2**. Suppose we have a group consisting of two processes, p & q that are mutually reachable. Both processes install an initial view $v_1 = \{(p,q)\}$. However, at a certain point in the execution time, p becomes unreachable from q (a partitioning occurred). So, p installs a different view containing only itself, $v_2 = \{(p)\}$, as well as q, which excludes p from its view and installs view $v_3 = \{(q)\}$.



(Figure2.2-a) (Figure2.2-b)

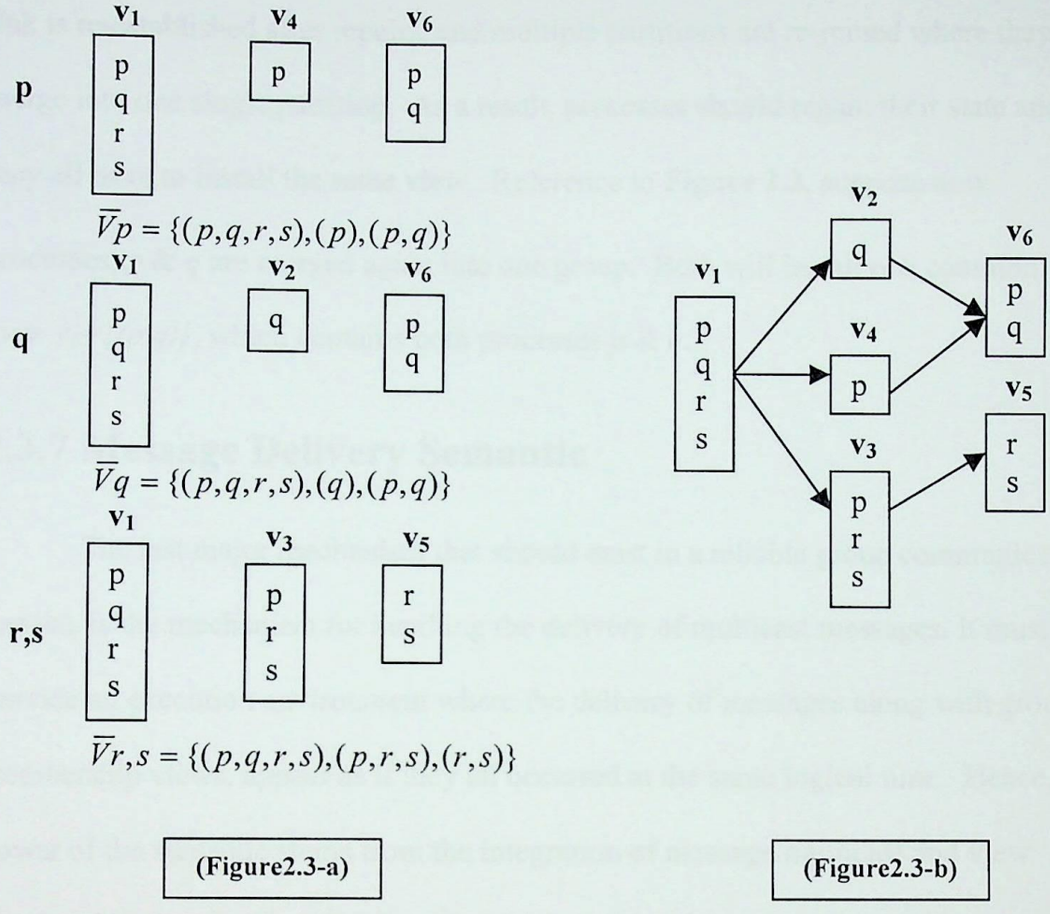
Figure 2.2: An Example of Clean Partitioning

- ***Non-Clean Partitioning***

In *Weak-Partial* or *Non-Clean Partitioning*, exceptions could occur causing a non-clean partitioning where the intersection among the partitioned groups, are not empty. In this case, views may not be distinct where we could experience the problem of overlapping views as discussed in [BDGB95]. Overlapping views occur, when a process is seen as reachable in different partitions, as depicted in the following example.

Consider four processes, *p*, *q*, *r* & *s*, fully operational and mutually reachable. All processes initially install the same view $v_1=\{(p,q,r,s)\}$. At a certain point in the execution time, *q* becomes partitioned from the rest of the group. Hence, *q* starts to install a successor view to v_1, $v_2=\{(q)\}$. Remaining processes *p*, *r* & *s* will detect that *q* is unreachable due to partitioning and they tend to install a new view excluding *q*, and installing $v_3 = \{(p,r,s)\}$. Processes, *r* and *s* were successfully able to install v_3,

however, a link failure occurred at p, and p is no longer reachable from the group. So, p has to install another view, $v_4=\{p\}$, including only itself.



(Figure2.3-a) (Figure2.3-b)

Figure 2.3: An Example of Non-Clean partitioning & Re-merging

As shown in **Figure 2.3**, one can find that p still appears reachable from r & s in v_3 while it is actually unreachable due to partitioning. Hence, p appears to participate in two views simultaneously, which implies that v_3 & v_4 are two overlapping views.

In systems like **Relacs**[BDGB95] and **Jgroup** [Mont99], they eliminated the overlapping views by assuming that sooner or later the following scenario would occur, where processes r & s will realize that p is no longer reachable, and it will exclude p from the group installing another view consisting of r & s only, stated as, $v_5=\{(r,s)\}$.

2.3.6.2 Group Re-Merging

As previously noted, link failures are not permanent, i.e., a communication link is re-established after repairs, and multiple partitions are re-joined where they all merge into one single partition. As a result, processes should regain their state and they all have to install the same view. Reference to **Figure 2.3**, suppose now processes, p & q are merged again into one group. Both will install one common view $v_6=\{(p,q)\}$, which contains both processes p & q.

2.3.7 Message Delivery Semantic

The last major mechanism that should exist in a reliable group communication service, is the mechanism for handling the delivery of multicast messages. It must provide an execution environment where the delivery of messages along with group membership views, appear as if they all occurred at the same logical time. Hence, the power of the semantic stems from the integration of message multicast and view changes installed by the group membership service.

Different types of message delivery semantics appeared in recent years to adapt with the problem of partition failures:

2.3.7.1 Virtual Synchrony Semantic

This type of semantics was designed to fit the "*no-partition*" assumption where it handles message delivery for only a single group that has not partitioned, or handles a group where the maximum number of operational processes reside when a partition occur. Virtual synchrony guarantees a consistent delivery of multicast messages where processes belonging to the group, must deliver the same set of messages in the same specified order. Its main drawback appears when a partitioning occur, where the system could deliver messages in an order inconsistent with the

order of messages imposed inside the group. Hence, virtual synchrony, has a limited capability, since it fails to handle network partitions, re-merges, or process recovery.

2.3.7.2 Extended Virtual Synchrony Semantic

It generalizes the notion of virtual synchrony to support process failure, process recovery, Network partitions and re-merges. Hence, it allows for different message deliveries among processes when partitioning occur. In the mean time, it stresses that all multicast messages must be consistently delivered. Hence, it requires that despite of partitioning, messages have to be delivered in *agreed total and safe order*.

As described in [AMMAC95], a *consistent total agreed order delivery* means that when a process delivers a message, it should already have determined all prior messages originated within its current view. While *safe order delivery* implies that when a process delivers a message, it must be guaranteed that every process has already received and will deliver the message unless that process crashes.

Restricting message delivery semantics would render system performance, since total and safe order delivery cannot be guaranteed when group re-merging occurs, resulting in message inconsistencies.

2.3.7.3 View Synchrony

View synchrony acquires a more relaxed structure than the extended virtual synchrony type. Besides allowing for partitions and re-merges, it allows for no-ordering semantics in its structure, but, it can only provide guarantees on messages delivered among processes who survived from one view to the next consecutive view belonging to the same partition.

2.4 System Models Supporting Partitioning

Much work has been done in the area of group communication. In this work, we are concentrating on those models that handle the problem of partitioning and re-merging. In addition, we show how this has affected their specifications of the group membership service; message multicast, delivery semantics and finally, failure detection.

2.4.1 Primary Partition Model

2.4.1.1 Specification

First attempts to adapt the partitioning problem are classified as the Primary Partition Model. This model allows for partitioning to exist. However, it only considers partitions that has the maximum number of processes to be detected as the only surviving group, where the rest of the processes in other minor partitions are considered faulty and can only join the primary partition group as new members. Hence, it assumes the **Fail-Stop** model, where a process or network link recovery is impossible. In addition, the group membership specification requires that at most a single-agreed view of the group is active at any execution time.

2.4.1.2 Related Work

- **ISIS Toolkit**

ISIS toolkit, [BCG91, Ami95], is considered one of the first projects to design group communication service into their system structure. Processes in ISIS can dynamically join or leave groups. The system allows multicasting of messages to members of the group to be in total order (*ABCAST*), and/or in causal order (*CBCAST*). A membership service is built to notify for any process changes due to voluntarily join/leave requests or due to process failures. It allows for *Membership*

Atomicity by guaranteeing that membership changes are totally ordered and synchronized among communicating group members. ISIS failure monitoring technique is based on the *Fail-Stop* model by allowing for *Failure Atomicity* to ensure that processes that are no longer within the group will transparently be terminating and can join only as new members. Hence, process recoveries, network partitions and re-merges are not supported. However, ISIS toolkit was the first to provide virtual synchrony semantics to handle the synchronization of multicast messages. It ensures progress of connected components inside the group, however, if a network failure occurred, ISIS may deliver messages in an order inconsistent with the order determined by the group. Unfortunately, ISIS is no longer commercially available.

- **Electra Object Model Toolkit (EOM Toolkit)**

This powerful toolkit was built at the University of Zuirch in 1995 [Maf95]. The power of this model stems from permitting object-oriented programming in its group communication service structure through a *CORBA*-compliant interface. Members of the group are viewed as objects where their detailed specifications are totally encapsulated. Electra system design is flexible enough to adapt with various communication subsystems like **ISIS** [BCG91], **HORUS** [RBG92] or **Transis** [DMS95-2]. The system provides asynchronous one-to-many communication. It allows for a reliable multicast operation for message passing among group members. Since EOM is a primary partition model, its membership service propagates changes only due to member failures or join/leave requests ensuring a single agreed and ordered view among objects of the group. Electra uses a failure suspector service to suspect failures allowing for both actual and virtual process failure assumptions. Hence, partitions and re-merges are not allowed. However, executions in Electra are virtually synchronous and consider the synchronization model of EOM where events

like the delivery of multicast messages, view changes and failures appear as if all occurred at the same logical time.

2.4.1.3 Problems with the Primary Partition Model

The primary partition model is found unsuitable to support 'partition-aware' applications that run in modern large scale asynchronous distributed systems. This is due to the fact that guarantees to have a single agreed view at any point in the execution time, is impossible. In addition, exception might occur where a primary group cannot be maintained. This could occur if the system splits into multiple small partitions where non of them fits to be selected. As well as, when a majority of the machines fail, a primary partition would be impossible to allocate. Hence, attempts started to head towards supporting multiple group partitioning.

2.4.2 Multiple Partition Model

2.4.2.1 Specification

In this model, group splitting when network link failure occur and, group re-merging when communication link is restored, are both allowed. Hence, group membership semantics serve not only changes in the process level, i.e., simple join/leave requests, but in communication between processes in the same partition and in changes among processes in multiple partitions. As a result, multiple concurrent views are installed, since processes in different partitions will have different perceptions of the membership of the group. Virtual synchrony will further be enhanced in this model to support partitioning which introduces the notion of extended virtual synchrony and view synchrony. In addition, failure detection technique will no longer assume the *Fail-Stop* model, but rather, a failure detection

mechanism will be supported to suspect both, actual or virtual process and link failures.

Systems that support this type of model are considered 'partition-aware' systems, since progress is made despite the existence of multiple concurrent partitions. The following related work specifies the characteristics of 'partition-aware' systems that fit the multiple partitioning model.

2.4.2.2 Related Work

- **Newtop System**

Newtop, [EMS95, Ami95], is a general purpose group communication protocol under the asynchronous communication environment. In this system structure, processes may belong to many groups with varied sizes. Communication is provided through reliable multicast preserving causality and total order delivery of messages. Newtop, handles process crashes and network partitioning, but it doesn't allow for process recoveries and group re-merging. A weak failure detection technique exists, where the system could suspect processes or communication links that have never been suspected or have never even failed. Its group membership service handles process changes due to suspected process failures, or network partitioning. But despite of the changes, Newtop, imposes total ordering of views and liveness to be preserved. Since partitioning is allowed, the message delivery semantic used is the Extended Virtual Synchrony type, but in its limited scale since group re-merging is not supported.

- **HORUS Model**

A redesign of the ISIS toolkit built in the Computer Science department at the Cornell University, is called the HORUS toolkit [RBG92, Maf95, Ami95]. HORUS system structure is described as 'a Box of Lego Blocks', since it is extensively layered

and highly configurable. In addition, HOURS is designed to support the group communication paradigm using unreliable and/or reliable multicast with FIFO, causal, or total ordering semantics. It supports both group partitioning and re-merging due to link or process failures. A *Membership Layer* is designed to handle group membership maintenance, as well as, an *EVS layer*, to maintain Extended Virtual Synchrony semantic.

- **Totem Protocol**

Totem protocol was built at the University of California, Santa Barbara [AMMAC95, Ami95]. It is built on a broadcast domain consisting of a finite number of processes providing reliable multicast and process membership service. A *Single-Ring Protocol* is used to handle processes within a certain broadcast domain, i.e., within a group, while a *Multiple-Ring Protocol layer* is devised to handle reliable delivery and ordering across the entire network. The model handles process failures and recovery as well as network partitioning and re-merging. Group membership service allows for concurrent views to exist with consistency guarantees, if partitioning is a clean-partition; i.e. if processes in different partitions are mutually unreachable. However, inconsistencies appear if a non-clean partitioning occur, since agreed and safe order delivery of messages is required.

- **Transis System**

Transis [DM95, Maf95, Ami95] is developed at the Hebrew University of Jerusalem. It provides group communication service in a partitionable network environment. Its communication subsystem provides causal and total order multicast. Traniss system configuration allows for network and process crashes along with partitioning and re-merging. Similar to Totem, Transis delivers messages in total agreed and safe order delivery allowing for Extended Virtual Synchrony. Hence,

group membership is 'still' not strong enough since total ordering guarantees would cause inconsistent message delivery when concurrent views are installed.

- **iBus Toolkit**

iBus, [iBus], is a commercial product that is under development at a Swiss Software Company located in Zuich. iBus is a 100% pure Java supporting the group communication paradigm. Its configuration is based on multicast channels where clients could subscribe to more than one channel through push/pull messages to receive results published from servers. iBus has an extensively layered structure similar to HORUS, providing a reliable/unreliable multicast layer, a membership layer to handle view changes but unfortunately, it has no support for message delivery semantic layer to synchronize between group membership changes and message deliveries.

- **RELACS Model**

RELACS communication subsystem [BDGB95] supports group based communication service over wide-area networks. Processes communicate through reliable/unreliable multicasts allowing sometimes for duplicate messages and no ordering guarantees to exist. The underlying system layers of RELACS contain a *View Change Layer* for group membership maintenance. It allows for multiple concurrent views to exist among different partitions, but ensures view agreement among members belonging to the same partition. Ordering guarantees are not restricted in RELACS, hence, View Synchrony semantics for message delivery is supported.

- **Jgroup System**

Jgroup is a system under development at the University of Bologna, Italy [Mont99]. It is a Java RMI based, implemented under Java distributed object model.

Processes are viewed as encapsulated objects, as in the Electra toolkit, where clients invoke methods transparent to the object groups, viewed as a single object or entity. Group Membership allows for view changes as a result of process failures or network link failures. Changes among objects belonging to the same partition should reach a final agreed view to be installed among all members. However, it allows for concurrent views to be installed among different partitions preventing overlapping views and ordering restrictions. View Synchrony is thus allowed to handle message delivery to become consistent with membership results. In addition, when partitions start to re-merge, a reconciliation protocol is devised to bring all processes that joined from different partitions back to a consistent agreed state.

2.5 Table of Comparison

The following table of comparison, **Table 2.1**, summarizes the above mentioned systems showing the rapid transition from the primary partition model to the multiple partitioning model along with their support to group splitting and re-merging. It also shows how message delivery semantics along with group membership service and failure detection mechanisms, affect the criteria and performance of the imposed models.

	ISIS Toolkit	EOM Toolkit	Newtop Protocol	HORUS	Totem System
Group Communication (GC)	-Supports GC -Processes Join and leave dynamically -Uses CBCAST and ABCAST multicast	-GC Handled by object groups. -Asynchronous 0ne-to-Many communication. -Reliable Multicasts	- A general purpose communication. -Reliable multicast with causal and total order delivery.	-Implements GC using unreliable & reliable FIFO, causal or total multicast services -Extensively layered and highly configurable.	-Single ring protocol to handle processes within a single broadcast domain -Multiple ring protocol to handle reliable delivery and ordering across the entire network.
Group Membership	-*Membership Atomicity*: Guaranteeing that membership changes are totally ordered and synchronized among group members. -Sends membership notification when group members has changed their state due to processes joining/leaving the group.	-Membership changes: propagates changes due to object failure or join/leave requests. -A single agreed, totally ordered view must exist.	-Designs a membership service to support concurrent existence of views. -imposes total ordering of views, and preserves liveness.	-A membership layer is provided for membership maintenance. -Membership changes are delivered to the application among the stream of regular messages	-With clean-partitioning, it allows for concurrent views to exist. -Agreed and safe order delivery is imposed
Network Partition	Not supported	Not Supported	Supported	Supported	Supported
Re-merges	Not supported	Not supported	Not supported	Supported	Supported
Message Delivery Semantics	First to provide Virtual Synchrony (Primary Component Model)	Virtual Synchrony (Primary Partition Model)	Virtual Synchrony (Extension of the primary component Model) Weak-type	Extended Virtual Synchrony (Multiple Partition Model)	Extended Virtual Synchrony (Multiple Partition Model)
Failure Detection Technique	-Failure Atomicity -Fail-stop type	Handles Actual and Virtual Process failure -Fail-Stop model.	-Handles process failure -no process recovery	A failure detection technique for communication problems	Handles process failures using a recovery protocol.
Problems	-ISIS may deliver messages in an order inconsistent with the order determined by the primary component in case of partitions. -No longer available.	-Message inconsistency when a partitioning occur. -Its underlying system model depends on ISIS.	- A weak failure detection technique allowing for false suspicions to processes or link failures that have never been suspected to have failed/ -Imposes total ordering -supports overlapping groups	GMP service is not specific	-Has exceptional performance when Totem is extended to multiple-rings -Message inconsistency appears when a non-clean partitioning exists.

Table 2.1: Primary-Partition & Multiple-Partition System Models

	Transis System	iBus	RELACS	Jgroup
Group Communication (GC)	-Provides group communication services in partitionable network. -Uses reliable ordered and causal group-multicast	-A commercial product written in Java supporting GC. -Publish/Subscribe model -Reliable/unreliable Multicast.	-A system explicitly designed to support communication over wide-area networks.	-An Extension of the Java Distributed Object Model (RMI) based on the GC paradigm.
Group Membership	-Membership changes are delivered to the application among the stream of regular messages in total agreed and safe order.	-A membership Layer is supported. Where Group membership protocol ensures that view changes are delivered to all members of a channel, ensuring they all agree on the views they receive.	-Contains a *View Change layer*: ensuring agreement among members belonging to the same partition, and allows for concurrent views to exist in multiple partitioning.	-Objects forming a group are kept informed about the current membership of the group itself, that may vary at runtime, reaching finally a single agreed view. It also allows for multiple concurrent views to exist.
Network Partition	Supported	Supported	Supported	Supported
Re-merges	Supported	Supported	Supported	Supported
Message Delivery Semantics	Extended Virtual Synchrony (Multiple Partition Model)	Not supported	View Synchrony	View Synchrony
Failure Detection Technique	Handles process and communication link failures.	Failure Notification service	Failure Detector	Handles process and communication failures.
Problems	Inconsistent message deliveries when concurrent views exist.	-The current version doesn't provide view synchrony; In other words, there is no guarantee that message deliveries are synchronized with view changes	-Allows delivery of duplicate messages -Provides no message sequencing guarantees. -Overlapping views	-Reconciliation Problem.

Table 2.1-Cont'd: Primary-Partition & Multiple-Partition System Models

2.6 Summary

The underlying system model that is to support our work should be asynchronous distributed system model with a formal specification of the group communication paradigm. It should allow for a reliable total and/or causal order multicast, with a support for multiple-partitioning. Group membership service must be formally specified, View Synchrony semantic should be supported and a failure detection mechanism should be used to handle suspicions and failures when partition and re-merges occur.

Chapter 3

3 Partitionable Group Membership Service (PGMS)

As described in chapter 2, group membership service plays a major role in the construction of a reliable group communication paradigm in asynchronous distributed systems. As a result, a formal and detailed specification for group membership must be stated for an efficient handling of process-to-process or multiple process communication inside the group, not only on normal join/leave operation but also in case of process failures, network partitioning failures and merges. So, the main focus of this chapter is concentrated on the group membership service specifications in multiple partitioning environments stated as Partitionable Group Membership Service, or PGMS.

3.1 "Need-To-Have" Characteristics of PGMS

The following, states the main characteristics that we need to have, or expect to find in Partitionable Group Membership Service requirement.

3.1.1 Single Partition Level

In this level, we consider those processes belonging to the same partition or within a single group. PGMS must guarantee that processes residing in the same group must reach final agreement in their view composition and install a single agreed view that matches the group's local perception. Changes in views reported by PGMS in this level, could result from changes in,

- **The Process itself:**

Where PGMS reports changes when a process dynamically join or leave the group, or when a failure occur, where a process has to be expelled from the rest of the group.

- **Between Process-Pairs**

PGMS not only considers changes inside each process, but also changes that occur between process pairs when they communicate within the group. Hence, there must exist a mechanism where each process could indicate the list of processes reachable from it during execution. So, when two processes belonging to the same partition are mutually reachable from each other, they must maintain the same view at the end of the execution process. However, temporary inconsistencies might occur during execution time causing processes to become unreachable. Temporary inconsistencies result from actual/virtual process failures or link failures and partitioning. Hence, when a process suspects its other party to be faulty, PGMS must allow this process to remove the suspected unreachable one from its list, and install a new view excluding that process. Moreover, PGMS must notify other reachable members of the change, so that they could change their views as well. Hence, Partitionable Group Membership Service must guarantee that each installed view is shared by all members of the group engaged or is affected by this change.

3.1.2 Multiple Partition Level

At this level, PGMS deals with changes occurring between processes in multiple partitions when link failures occur. So, PGMS must devise a partitioning model to allow for multiple concurrent views to exist, where each partition reports its final agreed view that might be different from other views residing in other partitions.

In addition, it should allow for a re-merging paradigm to exist, where the merged processes adjust their behavior to become consistent and to install a final view agreement among them.

3.2 Formal Specification of PGMS

In order to allow for the existence of the above mentioned characteristics, formal specifications of Partitionabl Group Membership Service must be stated and carefully defined. Much of the theoretical and experimental work done in this area is to define a set of formal requirements for the Group Membership Service. However, most of the numerous prior attempts have proven to be unsatisfactory and neither of them successfully installed a concrete specification of PGMS. Hence, Partitionable Group Membership Service is still considered an *open problem*.

Babaoglu [DBM97] stated that, in order to reach a formal specification of PGMS, we need the specification to be strong enough to preclude useless or trivial solutions as well as being weak enough to be implementable in asynchronous systems with failures encountered.

3.3 Related Work

A lot of work has been done in this area trying to enhance PGMS specifications. This section describes four major and recent works that attempt to formally specify set of PGMS requirements in order to allow for better system performance. These requirements are stated in detail including the problems that might emerge when they are strictly followed in the system.

3.3.1 Strong Group Membership (S-GMP) by Riccardi

Riccardi's [Ric93, RB93] major aim in her work is to support a group membership service that is strong enough to achieve the primary partitioning model. Though our aim is concentrated on the Partitionable Group Membership Service requirements, it is so important to first understand the group membership specification built to achieve the primary partition model and show how restrictions imposed by the model affects the performance of *S-GMP*.

3.3.1.1 S-GMP Properties:

Strong-GMP requirements built in Riccardi's work contains some trivial and non-trivial properties that have to be acquired and achieved,

3.3.1.1.1 GMP-0 (Base Case):

This trivial property is considered in the group initialization phase, where it requires a group to be defined. In which, an initial view must be set before interactions between members start.

3.3.1.1.2 GMP-1 (Validity):

The validity requirement controls the view installation inside the group. For a process that was once within the group and suddenly is no longer reachable or has failed, *S-GMP* must exclude the faulty process from the local views of the other members believing that this process has crashed. It also imposes a powerful restriction where installation of views is only acquired by the *S-GMP*, if certain events have previously occurred. So, it disallows the problem of *capricious view installation* that will be discussed in full in the coming models.

3.3.1.1.3 GMP-2 (Uniqueness):

Uniqueness is another trivial solution in *S-GMP* where it requires that every view change installed by the process during execution is unique and strictly defined.

3.3.1.1.4 GMP-3 (Sequence):

Since the model tries to achieve the primary partition concept, all processes within the group, when views are defined, must acquire the same sequence of local views. Such that, when the system tries to check a certain view at a certain execution time, views in all processes participating in the group must be identical.

3.3.1.1.5 GMP-4 (Liveness):

The final requirement built in this model ensures liveness during group operation. It checks that if a process p, a member in the group, suspects another process q to be faulty, *S-GMP* either removes p from the group view, or removes the suspected process q.

3.3.1.2 Problems in S-GMP:

Validity, *Sequence* and *Liveness* properties are non-trivial requirements that are of major interest to us, by which we can predict whether *S-GMP* specifications are formally defined or not. Unfortunately, when *S-GMP* is applied to achieve the primary partition model, major problems were encountered:

- **Informal and Ambiguous Specifications**

Anceaume in his paper [ACT95] was able to analyze the logic that Riccardi used to prove her assumptions. Riccardi expressed the *S-GMP* properties using *Temporal Logic,* which contains predicates, that use *Branching Time Logic.* Anceaume proved that there are flows in the formal description of *S-GMP* properties. He stated that some formulas used to express the requirements do not match the

informal English description that defines them. This problem was found in both *GMP-3* & *GMP-4* specifications. Consider the following flow that was found:

The formula that is used to express the (*GMP-4*), ***Liveness*** property, is stated as,

$$\Diamond Out - GP_q \vee Out - Gp_p$$

Where ($\Diamond$) is a branching time logic indicating that the event could hold for sometime during multiple execution runs. While *Out-GP,* indicates that the specified process will be removed from the group. *GMP-4* states in an informal English description that when a process q is found faulty, either the suspecting process p or the failed process q is removed from the view. The logic described above, enforces the removal of p or q at every run independent of the future system behavior, i.e., it imposes a "*uniform future*." However, if the logic is reformulated as,

$$\Diamond(Out - GP_q \vee Out - Gp_p)$$

Removal of p or q will be restricted depending on the system behavior, which is what the requirement informally indicates. As a result, ambiguities during execution may occur in Riccardi's work.

- **Violation of Safety**

As noted in (*GMP-3*) ***Sequence*** property, views at any run must be identical. Apparently, this requirement can only be partially specified if no failures or partitioning occur. However, since partitioning and failures must be explicitly confronted, inconsistencies between views might emerge resulting in the violation of this property. Consider the following example:

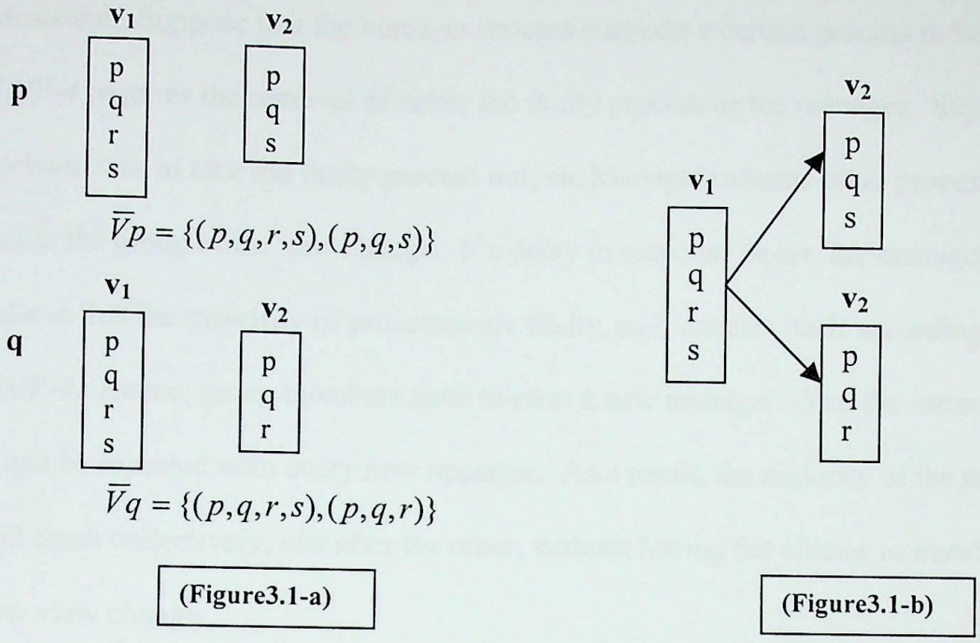


Figure 3.1: Violation of Safety Property (*GMP-3*) in Riccardi

As shown in **Figure 3.1**, suppose you have a primary partition that consists of four processes, *p*, *q*, *r*, and *s*. At a certain point in the execution time, *p* suspects that *r* is faulty and *q* suspects *s* to be faulty at the same time. As a result, *p* installs a new view excluding *r*, $v_2^p = \{(p,q,s)\}$, while *q* excludes only *s*, installing $v_2^q = \{(p,q,r)\}$. Now, v_2^p and v_2^q are not identical which violates the *GMP-3* property.

- **Violation of Liveness**

Another problem was found that violates the requirement stated in *GMP-4* as well. The system would run into a *Collective Suicide Problem.* Collective suicide, means that, processes within the group are collectively removed from the group and are considered faulty without having the chance to change the current view of the group.

Consider the following scenario, described in [ACT95], when processes inside the group reach agreement in their views, they report changes to a manager process

that handles and coordinates updates of those views ensuring that all views are consistent. Suppose that the manager process suspects a certain process to be faulty, *GMP-4* requires the removal of either the faulty process or the manager. Suppose the decision was to take the faulty process out; so, Manager informs other processes inside the group of the new change. If a delay in response occur, the manager could believe that the majoirity of processes are faulty, so it crashes itself according to *GMP-4*. Hence, group members have to elect a new manager. Yet, the same situation might be repeated with every new manager. As a result, the majority of the processes will crash collectively, one after the other, without having the chance to install the new view change.

From the above, mentioned problems, one can note that in primary partition model, it is not easy to construct formal specification for Partitionable Group Membership Service.

3.3.2 Newtop

We have mentioned Newtop before in the previous chapter, but our major concern now, is to study its group membership main specifications and properties. In this model, group membership was designed to support partitioning, allowing for distinct concurrent views to exist among the multiple surviving partitions [EMS95].

3.3.2.1 Membership Properties in Newtop

3.3.2.1.1 VC1 (Validity)

The validity requirement in Newtop is similar to the ***sequence*** property stated by Riccardi. However, it is applied in a multiple partitioning environment, where it ensures that the sequence of views installed by members that do not suspect each other in a certain partition, are identical.

3.3.2.1.2 VC2 (Liveness)

The Liveness property in Newtop, enhances Riccardi's ***Validity*** requirement. Consider a process q reachable from a functioning process p, and belongs to its current view v_1, crashes or becomes unreachable. Liveness guarantees that p will automatically install another successive view v_2 excluding the faulty process q.

3.3.2.1.3 VC3 (Virtual Synchrony)

Synchronization between multicast messages and membership views must be atomic ensuring consistent delivery of messages.

3.3.2.2 Problems in Newtop

Newtop could be easily satisfied by the following trivial solution: "*Every process p initially installs {p} as its first view of the group and never installs another view after this*" [ACT95]. This solution is satisfied by Newtop's group membership service, which raises a major problem in the formation of views, causing what is called ***Capricious View Splitting***.

- **Capricious View Splitting**

This problem allows the splitting of all processes into a single "one-member" view forever. This of course, is undesirable in our formal specification of group membership, since we do not aim at single process interaction, but at changes, due to the interaction between processes in the same partition or in multiple partitions, as described in section 3.1. This problem has been overcome in the new specifications of the group membership service designed in Dolev's work [DMS95-2].

3.3.3 Dolev's Partitionable Group Membership Service

A new framework for Partitionable Group Membership Service was maintained in this model. It does not only handle partitioning, but also handles re-

merging, ensuring agreement of views among the different merged partitions. This work was applied and implemented in the Transis system [DM95], see (**Table 2.1**-chapter 2).

3.3.3.1 PGMS Properties

3.3.3.1.1 M.1 (Self-Inclusion)

Self-Inclusion, is a trivial requirement which states that each member inside the group has his own set of views installed, and the whole group is a member of its installed membership events.

3.3.3.1.2 M.2 (Order)

When processes inside the group change their views, PGMS goes through these changes in the same order.

3.3.3.1.3 M.3 (Virtual Synchrony)

When membership changes, Virtual Synchrony ensures safe delivery of messages in causal order.

3.3.3.1.4 M.4 (Liveness/Termination)

This property ensures the termination of the membership protocol where blocking can never occur, by guaranteeing that if a process is suspected to be faulty, it has to be removed from the membership views within a finite time.

3.3.3.1.5 M.5 (Non-Triviality)

This Non-Triviality property solves the trivial solution encountered in Newtop. It guarantees that processes always have a chance to form a connected membership with other processes regardless of any event executed in the past.

3.3.3.1.6 M.6 (Uniformity)

It guarantees that the process partitioned from the group should continnue opearation as well as all machines interacting with that process.

3.3.3.2 Problems in Dolev's Work

Though the non-triviality solution was able to solve the problem of *capricious view splitting*, yet, it allows for another major problem to exist, called *capricious view installation.*

- **Capricious View Installation**

This problem arises in Dolev's PGMS specification, since it doesn't regulate the installations of the new views reported. This problem was prevented by Riccardi's validity requirement (*GMP-1*), where it restricts the installation of new views only if certain events have previously occurred. However, PGMS in Dolev's model neglects that major criteria which caused new views to be installed at any time in an arbitrary way and with no reason, without even considering whether this view is the result of a process join/leave or because of failures. This could also lead to the arbitrary removal of processes that have not failed or have not even suspected to have failed. Hence, as Anceaume [ACT95] and Dolev suggested, in order to eliminate useless protocols as well as capricious installation of views, a failure detection mechanism should be extended along with PGMS to detect process failures and suspicions, which would make the system behavior more predictable.

3.3.4 Partitionable Group Membership Service in Babaoglu

The most recent work in this area of research is done and, still, being updated in the Computer Science Department at the University of Bologna, Italy. Babaoglu's work, [BDGB95, BDM98, BDM99] in building a reliable group communication service supporting PGMS and View Synchrony, has actually been implemented in two major projects, **RELACS** [BDGD95] and **Jgroup** [Mont99], see **(Table 2.1–**

chapter 2). Focusing on Partitionable Group Membership Service, this work aims at building a set of formal specifications of PGMS avoiding flaws and trivial solutions found in the aforementioned models. In addition, it satisfies almost all of the "*Need-To-Have*" characteristics described in section 3.1.

3.3.4.1 PGMS Properties

Properties in Babaoglu are divided into three major sections. One that specifies properties related to the *single-partition* level, another part is related to the *multiple-partition* level, and finally, properties that applies to both levels. It also excludes view synchrony semantics from the set of PGMS specification and devises a separate set of properties and specifications for handling message delivery semantics, which has not been done in any previous work.

3.3.4.1.1 GMP Properties Acquired in a Single-Partition Level

View Accuracy and View Completeness are of fundamental importance in the construction of PGMS.

3.3.4.1.1.1 GM1 (View Accuracy)

View Accuracy requires that each process install views including those processes reachable from it, and exclude those that are considered unreachable. As well as, if a process q is permanently reachable from another process p, then p will include q in its local view forever.

3.3.4.1.1.2 GM2 (View Completeness)

View Completeness, on the other hand, allows PGMS to check for unreachable processes that were partitioned from the rest of the group and remains to be unreachable. In such case, PGMS ensures that all local views in every correct process will never include any of the unreachable sets.

3.3.4.1.2 Properties Acquired in Multiple-Partition Level

3.3.4.1.2.1 GM3 (View Agreement/Coherency)

Also stated as View Coherency in [BDM99]. This property is extremely crucial for the validity and termination of the protocol. This property applies when partitioning occurs, where views are re-adjusted to fit the current change. First of all, it checks that, all processes belonging to the current view *v* for a certain process *p*, install the same view, or else, *p* changes its current view with a new successor view including only those processes who survived that change. Second, consider two processes, *p* and *q* that are mutually reachable and have the same current view *v* initially installed. *p*, then, installs a successor view to *v*. So, PGMS enforces *q* to either install a successor view, or crashes. Finally, this property ensures that all processes that installed a successor view like *p*, must have initially installed view *v*.

3.3.4.1.3 Properties Related to Both Levels

3.3.4.1.3.1 GM4 (View Order)

As previously noted, other systems enforce total and consistent ordering of views among all members of the group despite of partitioning. However, Babaoglu relaxes the criteria somehow, to adapt with changes in views in multiple partitions. Hence, concurrent views are allowed, where ordering semantics is not enforced, while ordering of views is enforced among processes residing in the same partition that survived from one view change to the next successive view. In other words, if two views, *v* & *w* are installed in some order in a process, *p*, in the meantime, another process *q* installs the same view, then q must install those views in the same order found in *p*.

3.3.4.1.3.2 GM5 (View Integrity)

This property ensures that every view installed by a process should include itself. This ensures that views installed are not arbitrary.

3.3.4.2 Problems Solved in PGMS

PGMS in Babaoglu has been successfully able to prevent most of the problems encountered in previous models.

Solution (1): ***PGMS Prevents Capricious View Splitting***

This was satisfied in ***View Accuracy*** (*GM1*), by requiring that all views installed by two reachable processes must include each other.

Solution (2): ***PGMS Prevents Capricious View Installation***

View Accuracy *(GM1),* along with ***View Integrity*** *(GM5)*, helps in preventing this problem. By which, ensuring that each process installs a view, must include itself. This would prevent useless protocols to exist. In addition, *GM1* requires that, the composition of views can only be installed, if changes in events have previously occurred during execution time.

Solution (3): ***PGMS Prevents Lack of Liveness***

View Agreement property (*GM3*), guarantees liveness, by ensuring that view installations could never be delayed indefinitely or arbitrarily under stable system conditions.

3.3.4.3 Problems in PGMS

Since the work proposed by Babaoglu is still very recent, no paper was found that criticizes their work. My thesis will be the first work that aims at enhancing their PGMS specifications. This will be discussed in more depth in the rest of my thesis.

3.4 Summary

The following table, **Table 3.1**, summarizes the four models discussed in this chapter. It pinpoints their PGMS specification along with the problems encountered by those models. As well as, showing the transition while attempting to solve these

problems. In brief, this chapter proved that still, no complete and formal specification of Partitionable Group Membership Service is found. Almost all existing group membership requirements suffered from one or more of the following:
(1) Specifications are informal and ambiguous [Ric93, RB93], (2) capricious view splitting [EMS95], (3) capricious view installation [DMS95-1], (4) violation of safety and /or liveness [Ric93, DMS95-2].

However, recent work of Babaoglu gives better formal specification of PGMS that handles view changes not only on a single partitionable level, but also in multiple partitionable environments. It is *strong* enough to solve most of the aforementioned trivial problems and still is *weak* enough to be implementable in partitionable asynchronous distributed environment. Hence, my thesis focuses on this major work and enhances on it for better specifications of PGMS.

RICCARDI (S-GMP)	NEWTOP	DOLEV	BABAOGLU
GMP-0(Base-Case) An initial Group View must exists before interaction between group members. **GMP-1(Validity)** •Controls view installations in the group • SGMP must exclude faulty processes from the local views of other members. •Views are installed if certain events previously occurred. **GMP-2(Uniqueness)** Every view change installed during execution is unique and strictly defined. **GMP-3(Sequence)** When views are defined, all processes within the group must acquire the same sequence of local views. **GMP-4(Liveness)** It checks if a process *p*, a member in the group, suspects another process *q* to be faulty, either *p* is removed from the group view or *q* is removed ***Problems*** • *Informal and Ambiguous specifications.* • *Violation of Liveness:* Collective Suicide	**VC1(Validity)** The sequence of views installed by members that do not suspect each other, are identical. **VC2(Liveness)** If a process *q*, belongs to a *p*'s local view, leave the group, *p* installs another view that excludes this faulty process. **VC3(Virtual Synchrony)** The delivery of a message to the members of a group must be atomic with respect to view updates by the membership service. ***Problem*** Capricious view splitting—Solved in Dolev	**M.1(Self-Inclusion):** A group is a member of its installed membership events. **M.2(Order)** Changes in views are handled by the Group Membership in the same order. **M.3(Virtual Synchrony)** Ensures safe delivery of messages in causal order. **M.4(Liveness/Termination)** Guarantees that suspected processes are removed from the membership view within a finite time. **M.5(Non-Triviality):** • Prevents capricious view splitting. • Guarantees that regardless of any events executed in the past, any process has a chance of forming membership with other processes. **M.6(Uniformity):** Specifies what should happen when a certain membership view is installed. ***Problem*** Capricious View Installation	**GM1(View Accuracy)** Each process should install views that include all processes reachable from it and exclude those that are unreachable. **GM2(View Completeness)** Ensures that all local views in every correct process will never include any set that remains to be unreachable. **GM5(View Integrity)** Every view installed by a process includes the process itself. **GM3(View Agreement)** (1) All processes belonging to the current view *v*, for a certain process *p*, must either include the same view or *p* installs another successor view including only processes that survived the change along with *p*. (2) If *p* & *q* are two mutually reachable processes having the same view *v*, then *p* installs another successor view. *q* must either install a successor view or crashes. (3) All processes that installed a successor view *w* like in *p*, must have initially installed view *v*. **GM4(View Order)** Ordering are not guaranteed within multiple partitions but enforced among processes residing in the same partition, and have survived from one view change to the other. ***SOLVES:*** • *GM1* solves capricious view splitting • *GM1* & *GM5* solves capricious view installation • *GM2+GM3*Prevents Lack of liveness

Table 3.1: System Models Supporting PGMS

Chapter 4

4 The Jgroup Reliable Distributed Object Model

4.1 What is Jgroup?

As previously mentioned, Jgroup is an extension to Java distributed object model based on Group Communication [Mont99]. Its novelty over other existing object group systems is in its integration between two major requirements by which ensuring system reusability, interoperability and dependability. First requirement ensures that all objects interact via Remote Method Invocations (RMI) whether being internal to server groups or external to clients, unlike existing systems that use different communication paradigms for server objects than those used for services provided for clients. The other novel requirement is in its support for partitionable group communication paradigm that allows multiple views of the same group to exist concurrently by which ensuring dependability of the system and making it possible for partition-aware applications to be built and enhanced. Our major interest in this work is concentrated on the structure and algorithm used for building the underlying partitionable group communication service.

4.2 Partitionable Group Communication Service in Jgroup

The Jgroup communication architecture is an integration between *a group membership service* and a *reliable message multicast service*. The Membership service task is to keep members consistently updated with any changes that may occur

in the group as new members join/leave the group or due to failures, crashes, Network partitioning and re-merging. This is done via the installation of new views where its composition is shared and agreed upon by all members of the same group. As of the reliable message multicast service, it is the means by which members of the group communicate through the send and receive of multicast messages. Again focus will head towards the group membership service in the partitionable environment and its underlying properties making sure that the system's membership properties matches our desired PGMS requirements described in chapter 3.

4.3 The Group Membership Service

A group consists of a number of servers having the same interface and is uniquely identified by a group key. The partitionable group membership service PGMS in Jgroup keeps servers in the group informed of any changes in the group's current view through *vchg()* events. A view is denoted by v and has a globally unique *ID*. The composition of the view, $\overline{v}$ consists of a set of process names belonging to the current view. The last view installed by process p at time t is denoted by $view(p,t) = v$. The immediate successor of view v is w, denoted by $v \prec w$. The successor relation belongs to those views related to the same group. Those views that are not related are considered concurrent views. The composition of the installed views should not be arbitrary. Rather, it should reflect reality by ensuring reachability, i.e., processes should only install and change their views with other processes that communicate with directly. Specifications of the Partitionable Group Membership Service (PGMS) used in Jgroup are summarized in the following properties[BDM98, Mont99]:

GM1 (View Accuracy):

If there is a time after which two correct servers are permanently and mutually reachable, then eventually all views installed by one will contain the other. Formally,

$$\exists t_0, \forall t \geq t_0 : p \rightarrow_t q \wedge p \in Correct(C) \Rightarrow \exists t_1, \forall t \geq t_1 : q \in \overline{view(p,t)}$$

Correct(C): denotes those processes that never crashed and are correct

GM2 (View Completeness):

If there is a time after which two correct servers are permanently and mutually unreachable, then eventually all views installed by one will exclude the other. Formally,

$$\exists t_0, \forall t \geq t_0, \forall q \in \Theta, \forall p \notin \Theta : p \not\rightarrow_t q \Rightarrow \exists t_1, \forall t \geq t, \forall r \in Correct(C) - \Theta : \overline{view(r,t)} \cap \Theta = 0$$

Θ : A certain partition

GM3 (View Agreement/Coherency):

(i) ***if a correct process p installs view v, then either all processes in $\overline{v}$ also install v, or p eventually installs an immediate successor to v. Formally,***

$$p \in Correct(C) \wedge vchg(v) \in \sigma(p, \mathrm{T}) \wedge q \in \overline{v} \Rightarrow (vchg(v) \in \sigma(q, \mathrm{T})) \vee (\exists w : v \prec_p w)$$

(ii) ***If two processes p and q initially install the same view v and p later on installs an immediate successor to v, then eventually either q also installs an immediate successor to v, or q crashes. Formally,***

$$vchg(v) \in \sigma(p, \mathrm{T}) \wedge vchg(v) \in \sigma(q, \mathrm{T}) \wedge v \prec_p w_1 \wedge q \in Correct(C) \Rightarrow \exists w_2 : v \prec_q w_2$$

(iii) ***When process p installs a view w as the immediate successor to view v, all processes that survive from view v to w along with p have previously installed v. Formally,***

$$\sigma(p, t_0) = vchg(w) \wedge v \prec_p w \wedge q \in \overline{v} \cap \overline{w} \wedge q \neq p \Rightarrow vchg(v) \in \sigma(q, [0, t_0[)$$

where $\sigma(p,\mathrm{T})$*:* is the global history of an execution of an event for a certain process p at a certain time interval T. Such that, if an event e is executed $\sigma(p,\mathrm{T}) = e$, while if no event occurred $\sigma(p,\mathrm{T}) = \varepsilon$

GM4 (View Order)

The order in which processes install views is such that the successor relation is a partial order. Formally,

$$v \prec w \Rightarrow w \not\prec v$$

GM5 (View Integrity)

Every view installed by a process includes the process itself. Formally,

$$vchg(v) \in \sigma(p,T) \Rightarrow p \in \bar{v}$$

GM6 (Merging Rule)

Two views that merge into a common view must have disjoint compositions.

$$(v \prec u) \wedge (w \prec u) \wedge (v \neq w) \Rightarrow \bar{v} \cap \bar{w} = \phi$$

4.4 Partitionable Group Membership Algorithm

In this section we present the algorithm used by Jgroup to implement the PGMS service in partitionable asynchronous systems. The overall structure of the system is shown in **Figure 4.1** below. In this figure one can see that PGMS layer is placed between the application layer and the Multi-Send Layer (MSL). This MSL service is responsible for reporting any changes that occur in the network or any suspects conceived by the Failure Detector service (FD) to the PGMS layer by delivering sent messages to all processes in a certain destination for a certain group. Messages sent by MSL and used by the PGMS layer are of three types, *msend()*, *mrecv()* and *msuspect()*. A process could m-send or m-recv a message when it

communicaties with MSL through its notations *msend()* and *mrecv()*, respectively.This is used to distinguish between the send and recv primitives used in MSL with those used when a process directly communicates with the network layer. When events are m-received, m-suspected, or a new join or leave request is instantiated, the PGMS layer reports to all servers participating in the group that a change has occurred and a new view must be constructed and installed following the PGMS specifications. When the view is finally installed and the network is stable, i.e., all processes are mutually reachable from each other, the PGMS service reports to the application the *vchg()* and becomes ready to receive any new requests.

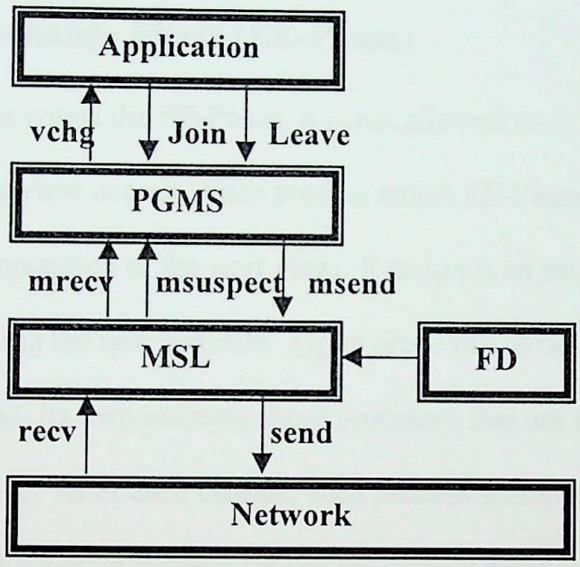


Figure 4.1: Jgroup Membership Service Architecture

4.5 Phases of PGMS

The partitionable Group Membership Algorithm has three major phases:

1. IDLE Phase

A process remains in *IDLE* phase until either, it is informed by the Multi Send Layer that a change in reachability has occurred and a new *m-suspect* event will be received or, it *m-receives* a message from another process that has observed this change. Hence, the process moves to the ***Agreement Phase***, which consists of two

sub-phases: the Synchronization Phase (S-phase) and the Estimate Exchange Phase (EE-Phase.)

2. Synchronization Phase (S-Phase)

The Synchronization phase synchronizes the different invocations of the algorithm ensuring that all processes participating in the next view have the same agreed round. This is done by each process sending a *SYNCHRONIZE* message to those processes perceived as reachable containing a version number and waits for response. To exit the S-Phase, each process must update all reachable processes' version numbers and set its agreed round number to the current agreed number.

3. Estimate Exchange Phase (EE-Phase)

Once a process enters the EE-Phase, it is not allowed to exit unless an installation of a new view occurs. Each process enters EE-Phase with its own estimate for the composition of the next view. It m-sends an estimate message to all participants containing the new estimate. Upon receiving those messages, the process removes from its own estimate those processes that are excluded from the estimate of the sender. After each change, each process sends an agreement proposal message to a certain process among the current estimate selected as a Coordinator. When the Coordinator *m-receives* proposal messages and observes agreement among each received proposal message, it installs the final complete view consisting of all processes in the message.

If the network is stable, and no upcoming events have occurred during EE-Phase, the agreement algorithm will terminate and return back to the IDLE State. However, if a change in the reachability relation occurred, the agreement algorithm will not exit, and it will return back to the Synchronization Phase and starts a new execution. A detailed description of Jgroup membership algorithm's specification and

implementation is shown in the next section. On the other hand, the algorithm can be described in the following finite state machine, **Figure 4.2**.

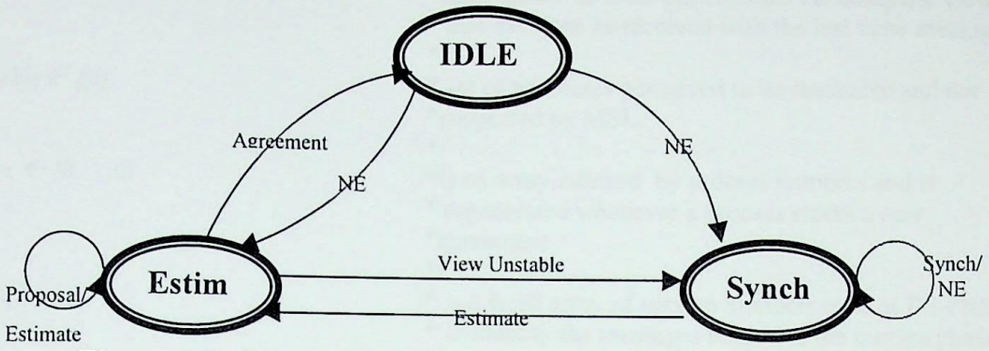


Figure 4.2: Jgroup Membership Algorithm State Diagram

4.6 PGMS Specification and Implementation

In this section, we give a detailed description of the Partitionable Group Membership Algorithm specification and implementation of Jgroup [BDM98], which will be needed as a reference in my newly proposed algorithms since they will be part of the Jgroup Toolkit.

The following is the list of global and local variables used in the algorithm as well as the message types used.

Global Variables:

View ← *(UniqueID(), {p})*	/* the view composition consisting of an identifier ID * and a view composition of the last view installed */
cview ← *view*	/*same fields as view but contains the complete view * that has been m-received with the last view message */
reachable ← *{p}*	/* set of processes perceived to be reachable and not *suspected by MSL */
version ← *(0,...,0)*	/*is an array indexed by process numbers and is * regenerated whenever a process enters a new *agreement */
agreed[]	/* is a fixed array of version numbers used at EE-Phase * to identify the messages related to the current phase */
estiamte ← *reachable*	/* a set of processes used as a proposal of the next view */
Symset[q]	/* an array indexed by the process number which *contains the last value stored in variable reachable * such that q ∉ reachable at a certain process p. */
stable	/* a boolean variable indicating whether the last * installed view corresponds to the reachability set * received from MSL. If not, the process will re-enter * the agreement phase again */
ctbl[]	/* is the coordinator table indexed by the process# * and stores the proposal message m-received from * each process participating in the current view. */
V[q]	/* the sender's current version number in the received * message */
(II-P)	/* *II* represents those processes that can communciate. * *P* are the new reachable set received from MSL */

Local Variables:

Event	/*stores the last event performed by a process p */
synchronized ← *{p}*	/*used in S-Phase and contains the set of processes *from which p has m-received an answer to its *synchronize request. */
installed	/*is a boolean variable used in EE-Phase, whose value *becomes true when the agreement protocol can *terminate */

Message Types

<SYNCHRONIZE, V_p, V_q, P>	/*used during S-Phase. V_p is the version# of the sender *V_q is the version# of the destination. *P* represents the *last symmetry set associated with the destination *process. */
<SYMMETRY, V, P>	/*used in both S-Phase & EE-Phase. Handles the case *where reachability received from MSL is asymmetric. **V* is the version number of the sender. *P* is its associated *current reachable set of processes at the time of sending */
<ESTIMATE, V, P>	/*used during EE-Phase. This message is m-sent to *processes belonging to the current estimate. Whenever * this estimate changes, a new message is sent. **V* is the version number array. **P* is the set of processes in the current estimate of sender */
<PROPOSE, S>	/*used during EE-Phase. A proposal message is m-sent to *the coordinator each time the next view changes. **S* contains the current sender's estimate status that will *be stored in the coordinator ***ctbl[]*** as an entry. */
<VIEW, w, C>	/*used during EE-Phase. This message is m-sent by the * coordinator after composing the new view to all *participant processes. *W* is the new complete view. **C* is the value of the coordinator table when agreement is *reached. */

Procedure PGMS():

This procedure is the main method for each running process. It describes the different cases that will cause the process to move from its IDLE state and start the agreement protocol. The process will continue remaining IDLE until a new event is m-received.

```
Procedure PGMS()

while true do
        wait-for event
        case event of

                1. msuspect(P):
                        foreach r ∈ (II-P) – reachable do symset[r] ← reachable
                        msend (<SYMMETRY, version, reachable>, (II-P) – reachable)
                        reachable ← II – P
                        AgreementPhase()

                2. mrecv (<SYNCHRONIZE, Vp, Vq, P>, q):
                        if (version[q] < V[q]) then
                                Version[q] = V[q]
                                if (q ∈ reachable) then
                                        Agreement Phase()
                                endif
                        endif
        endcase
endwhile
```

1. msuspect (*P*) Event:

This event is sent by the *MSL* service, notifying that a change in the reachable set of processes has occurred. Hence, process *p* updates its *symset[]* array with all processes perceived as reachable. And sends a *SYMMETRY* message to those processes notifying them of the update. In addition, it updates its own reachable set variable and then calls the *Agreement Phase()*.

2. mrecv (<*SYNCHRONIZE, Vp, Vq, P*>, *q*) Event:

This message is a wake-up message from another process that observed such a change. By which, *p* is notified to start calling the *AgreementPhase()*. Meanwhile, process *p* updates the sender's version number after making sure that *q* is part of its reachable set.

Procedure AgreementPhase()

This is the main part of the Agreement algorithm which takes the new estimate of the new composed view and continues calling the S-Phase and EE-Phase until a final agreement is reached and no upcoming events occur, i.e. view is stable.

```
Procedure AgreementPhase()

repeat
        estimate ← reachable                    /*initializing the next view estimate*/
        version[p] ← version[p]+1               /*generating a new version number*/
        SynchronizationPhaseInit()
        EstimateExchangePhaseInit()
until stable
```

Procedure SynchronizationPhaseInit()

Starting the Synchronization Phase, each process m-sends a synchronization message containing the process version number to those processes perceived as reachable and waits for response. Upon receiving each response message, the process

updates the sender's version number and sends another synchronous message to confirm the agreed round invocation number. The following, are the cases that cause the process to terminate the S-phase and move to the EE-Phase.

```
Procedure SynchronizationPhaseInit()

Synchronized ← {p}
Foreach r ∈ estimate – {p} do
        msend (<SYNCHRONIZE, version[r], version[p], symet[r]>, {r})

while (estimate ⊈ synchronized) do
        wait-for event
        case event of

                1. msuspect (P):
                        foreach r ∈ (II - P) – reachable do symset[r] ← reachable
                        msend (<SYMMETRY, version, reachable>, (II - P) - reachable)
                        reachable ← II – P
                        estiamte ← estiamte ∩ reachable

                2. mrecv (<SYMMETRY, V, P>, q):
                        if (version[p] = V[p]) and (q ∈ estimate) then
                                estimate ← estimate – P

                3. mrecv (<SYNCHRONIZE, Vp, Vq, P>, q):
                        if (version[p] = Vp) then
                                synchronized ← synchronized ∪ {q}
                                agreed[q] ← Vq
                        if (version[q] < vq) then
                                version[q] ← Vq
                                msend (<SYNCHRONIZE, version[q], version[p], symset[q]>, {q})
                        endif

                4. mrecv (<ESTIMATE, V, P>, q)
                        version [q] = V[q]
                        if (q ∉ estimate) then
                                msend (<SYMMETRY, version, estimate>, {q})
                        elseif (version[p] = V[p]) and (p ∈ P) then
                                estiamte ← estimate ∩ P
                                synchronized ← P
                                agreed ← V
                        endif
        endcase
endwhile
```

1. msuspect (*P*):

When a new suspect event occur, the MSL layer sends an m-suspect event message. While at Synchronous Phase, each process will depict the new change, updates its set of reachable processes and sends a *SYMMETRY* message with the

corresponding reachable members. It also removes those processes that become unreachable from its current view estimate set before starting the EE-Phase.

2. mrecv (<*SYMMETRY, V, P*>, *q*):

If a process *p* receives a *SYMMETRY* message with its last version number from a process *q* which belongs to the current estimate set and runs under the same invocation round, *p* removes *q* and all its corresponding reachable set from its current estimate.

3. mrecv (<*SYNCHRONIZE, Vp, Vq, P*>, *q*):

When a process *p* receives its last version number from another process *q*, it adds *q* to the synchronized variable. *p* only exits the S-Phase when it receives a reply to its *SYNCHRONIZE* message from each process in the current estimate, or it m-receives a message from a process in its view that has already entered EE-Phase.

4. mrecv (<*ESTIMATE, V, P*>, *q*)

When a process *p* receives an *ESTIMATE* message from a process *q* that already knows *p*'s current version number and is still part of its current view estimate, *p* adds *q* to its synchronized variable to guarantee the termination of the Synchronization phase.

Procedure EstimateExchangePhaseInit()

This phase handles the modification and installation of the view estimate. View estimates are modified when the sender receives an estimate from a reachable process but different from his own. Consequently, it intersects its estimate with the one received from the sender. This continues until the group coordinator receives equal estimates from all participants; then the installation of the view takes place. Processes do not exit the agreement phase unless a final agreed view is installed.

Procedure EstimateExchangePhaseInit()

installed ← **false** /*set to true when the new view is installed*/

repeat
wait-for *event*
case *event* **of**

1. **msuspect** (*P*):
   foreach r ∈ *(II - P) – reachable* **do** *symset[r]* ← *reachable*
   msend (<SYMMETRY, version, reachable> (II - P) - reachable)
   msend (<ESTIMATE, agreed, estimate>, (II - P) - reachable)
   reachable ← *II – P*
   if (estimate ∩ *P* ≠ *0) then*
   SendEstimate(estimate ∩ *P)*

2. **mrecv** (<*SYMMETRY, V, P*>, *q*):
   if *(agreed[p] = V[p])* or (agreed[q] ≤ V[q] and)*(q* ∈ *estimate)* **then**
   SendEstimate(estimate ∩ *P)*

3. **mrecv** (<*SYNCHRONIZE, Vp, Vq, P*>, *q*):
   version[q] ← *Vq*
   if *(agreed[q]* ≺ *Vq) and (q* ∈ *estimate)* **then**
   SendEstimate(estimate ∩ *P)*

4. **mrecv** (<*ESTIMATE, V, P*>, *q*)
   if *(q* ∈ *estimate)* **then**
   if (p ∉ P) **and** (agreed[p] = V[p] **or** agreed[q] ≤ V[q]) **then**
   SendEstimate(estimate ∩ *P)*
   elseif (p ∈ *P*) **and** (∀*r* ∈ estimate ∩ *P : agreed[r] = V[r]*) **then**
   SendEstimate(estimate - P)
   endif

5. **mrecv** (<*PROPOSE, S*>, *q*):
   ctbl[q] ← *S*
   if *(q* ∈ *estimate)* **and** *CheckAgreement(ctbl)* **then**
   InstallView(UniqueID(), ctbl)
   Installed ← **true**
   endif
6. **mrecv** (<*VIEW, w, C*>, *q*):
   if (C[p].cview.id = cview.id) and *(q* ∈ *estimate)) then*
   InstallView(UniqueID(), ctbl)
   Installed ← **true**
   endif

endcase

unitl *installed*

1. **msuspect (*P*):**

When a new suspect event occur, during EE-Phase, each process will depict the new change, and sends a *SYMMETRY* message with the corresponding reachable members. It also removes the suspected set and resends its new estimate message.

2. **mrecv (<*SYMMETRY, V, P*>, *q*):**

If a process *p* receives a *SYMMETRY* message with its last version number from a process *q* which belongs to the current estimate set and runs under the same invocation round, *p* removes *q* and all its corresponding reachable set from its current estimate and resends its estimate set.

3. **mrecv (<*SYNCHRONIZE, Vp, Vq, P*>, *q*):**

When a process *p* receives *SYNCHRONIZE* message from another process *q*, where *q*'s agreed number is greater than *p*'s current agreed number. It implies that *q* has already installed a view excluding *p* and entered a new phase. Hence, *p* removes the sender process from the current estimate, updates the sender's version number and re-sends the new estimate.

4. **mrecv (<*ESTIMATE, V, P*>, *q*)**

When *p* receives an *ESTIMATE* message from a process *q,* while at EE-Phase, it first checks that they both belong to the same agreed round by checking the agreed number. If *q*'s estimate is not equal to *p*'s estimate, it intersects both estimates and resends another estimate message announcing the new modification.

5. **mrecv (<*PROPOSE, S*>, *q*)**

Now the coordinator role takes place. Whenever it receives a proposal message from a process it stores it in its *ctbl[]* buffer. After that it calls procedure *CheckAgreed()*, which compares between all proposals. If all is equal, a call to

InstallView() procedure is performed where the coordinator locally installs the new view. In addition, it sends a *VIEW* message to the rest of the processes for them to install the agreed view.

6. **mrecv** (<*VIEW, w, C*>, *q*)

Upon receiving the *VIEW* message, each process participating in the current view, calls *InstallView()* and installs the new complete view and sets installed to true implying that it is back to the IDLE state.

Procedure SendEstimate(*P*):

This procedure is used to modify the view estimate and to inform the other processes of the change. Each process sends a new estimate message to all processes in its current view and a proposal message to the group coordinator.

Procedure SendEstimate(*P*)
estimat ← *estimate* – *P*
msend (<*Estimate, agreed, estiamte*>, *reachable* – {*p*})
msend (<*PROPSE, (cview, agreed, estiamte)*>, *Min(estiamte)*)

Function CheckAgreement(*C*):

This function is used by the group coordinator when a process m-receives a propose message. It verifies whether the proposals stored in the coordinator table buffer are in agreement, i.e., all estimates are equal.

Function CheckAgreement(*C*)
return $(\forall q \in C[p].estimate : C[p].estimate = C[q].estimate)$
and $(\forall q, r \in [p].estimate : C[p].agreed[r] = C[q].agreed[r])$

Procedure InstallView(w,*C*)

This procedure handles the installation of the new view. It creates the new view *ID* and composes the new view based on the current estimate

Procedure InstallView(w,C)

msend (<VIEW, w, C>, C[p].estimate – {p})

if ($\exists q, r \in C[p].estimae : q \in C[r].cview.comp \wedge C[q].cview.id \neq C[r].cview.id$) **then**

view ← ((w, view.id), {r | r ∈ *C[p].estimate* ∧ *C[r].cview.id = cview.id* })

else

view ← ((w, ⊥), C[p].estimate)

generate vchg(view)

cview ← (w, C[p].estiamte)

stable ← (view.comp = reachable)

and $\forall q, r \in C[p].estimate : C[p].agreed[r] = agreed[r]$)

NOTE: ⊥ means that view *w* is a complete view

4.7 Termination of PGMS in Jgroup

The Jgroup algorithm terminates in *four* rounds. Such that, if temporary lack of symmetry in the network occurred that may cause reachable members to differ in their perception of failures and reconnections, the agreement phase repeats until all of the servers detect the change in network connectivity and the network stabilizes. So, it terminates in at most 4δ time.

To give an example of a four round algorithm, consider a scenario of a process joining a group:

First Round: The first round starts at S-Phase, where each process *m-sends* a synchronize message to all reachable processes in the group informing each other with the new change.

Second Round: This round will not start unless each process receives a reply for the first message sent. Upon receiving all replies, each process again resends *SYNCHRONIZE* message in case the agreed round number has not been updated.

Third Round: This round will start when all processes are synchronized and ready to enter the Estimate Exchange phase (EE-phase.) Now, each process sends an *ESTIMATE* message including the new process to all processes perceived as reachable, along with a *PROPOSE* message to the group coordinator.

Fourth Round: If all estimates are equal, and the coordinator observes agreement in all proposals received, it installs the new view locally then sends a *VIEW* message to all participants. Otherwise, the agreement algorithm will restart.

4.8 Why Did I Choose Jgroup?

Jgroup is an experimental toolkit implemented by a team of researchers in the University of Bologna, Italy. I decided to use Jgroup since it is the first toolkit that applies partitioning and re-merging in the group communication paradigm. It also supports view synchrony while all previous systems only have support to virtual synchrony. In addition, Jgroup uses the facilities offered by Baboglue's partitionable group membership service. The major strengths and weaknesses in the PGMS algorithm implemented in Jgroup are:

4.8.1 View Agreement is a Monotone Decreasing Algorithm

Meaning that one cannot start a new agreement phase unless the current agreement phase has terminated. This means that processes can be added to the view estimate only at the beginning of the agreement phase. Once the algorithm has entered the EE-Phase, i.e., while some processes have already initiated agreement towards a new view, meanwhile, some processes may have become m-suspected, those processes cannot participate in a new agreement phase unless the previous phase has already terminated. This implies that some views installed by the algorithm will be considered unstable and the view installed will be obsolete which will require the agreement phase to re-start. However, it is guaranteed to terminate independent of any failure scenarios.

4.8.2 Liveness/Termination

As mentioned earlier, Jgroup guarantees termination for every execution even when network reachability is highly unstable. It tolerates any number of failures and repairs that occur before and during the agreement protocol. In complex failure scenarios, this could imply degraded performance. However, in common failure scenarios the performance of the algorithm is reasonable. Termination is achieved by two actions.

1. The MSL layer does not report any suspect events that would cause processes to be added to the initial estimate.
2. Processes exchanging their estimate messages allow for the removal of processes that have been removed by others to coincide exactly with those processes that agree on the composition of the next view.

4.8.3 PGMS Algorithm is a 4 Round Algorithm

During the execution of the agreement phase, especially when all estimates are equal, an extra overhead occurs when sending the *ESTIMATE* message to all participating process since, this message will be ignored anyway. Of course this would degrade the performance of the algorithm by increasing the execution time and delaying the execution of the client requests since this won't be handled unless the group status is IDLE. In addition, couldn't the algorithm be enhanced to reach agreement in less number of rounds rather than sticking only to the 4 round algorithm.

4.9 Summary

This chapter presented the structure and specification of the Jgroup model, in addition to its underlying PGMS properties. The group membership service algorithm

is clearly stated showing that in order to install a final agreed view, the algorithm should go through two main phases, the Synchronize phase, *S-Phase*, and the Estimate Exchange phase, *EE-Phase*. The algorithm terminates in four execution rounds. From the strengths and weaknesses mentioned in the chapter, one can find that, Jgroup is still an evolving project that can still be enhanced and improved.

Chapter 5

5 A Client-Server Oriented Group Membership Algorithm in WANs

5.1 Introduction

This algorithm is a novel group membership service designed explicitly for wide-area networks [KSMD98, KSMD99]. It provides clients with full virtual synchrony semantics. I have chosen this algorithm for further investigation since it provides a novel idea on its group membership service. It attains low message overhead by agreeing on membership within a single execution round in most cases. If this idea could be incorporated into Jgroup, the algorithm would attain better performance.

5.2 The Group Membership Service

The group membership algorithm is designed explicitly for the WAN environment. Hence, a group membership supporting multiple groups in a WAN must try to avoid flooding the network. This is done by, propagating membership updates only to those who need them. Hence, not all processes but, a set of dedicated servers are engaged in the membership process.

This employs a Client-Server oriented approach. A "process-level group membership", is where clients are members of each group. While the "server-level group membership" are those responsible for membership maintenance and not with message transmission within the group. The membership algorithm is implemented in

an asynchronous message-passing environment. Communication is achieved via exchanging messages.

5.3 The Membership Structure

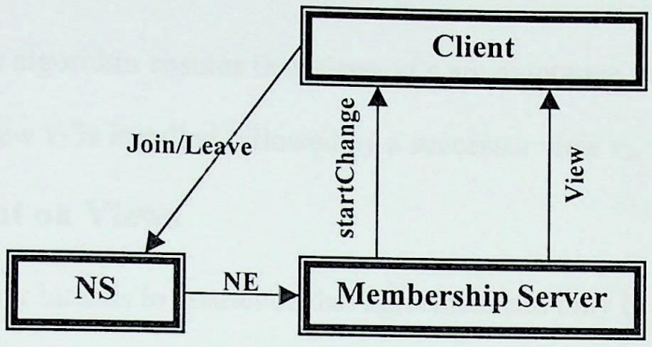


Figure 5.1: The Membership Service Client-Server Architecture

As shown in **Figure 5.1** above, the environment model of the Client-Server architecture consists of a network event notification service, symbolized as *NS*. This service acts as, compared to Jgroup architecture, the failure detector model along with the MSL layer of Jgroup. It reports and detects failure information along with information about processes requesting join or leave. These notifications are reported via notification events (*NEs*). The Membership Server on the other hand, stores those notifications in a variable called *NSView* of a group *G*. Note, *NSView* is not a membership view since it has no unique identifier. The server sends two types of events:

1. ***StartChange(G, StartChangeNum):***

   Indicates to the clients that the server has received a network event and now is engaging in a membership change for group *G*.

2. ***View(G,V):***

   This is the new view of group *G*. Each view should contain the following

<id, members, StartChangeNum>. Where *id* is the unique identifier of the view, *members* are the set of processes participating in the current view, and *startChangeNum* is used to compute the identifier of the new view.

The main group membership properties indicated in this work are:

1. View Identifier Local Monotonicity

By this, the algorithm ensures that views *id*'s are monotone increasing. Meaning that, when a view v_1 is installed followed by a successor view v_2, then $v_2.id > v_1.id$.

2. Agreement on Views

This property intends to guarantee that agreement can only be reached when the network stabilizes and the notification service does not suspect correct and reachable processes. So, suppose a group *G* with a set of clients *CS* and a set of servers *SS*. If there is a time after which the *NSView* of all servers in *SS* contains exactly the same set of clients in *CS* and there is no new suspect events from the *NS*, the view installed by all group members is *V.members* = *CS*. Compared to Jgroup, this property is similar to the **Termination/Liveness** property in Jgroup though not explicitly stated in the PGMS specifications.

5.4 The Membership Algorithm

In this section, we will discuss the implementation of the group membership algorithm of the Client-Server model. The membership algorithm starts when an *NE* is received. Upon receiving the change notification from the notification service, the server sends a *startChange()* message to its clients notifying them that it is starting a new change in agreement. It also multicasts a proposal message to all of the other servers participating in the new view to agree on a new view identifier that is greater

than the previous view installed and to be used when the new view is being installed. The algorithm is divided into three major phases.

5.5 Phases of the Group Membership Algorithm:

1. IDLE Phase

Servers remain at this phase as long as the notification server has not sent any notification of a change in the network stability, process failure, or process join/leave. If this occurred, servers sends a *startChange* message to its clients and starts either the Fast Agreement algorithm or the Slow Agreement algorithm for installing the new view.

2. Fast Agreement

In this phase, agreement is reached in one execution round only. Servers send proposal messages of type *FA*, a tag representing that the server is in the fast agreement phase, to the other servers notifying them of the new change. Upon receiving the proposal message from all servers, where all are of type *FA*, and all are equal, the *startChangeNum* variable will initiate the new view identifier and the new view is installed immediately by each server participating in the current event. However, Fast Agreement algorithm may become a non-terminating algorithm, i.e., the algorithm could be blocked due to the lack of symmetry in the communication system. When this is detected the Slow Agreement algorithm takes over.

3. Slow Agreement

The Slow Agreement in this model is different than that in Jgroup. It is instantiated when the *FA* algorithm is blocked. Similarly, each server sends a proposal message to each other but this time with an *SA* message which indicates that the process has started the slow agreement phase. The Slow Agreement algorithm is

synchronized compared to the Fast Agreement. Since, the set of proposal messages received should have the same proposal number denoted by *propNum.* The algorithm then tests if agreement is reached where it will install a final agreed view and returns back to the *IDLE* state. However, if the *SA* encountered an *NE* from the notification server while running the Slow Agreement, it returns to the *FA* algorithm to avoid installation of obsolete views and sends a proposal message of type *FA*.

The above three phases can clearly be described in the following finite state machine, **Figure 5.2**. A full description of the algorithm specification and implementation is shown in the following section.

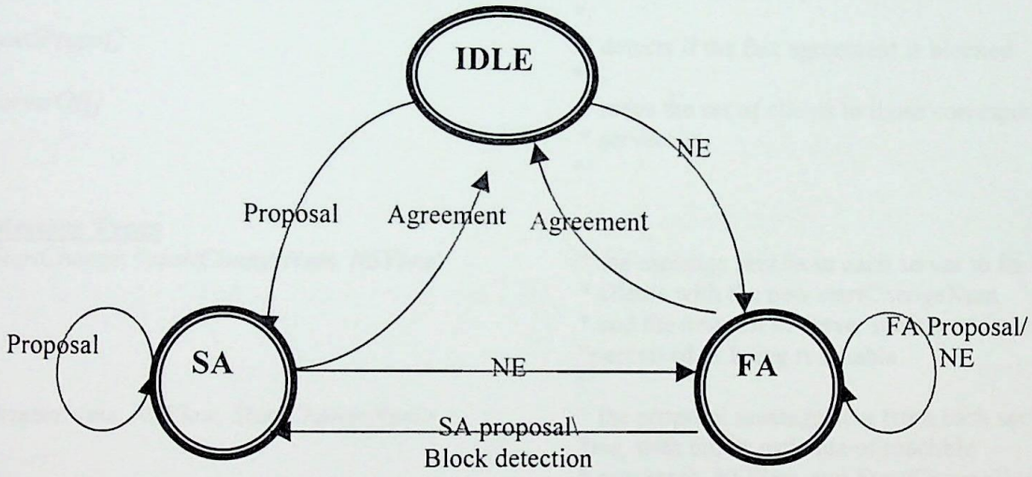


Figure 5.2: Client-Server Membership Algorithm State Diagram

5.6 Membership Algorithm Specifications

The following is the list of variables and message types used in the algorithm:

running	/* used to specify which state or phase is * currently being run * (*IDLE*, *FA*, *SA*) */
NSView[]	/* contains the set of *NE*s received from the * notification service */
props[]	/* is a buffer that stores the recent proposal * messages received from each server */
curView	/* contains the recent view installed */
startChangeNum	/* ensures that every startChange message sent * to clients have a monotonically increasing * number */
propNum	/* ensures that every proposal sent from each * server has a monotonically increasing id * number that is unique */
usedProps[]	/* detects if the fast agreement is blocked */
serverOf[]	/* maps the set of clients to those corresponding * servers */

Message Types

StartChange <startChangeNum, NSView>	/* the message sent from each server to its * clients with the new startChangeNum * and the new list of server members currently *perceived as being reachable. */
propsal<me, NSView, StartChangeNum>	/* the proposal message sent from each server **me,* with the its estiamte of reachble * processes, *NSView*, and *StartChangeNum* * used to calculate the *viewid* */
curView<viewid, NSView, props>	/* the view message sent when final agreement * is reached where a new view will be * installed. Along with the message: * *viewid*: the new view identifier * *NSVIEW*: set of members composing the * new view. **props*: the current proposal. */

5.6.1 The Fast Agreement Algorithm

The Fast Agreement Algorithm is initiated upon receiving a Network Event (*NE*) from the notification service or after receiving a *proposal* message from another server in the group that has already detected the change.

1. On receive NE n:

As shown in the pseudo code below, note that code written in gray will not be used druning fast agreement due to the fact that the algorithm terminates in only one round. So, there is no means of checking whether all proposal messages received within the same membership change are synchronized. When a network event is received, every server in the group sends a *startChange* message with a new *startChangeNum* to the set of local clients informing them that a change has occurred in the membership between servers in the group. Each server also sends a proposal message to all servers perceived as reachable.

On receive NE n:

```
NSView = NSView ∪ n.joining\ n.leaving

If (local(NSView) ≠ {}) then
        startChangeNum = max (curView.id, startChangeNum +1)
        send startChnage <startChangeNum, NSView> to local (NSView)
        running = FA
        propNum = max (propNum, props[serversOf(NSView)].propNum) + 1
        proposal p = <me, NSView, startChangeNum, FA, usedPRops[serversOf(NSView)],
                                                                    propNum>
        send p to serversOf(NSView) \ {me}
        deliver p immediately to myself
endif
```

2. On receive proposal inProp:

Views are not installed in Fast Agreement unless each server receives a proposal message from the set of reachable processes and when all proposals are identical. Hence, upon receiving a proposal message from each server in *NSView*,

the proposal is stored in the *props[]* buffer. If all proposals are identical, using *TestIfAgreementReached()* procedure, the server sends a view message with the new view identifier composed from *startChangeNum* variable. When the view message is sent, the *proposal[]* buffer is set back to *null* and state *running* is set back to *IDLE* state.

On receive proposal inProp:

```
props[inProp.sender] = inProp
if (inProp.members = NSView) then
        if (TestIfSAProposalNeeded (inProp)) then SendSAProposal (inProp) endif
        if (TestIfAgreementReached()) then
                curView = <max (props[serversOf (NSView)].startChangeNum) + 1,
                                NSView, props[serversOf (NSView)].startChangeNum>
                forall s ∈ serversOf (NSView)
                        usedProps[s] = props[s].propNum
                        props[s] = null
                endfor
                running = none
                send curView to local (NSView)
        endif
endif
```

TestIfAgreementReached()

This function loops on all servers in the *NSView*, and compares between the members in each proposal in *props[]* buffer with the reachable set of *NSView*. If all proposals are identical and equal to *NSView*, the function returns true.

TestIfAgreemnetReached()

```
if ( running = FA) then
        return (∀s ∈ serversOfNSView() : props[s].members = NSView ∧  props[s].type = FA
else
        return (∀s ∈ serversOfNSView() : props[s].members = NSView ∧  props[s].type = SA ∧
                                        props[s].propNum = propNum)
endif
```

5.6.2 The Slow Agreement Algorithm

The Slow Agreement algorithm starts whenever the Fast Agreement algorithm blocks. Blocking means that termination is not reached due to transient conditions in the communication systems. This would cause the existence of obsolete proposals. This is due to the fact that the fast algorithm is only a single round algorithm. Hence, there exists no possibility of ensuring that all messages received are synchronized by having an agreed round number, for instance, as provided in Jgroup. Hence, the proposals will not be stable and the Fast Agreement algorithm will not terminate.

When this is detected via the function *TestIfSAPropsalNeeded()*, the Slow Agreement algorithm takes place. Different from Jgroup, if a network event (*NE*) occurred at that time, the membership algorithm is re-started and the previous invocation is totally ignored.

TestIfSAPropsalNeeded(*proposal inProp*)

```
if (running ≠ SA ) then
        return (running = none ∨ inPRop.usedProps[me] = propNum ∨ inProp.type = SA)
else
        return (propNum < inProp.propNum)
endif
```

Slow agreement starts when a process sends a proposal message of type *SA*. This time all proposals are synchronized by ensuring that all proposals received carry the same *propNum*. A process can either initiate the slow agreement or join it.

1. Servers Initiating Slow Agreement

If this is the case, a new slow agreement round will be initiated due to *FA* block detection. The *propNum* is set to be greater than any *propNum* of any proposal message the process has previously sent.

2. Servers Joining Slow Agreement

Those servers are the one that do not currently run the Slow Agreement algorithm but received a proposal message of type *SA*. The installation of views will occur at the same round using the same *propNum* of its last proposal message sent. Once all proposals are checked to be identical, the new agreed view will be sent and *SA* algorithm will terminate.

SendSAPropsal (propsal inProp)

```
startChangeNum = max (curView.id, startChangeNum + 1)
send startChange <startChangeNum, NSView> to local(NSView)
running = SA
if ( inProp.type = FA) then
        propNum = max (propNum + 1, props[serversOf(NSView)].propNum)
else
        propNum = max (propNum, props[serversOf(NSView)].propNum)
endif
proposal outProp = <me, NSView, startChangeNum, SA, usedProps[serversOf(NSView)],
                                                                    propNum>
send outPtop to serversOf(NSView) \ {me}
deliver outProp immediately to myself
```

TestIfAgreementReached()

This function again loops on all servers in the *NSView*, but in *SA* it not only compares between the members in each proposal in *props[],* but also ensures that all servers agree on a common *propNum*. If both conditions are satisfied, the function returns true and *SA* will terminate.

TestIfAgreemnetReached()

```
if ( running = FA) then
        return ( ∀s ∈ serversOfNSView() : props[s].members = NSView ∧  props[s].type = FA
else
        return (∀s ∈ serversOfNSView() : props[s].members = NSView ∧  props[s].type = SA ∧
                                         props[s].propNum = propNum)
endif
```

5.6.3 Combining Both Algorithms

Joining the specification of both algorithms, refer to the finite state machine, **Figure 5.2**, the algorithm works as follows: When a server receives a *NE* from the notification service while at *IDLE* state, the Fast Agreement algorithm is initiated. Server sends a proposal message of type *FA* to all reachable servers, and waits for response. Upon receiving the proposals, the algorithm will detect whether *SA* proposal needs to get started. If *SA* is needed, the server resends another proposal message to all participants of type *SA*. In both cases, the server awaits for responses from all servers, and if all proposals are checked to be the same, the new view is installed and the server returns back to *IDLE* state. As mentioned earlier, if a new *NE* occurred while running the *SA*, the Fast Agreement algorithm is restarted avoiding the installation of obsolete views.

5.7 Termination of the Membership Algorithm

The Client-Server model was designed to minimize the number of exchanged messages among servers. In most cases, when a change occurs and the membership algorithm is initiated, the algorithm terminates in one round of execution, i.e., in a δ time latency. If instability still persists, the algorithm terminates within at most 3δ time of latency after the network is stabilized.

5.8 Why Did I Choose the Client-Server Model?

Though the Client-Server model does not follow all the specifications and properties of PGMS specified earlier, it provides a novel idea in decreasing the number of message exchange among servers while at agreement phase. Thus, this would allow for a faster agreement to be reached which yields to a faster termination

of the membership service. Hence, the client requests could be served faster since it blocks as long as servers are running the membership service. Different from Jgroup, the following presents the major strengths and weakness in the algorithm,

5.8.1 View Agreement Avoids Obsolete Views, but, is Not a Monotone Decreasing Algorithm

Unlike Jgroup, the algorithm does not block when the Slow Agreement algorithm starts. Meaning, if a new network event occurred while at *SA*, the algorithm re-starts the fast algorithm without completion of the previous transaction. This would prevent installation of unstable views. However, this implies that installation of views will not be reached unless the network is completely stable. This of course may not guarantee termination. In Jgroup, each invocation of the agreement protocol ends with the installation of a new view, even if network stability is highly unstable.

5.8.2 The Algorithm is Not Synchronized at Fast Agreement

Although agreement can be reached in only one round of execution, yet this has a drawback in the impossibility of synchronizing the received proposals before installing the new view. This may lead obsolete proposals to exist that causes the Fast Agreement algorithm to block. The reason why this is not done, is that it would require another execution round. Jgroup, on the other hand, ensures that all proposals received have the same agreed round number and thus avoid the possibility of having obsolete proposals.

5.8.3 Execution in Less Number of Rounds

This is one of the major advantages of this algorithm, which Jgroup lacks. If Jgroup could reach agreement in less number of rounds following that idea, yet avoiding the drawbacks encountered in the Client-Server model and still preserving the PGMS properties, this would be a major enhancement in the Jgroup model. This is my contribution in the next coming chapters.

5.9 Summary

To summarize both chapters 4 & 5, the following comparison table **Table 5.1** shall pinpoint the difference between the group membership algorithm in Jgroup and the Client-Server models.

	Jgroup	**Client-Server Model**
Finite State Machine	(Figure 4.2)	(Figure 5.1)
Termination under unstable conditions	Terminate	No-Termination
Termination under stable conditions	Terminate	Terminate
Number of execution rounds	4 rounds in all cases	1 round in most cases 3 rounds in case of instability
Unstable Views	Yes	No
Obsolete Proposals	No	Yes
Synchronization (Agreed round number)	Guaranteed	Not guaranteed in FA Guaranteed in SA

Table 5.1: Comparison Table between the Membership Algorithm in Jgroup & the Client-Server Models

Chapter 6

6 Enhanced Jgroup Algorithm I

6.1 Algorithm Description

This algorithm is designed to show that Jgroup algorithm can still perform efficiently but in less execution time. The membership algorithm in Jgroup was designed in such a way to allow every process to enter both phases the Synchronization Phase and the Estimate Exchange Phase. In addition, each process must start by sending a *SYMMETRY* message to all processes it views as reachable when it joins the group; notifying them, that they will be part of its estimate set when it starts the agreement phase. Followed by, a *SYNCHRONIZE* message during the S-Phase, then it has to enter the EE-Phase by sending both an *ESTIMATE* and a *PROPOSAL* message, by which a final view will be installed. This process will be repeated, until the view becomes stable where, each process in the current estimate will receive a *VIEW* message from the group coordinator to install the new agreed view. However, it was found that in some situations, especially when the network is stable and no upcoming events occur, the *ESTIMATE* messages sent and received will be ignored, since no change has occurred and all processes have equal estimates.

To give an example of such a situation, consider a scenario of a single process partitioning from a group:

First Round: The first round starts at S-Phase, where each process sends a synchronize message to all reachable processes in the group informing each other with the new change.

Second Round: This round will not start unless each process receives a reply for the first message sent. Upon receiving all replies, each process again resends *SYNCHRONIZE* message in case the agreed round number has not been updated.

Third Round: This round will start when all processes are synchronized and ready to enter the estimate exchange phase (EE-phase.) Now, each process sends an *ESTIMATE* message, excluding the partitioned process, to all processes and a *PROPOSAL* message to the group coordinator.

Fourth Round: If all estimates are equal, which is the case in this example, the *ESTIMATE* message will be ignored. The coordinator observes agreement in all proposals received, and it installs its new view locally then sends a view message to all participants.

From this example, one can find that the algorithm is not flexible enough to allow the exchange of estimate messages only when it is needed, i.e., when the estimates of all participants are not equal. A new algorithm, ***AlgorithmI,*** allows for this criteria to exist by which the number of exchange messages will be reduced and the algorithm will terminate in less execution time. From the following pseudo code, one can depict the main changes in the algorithm.

6.2 Algorithm Specification

The following is the list of additional variables used for the proposed algorithm, ***AlgorithmI***. As well as the message types used.

Global Variables:

```
Recvdprop          /* An integer variable used to detect the number of
                    * received proposals sent during S-Phase
                    */
EqualEstimate      /*a variable used to count all equal estimates
                    * while at synchronous phase
                    */
flag ← true        /*This flag is used to distinguish between proposals
                    * sent to the coordinator while at S-Phase or at
                    * at EE-Phase
                    */
```

As of the messages, a modification to the *SYNCHONIZE* and the *PROPOSAL* messages were made. In the encoded scheme of the *PROPOSAL* message, a flag has been added to distinguish between the proposal messages sent while at S-Phase or at EE-Phase. This is done to avoid the possibility of using a proposal, sent earlier, while another new proposal will be initiated, if the process starts the Estimate Exchange phase. This is expected to occur since the system is asynchronous.

Message Types

```
<PROPOSAL, S>      /*A proposal message is m-sent to the coordinator each
                   *time view changes.
                   *S contains<flag, key, cvid, cview, estimate, tmembers>
                   * flag: indicates if proposals are sent while EE-Phase
                   * or if EE-Phase is not needed to begjn.
                   * cvid: the last view id installed by the process
                   *cview: the last proposal sent by the process
                   *estimate: the new host proposal containing the current
                   *estimate set viewed by the process
                   *tmembers: contains the participants of the new proposal
                   */
```

As of the *SYNCHRONIZE* message, the estimate set perceived by the process can now be viewed in the *SYNCHRONIZE* message. This is important since it allows the algorithm to choose the correct *PROPOSAL* message to be sent. If estimates are equal, and the network is stable, the algorithm would allow a *PROPOSAL* message with flag set to true indicating that there is no need to send an *ESTIMATE* message, otherwise, a *PROPOSAL* with flag set to false will be sent.

<SYNCHRONIZE, V_p, V_q, P>

/* V_p: is the version# of the sender
* V_q: the version# of the destination.
* *P*: represents the <estimate, agreed, symset>
* *symset*: the last symset associated with the destination
* *estimate*: the current reachable set perceived
* *agreed*: the last agreed invocation rounds
*/

Procedure AgreementPhase()

This is the main part of the agreement algorithm which takes the new estimate of the new composed view and continues calling the S-Phase only, while EE-Phase will be called during the *SynchronizationPhaseInit()* procedure.

Procedure AgreementPhase()

repeat
    estimate ← *reachable* /*initializing the next view estimate*/
    version[p] ← *version[p]+1* /*generating a new version number*/
    Synchronization PhaseInit()
until *stable*

Procedure SynchronizationPhaseInit()

Different from the SynchronizationPhaseInit() process in Jgroup, this algorithm allows processes to send a proposal message to the coordinator sending an *ESTIMATE* message. This start when all processes are synchronized, meaning that all participants have updated their version numbers, a common agreed invocation round has been set, and estimates of all processes are equal. Now each process is allowed to send a proposal message with its current perceived estimate. S-phase can now terminate by installing a final agreed view when the coordinator perceives that all proposals are in agreement. However, if estimates are not equal, or a suspect event occurred, processes will initiate an *ESTIMATE* message during EE-Phase.

```
Procedure SynchronizationPhaseInit()

Synchronized ← {p}
Foreach r ∈ estimate – {p} do
        msend (<SYNCHRONIZE, version[r], version[p],estimate, agreed, symet[r]>, {r})

while (estimate ⊈ synchronized) do
        wait-for event
        case event of

                5.  msuspect (P):
                        foreach r ∈ (II - P) – reachable do symset[r] ← reachable
                        msend (<SYMMETRY, version, reachable>,(II - P) - reachable)
                        reachable ← II – P
                        estiamte ← estiamte ∩ reachable

                6.  mrecv (<SYMMETRY, V, P>, q):
                        if (version[p] = V[p]) and (q ∈ estimate) then
                                estimate ← estimate – P

                7.  mrecv (<SYNCHRONIZE, Vp, Vq, P>, q):
                        if (version[p] = Vp) then
                                synchronized ← synchronized ∪ {q}
                                agreed[q] ← Vq
                                equalEstimate<- equalEstimate + 1
                        if (version[q] < vq) then
                                version[q] ← Vq
                                msend (<SYNCHRONIZE, version[q], version[p], symset[q]>, {q})
                        if (estimate = synchronized && q ∈ estimate) then
                                if (estimate = equalEstimate) then
                                        estimat ← estimate – P
                                        msend (<PROPSOSAL, (true cvid, cview, estiamte)>,
                                                        Min(estiamte))
                                endif
                                else
                                        EstimateExchangePhaseInit()

                        endif

                8.  mrecv (<ESTIMATE, V, P>, q)
                        version [q] = V[q]
                        if (q ∉ estimate) then
                                msend (<SYMMETRY, version, estimate>, {q})
                        elseif (version[p] = V[p]) and (p ∈ P) then
                                estiamte ← estimate ∩ P
                                synchronized ← P
                                agreed ← V
                        endif

                9.  mrecv (<PROPOSAL, S>, q):
                        ctbl[q] ← S
                        if (flag=true)                  /*sender is at synchronous phase*/
                                recvdprop ← recvdprop + 1
                                if (recvdprop = esitmate.size and CheckAgreement(ctbl) then
                                        InstallView(UniqueID(), ctbl)
                                        Installed ← true
                                endif
                        endif
        endcase

endwhile
```

3. mrecv (<*SYNCHRONIZE, Vp, Vq, P*>, *q*):

When a process *p* receives its last version number from another process *q*, it adds *q* to the synchronized variable. When *p* receives a reply to its *SYNCHRONIZE* message from each process in the current estimate, it adds those processes to synchronized and checks their perceived estimate set. If all processes have their current estimate matches the process estimate set, a proposal message with flag set to true is sent to the coordinator indicating the possibility of installing an agreed view. However, if estimates are not yet stable, the process will start EE-Phase to handle such instability.

5. mrecv (<*PROPOSAL, S*>, *q*)

Now the coordinator role takes place. Whenever it receives a *PROPOSAL* message from a process it stores it in its *ctbl[]* buffer. After that, it checks if the current status is S-Phase and the *PROPOSAL* message flag is set to *true*, then the message is accepted, else the message will be ignored. The coordinator will allow for the installation of the new view, if the number of received proposals to the coordinator are equal to the number of processes participating in the current estimate, and also if all proposals are the same, by calling procedure *CheckAgreed()*, which compares between all proposals. If both conditions are satisfied, a call to *InstallView()* procedure is performed where the coordinator locally installs the new view. In addition, it sends a *VIEW* message to the rest of the processes to install the new agreed view.

6.3 Performance Results

In this section, we show the performance measurements of *AlgorithmI* with respect to the original Jgroup algorithm. The algorithms were run using SUN

workstations. In each test, we ran a series of membership changes per configuration in which the handle suspect event is alerted of the new change in the membership of the group where a new agreement phase must start. This is done using a simulator that simulates partitioning and re-merging of processes. The changes we have applied are changes in which the membership grew either by joining more processes or merging processes after partitioning. Also, changes where the membership shrank by removing and partitioning processes from the group. As stated earlier, all transactions are executed in four rounds. However, in situations where the network is stable and no upcoming events occurring the algorithm, does not send *ESTIMATE* messages, rather a *PROPOSAL* message is directly sent and a final stable view is installed.

The graph in **Figure 6.1**, represents the average observed time of the membership algorithm with respect to the number of members in the group. The measurements show that, there appears to be a linear increase in the server's response to the different membership changes imposed. As shown, the slope of *AlgorithmI* appears to be smaller than the slope of the original algorithm. This implies that *AlgorithmI* has successfully achieved its purpose in enhancing Jgroup algorithm by decreasing the execution time needed to terminate the membership algorithm and reach final stability.

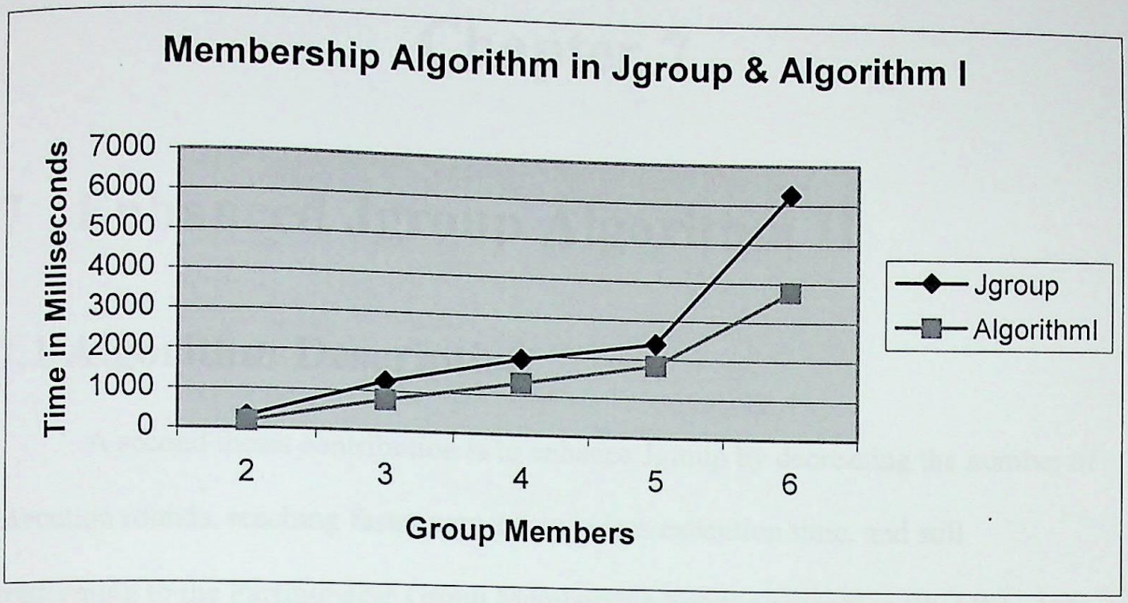


Figure 6.1: Performance Measurements of Partitionable Group Membership Algorithm in Jgroup (Original Jgroup vs. Algorithm I)

6.4 Summary

In this chapter, we described a new enhancement of Jgroup. A new algorithm, *AlgorithmI*, was proposed with the aim to allow Jgroup algorithm to execute in less execution time even if the algorithm still executes like the original Jgroup in four rounds. The algorithm used was described in section 6.2 indicating the changes imposed in the system. Finally, in section 6.3, simulation results were collected from both the original Jgroup algorithm and *AlgorithmI* applying different partitioning and re-merging cases per configuration. The result is shown in **Figure 6.1**, showing that Jgroup performance has been enhanced using *AlgorithmI*.

Chapter 7

7 Enhanced Jgroup Algorithm II

7.1 Algorithm Description

A second thesis contribution is to enhance Jgroup by decreasing the number of execution rounds, reaching faster agreement in less execution time, and still conforming to the Partitionable Group Membership Service properties (PGMS). This is done in *AlgorithmII,* which combines between Jgroup algorithm and the Client-Server model algorithm mentioned in chapter 5. As previously mentioned, Jgroup reaches agreement in maximum of four rounds while the client-server model reaches agreement in one round at most cases and in three rounds when the fast agreement algorithm does not terminate. In order to allow Jgroup to terminate in one round, we would sacrifice the synchronization property. In which, we would allow obsolete messages to exist which is the major drawback found in the client-server model. So, it was found that in order to keep this major criteria stable, the algorithm would better terminate in maximum of two rounds for faster agreement and in four rounds when the network is not stable and estimates are not the same.

From the finite state machine depicted in **Figure 7.1,** one can notice the changes that occurred in the Jgroup algorithm. Now agreement could be reached not only at the Estimate Exchange phase, slow agreement algorithm, but also at Synchronization phase, fast agreement algorithm. The major modifications reside in the S-Phase part of the algorithm to allow it to both detect synchronization of the received messages, which is considered the first execution round, in addition to,

finding all messages received during S-Phase have equal estimates and the algorithm could install an immediate view, the second round, and terminate successfully without instantiating neither the Proposal nor the ESTIMATE messages used during the EE-Phase. The following section is the pseudo code description and specification of the enhanced Jgroup ***AlgorithmII***.

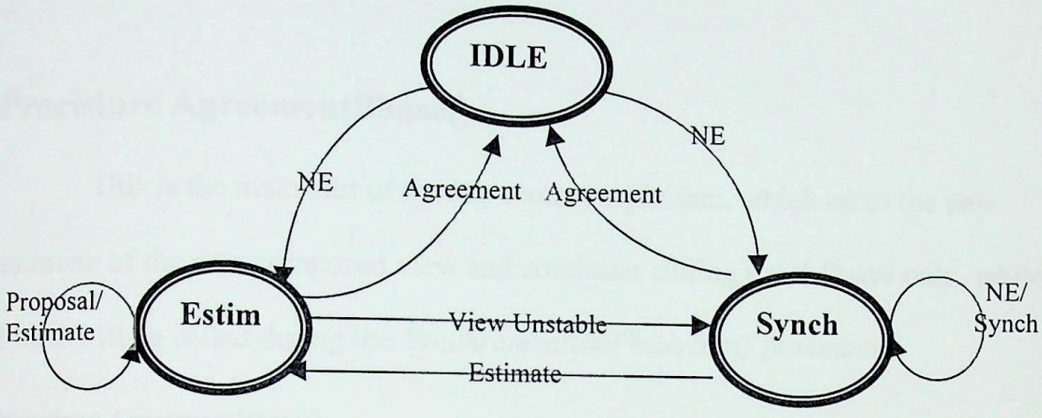


Figure 7.1: Enhanced Jgroup Algorithm State Diagram

7.2 Algorithm Specifications

In this section, a full description of the enhanced Jgroup algorithm is presented. We first start with the new variables needed along with the changes in the message types structure.

Global Variables:

```
equalEstimate          /* an integer variable that is incremented when the
                        * estimate received from sender is equal to estimate of
                        *receiver while at Synchronous Phase.
                        */
```

In addition, the structure of the *SYNCHRONIZE* message has been modified to allow sending along with the sender's & receiver's version numbers and symmetry set, its current perceived estimate and the agreed invocation round numbers.

Message Types

<*SYNCHRONIZE*, V_p, V_q, *P*>

```
/*used during S-Phase. Vp is the version# of the sender
*Vq is the version# of the destination. P represents the
*<symset, estimate, agreed> where
* symset: the last symset associated with the destination
* estimate: the current reachable set of processes
* agreed: the last agreed invocation round numbers
*/
```

Procedure AgreementPhase()

This is the main part of the Agreement algorithm, which takes the new estimate of the new composed view and continues calling the S-Phase only, while EE-Phase will be called during the *SynchronizationPhaseInit()* procedure.

```
Procedure AgreementPhase()

repeat
        estimate ← reachable          /*initializing the next view estimate*/
        version[p] ← version[p]+1     /*generating a new version number*/
        Synchronization PhaseInit()
until stable
```

Procedure SynchronizationPhaseInit()

Different from the **SynchronizationPhaseInit()** process in Jgroup and in ***AlgorithmI***, this algorithm allows processes to send a single *SYNCHRONIZE* message while at S-Phase rather than sending two messages, one to update its version number at the receiver side and then follows another message to update its agreed round number. This is achieved by including the agreed round number list of each process in the encoded *SYNCHRONIZE* message sent. When this message is received in the first round by a process participating in the current change, it not only updates the sender's version number but also updates the sender's agreed round number. A special case might occur, where two SYNCHRONIZE messages must be sent.

Consider a case where process *p* receives a *SYNCHRONIZE* message from process *q* before being notified by the suspect event that a new change has occurred. In this case, only the version number of *q* will be updated until process *p* could enter the Agreement phase. When process *p* sends a SYNCHRONIZE message to *q*, process q finds that its version number has been updated while the agreed number in *p*'s agreed list has not yet been modified. This is the case, where q has to re-send a *SYNCHRONIZE* message, second round, back to *p,* making sure that they both exist in the same invocation round.

When synchronization is reached in the first round, each process checks that the estimate set stored in the SYNCHRONIZE message of every process participating in the current view matches its current estimate. If this is the case, a final view, will immediately be installed by the group coordinator and a view message will be sent, second round, to the rest of the group members participating in the new view. In cases, where the network is unstable, i.e., in situations where the current estimates perceived by the processes are not the same, the Synchronous Phase terminates by calling the Estimate Exchange Phase to handle such instability.

3. **mrecv (**<*SYNCHRONIZE, Vp, Vq, P*>, *q*)**:**

   Upon receiving this message, three cases might occur:

   1. If the sender's version number is less than what the destination process has in its current estimate buffer, it immediately updates both the sender's version and agreed numbers.
   2. If the sender's message contains the current version number of the destination, the destination process updates the sender's agreed number and compares between both estimates

3. If the destination perceives its current version number updated by the sender, but its agreed number has not been updated in the sender's estimate. Sender's version and agreed round number is updated while sending another *SYNCHRONIZE* message to avoid using obsolete messages. Whenever, the coordinator process perceives that all messages are synchronized and all estimates are equal, an immediate view will directly be installed, otherwise EE-Phase begins.

Procedure SynchronizationPhaseInit()

Synchronized ← {p}
Foreach *r* ∈ *estimate – {p} do*
msend (<SYNCHRONIZE, version[r], version[p], symet[r]>, {r})

while *(estimate ⊈ synchronized)* **do**
wait-for *event*
case *event* **of**

1. **msuspect** (*P*):
   foreach r ∈ *(II - P) – reachable* **do** *symset[r]* ← *reachable*
   msend (<SYMMETRY, version, reachable>, (II - P) - reachable)
   reachable ← *II – P*
   estiamte ← *estiamte* ∩ *reachable*

2. **mrecv** (*<SYMMETRY, V, P>, q*):

   if *(version[p] = V[p])* **and** *(q ∈ estimate)* **then**
   estimate ← *estimate – P*

3. **mrecv** (*<SYNCHRONIZE, Vp, Vq, P>, q*):

   if *(version[p] = Vp)* **then**
   synchronized ← *synchronized* ∪ *{q}*
   if *(agreed[q]< Vq)*
   agreed[q] ← *Vq*
   if *(checkEstimate(P))*
   equalEstimate ← *equalEstimate+1*
   endif

```
                if (version[q] < vq) then
                        version[q] ← Vq
                        if (agreed[q] < Vq)
                           agreed[q] = Vq
                        if (version[p] = Vp)
                           msend (<SYNCHRONIZE, version[q], version[p], symset[q]>,
                                                                          {q})
                        else
                           synchronized ← synchronized ∪ {q}
                           if (checkEstimate(P))
                                   equalEstimate ← equalEstimate+1
                        endif
                endif
                if ( estimate.size = synchronized.size)
                        if (estimate.size = equalEstimate)
                                installImmediateView()
                        else
                                EstimateExchagePhaseInit()
                endif

        1.      mrecv (<ESTIMATE, V, P>, q)
                version [q] = V[q]
                if (q ∉ estimate) then
                        msend (<SYMMETRY, version, estimate>, {q})
                elseif (version[p] = V[p]) and (p ∈ P) then
                        estiamte ← estimate ∩ P
                        synchronized ← P
                        agreed ← V
                endif

    endcase

endwhile
```

CheckEstiamte(C)

This procedure is used to compare between the estimate messages received while at Synchronous phase. If sender's estimate is equal to the destination's current estimate, a boolean variable is returned.

```
Procedure CheckEstimate(C)
    return   (∀q ∈ C[p].estimate : C[p].estimate = C[q].estimate)
```

InstallImmediateView()

This process is instantiated only when the process is the group coordinator. The proposal of the coordinator is encoded and stored in the *ctbl[]* buffer, then a call

to the **installView()** process is initiated to send the view message to all processes engaged in the current view.

```
Procedure installImmediateView()
        if (p ∈ estimate && p = coordinator)
                S = encodeImmediatePropsal()
                ctbl[p] = S
                InstallView(UniqueID(), ctbl)
        endif
```

7.3 Performance Results

In this section, we describe the performance measurements of *AlgorithmII* with respect to the original Jgroup algorithm. The same test cases applied in *AlgorithmI* were also applied in this algorithm. It was found that most partitioning and merge cases applied per configuration were resolved by the fast agreement algorithm. However, when instability occur where processes view estimates differently from other processes participating in the same view, the fast agreement algorithm at S-Phase can never terminate in two rounds, in which the estimate exchange phase or slow agreement algorithm must take place. This becomes obvious especially in cases when processes newly join the group.

The graph in **Figure 7.2** shows the average observed time of the slow agreement algorithm and the fast agreement algorithm with respect to the membership changes being reported by the members of the group. The measurements show that, there appears to be a linear increase in the server's response time of the fast agreement algorithm with a small slope compared to the slow agreement algorithm that is also linear but with a larger slope. This indicates of course, that the slow agreement algorithm is slower than the fast agreement algorithm, as expected.

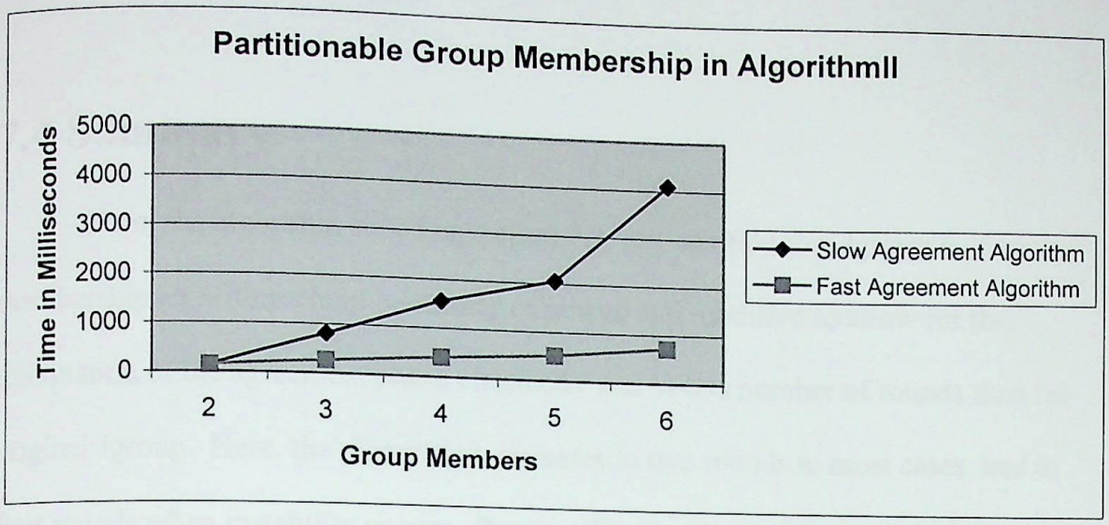


Figure 7.2: Partitionable Group Membership in AlgorithmII

Another comparison is shown in the graph of **Figure 7.3**; it compares between the execution time of *AlgorithmII* with respect to the original Jgroup algorithm. As shown in the figure, the slope of *AlgorithmII* appears to be smaller than the slope of the original algorithm. This implies that *AlgorithmII* was able to execute the same number of membership changes in less execution time compared to original Jgroup.

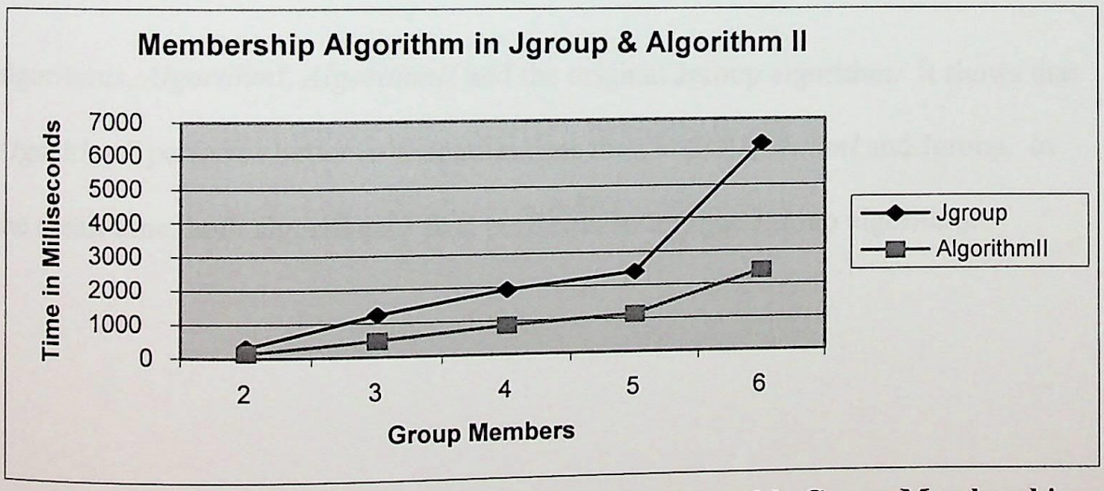


Figure 7.3: Performance Measurements of Partitionable Group Membership Algorithm in Jgroup (Original Jgroup vs. AlgorithmII)

7.4 Summary

From the algorithm stated in section 7.2, one can find that Jgroup's membership algorithm could be further enhanced and modified to allow for the termination of the agreement phase efficiently and in less number of rounds than the original Jgroup. Here, the algorithm terminates in two rounds in most cases, and in four rounds when instability occurs. Proving the correctness of *AlgorithmII* is shown both theoretically and experimentally. A theoretical proof is presented in *Appendix A*, which proves that the algorithm successfully terminates in all cases, either slow or fast, and also abides to the PGMS specifications stated in Jgroup. In addition, The experimental results shown in section 7.3 proved that the algorithm could terminate either in fast agreement when the network is stable or in slow agreement when instability of the current view occurs.

To conclude this chapter, a general overview of the work done can now be emphasized. The following graph in **Figure 7.4**, is a combination of the three algorithms, *AlgorithmI*, *AlgorithmII* and the original Jgroup algorithm. It shows that *AlgorithmII* performs better in execution time than both *AlgorithmI* and Jgroup. In the mean time, both algorithms I & II performs better than Jgroup algorithm.

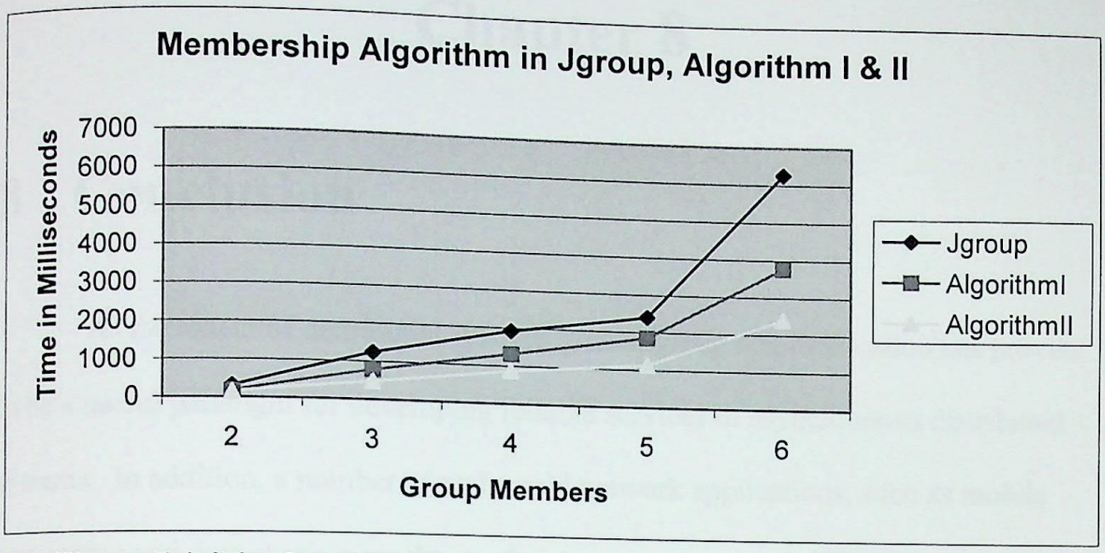


Figure 7.4: Membership Algorithm performance in Jgroup, AlgorithmI, Algorithm II

Another graph in **Figure 7.5**, is a representation of the speed-up of the new contributed algorithms. As shown in the figure, *AlgorithmII* has achieved a maximum speedup of up to 61.6% better than the original algorithm while *AlgorithmI* has achieved a maximum speedup of up to 45% of the original Jgroup.

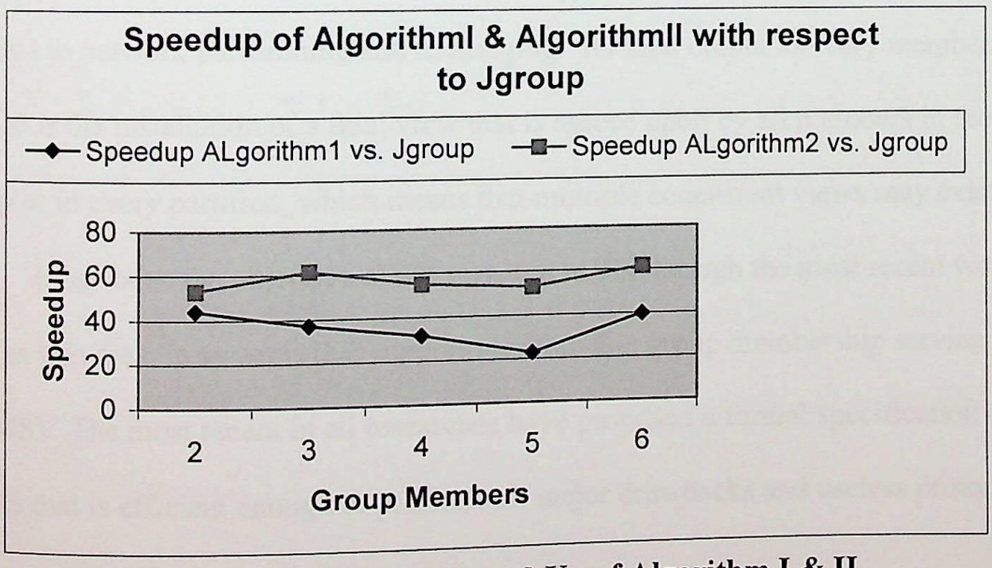


Figure 7.5: Maximum Speed-Up of Algorithm I & II

Chapter 8

8 Conclusion

As the extent of distributed systems grows, group communication has proven to be a useful paradigm for developing reliable services in asynchronous distributed systems. In addition, a number of real world network applications, such as mobile computing and data sharing, might involve cooperation among different sites. This implies that, those applications, by nature, should respond to network partitions and failures by determining its performance level and its QoS. *Partition-aware* applications are those that are capable of making progress in multiple concurrent partitions without blocking [BDMR97]. Hence, systems that support group communication under partitionable environment must support a reliable partitionable group membership service. PGMS should not only support membership changes among members in the same group like server join/leave or crash, but should also respond to network partitioning and re-merging. Its final output to every membership change is the installation of a final view that is agreed upon by all members in the group or in every partition, which means that multiple concurrent views may exist.

It has been the objective of our research to dig through the most recent works done in this area, in systems that support partitionable group membership service (PGMS). The most recent in all researches have proposed a formal specification of PGMS that is efficient enough to preclude all major drawbacks and useless protocols that were found in earlier works.

8.1 Research Goals

8.1.1 The Jgroup Reliable Distributed Object Model vs. the Client-Server Oriented Group Membership Algorithm

In chapters 4 & 5, we presented the main system structure of both models with a thorough study of their group membership algorithms along with their major strengths and weaknesses.

8.1.2 Enchancement on Jgroup, AlgorithmI

This algorithm, presented in chapter 6, is implemented inside Jgroup system model with the aim to enhance the performance of Jgroup. Performance results for this algorithm showed that *AlgorithmI* has successfully reached its aim and Jgroup has executed different membership changes that occur in the group in less execution time than the original algorithm.

8.1.3 Enhancement on Jgroup, AlgorithmII

Chapter 7 presented the second algorithm contributed to enhance the performance of the group membership service in Jgroup model. This algorithm is a combination of ideas presented in the original Jgroup and the client-server model. However, it successfully was able to avoid the drawbacks found in those aforementioned models. The following table, **Table 8.1**, is a comparison table that compares between the new contribution for Jgroup membership service with respect to the original Jgroup and the Client-Server systems.

	Jgroup	Client-Server Model	Enhanced Jgroup *AlgotithmII*
Finite State Machine	(Figure 4.2)	(Figure 5.2)	(Figure 7.1)
Termination under unstable conditions	Terminate	No-Termination	Terminate
Termination under stable conditions	Terminate	Terminate	Termiante
Number of execution rounds	4 rounds in all cases	1 round in most cases 3 rounds in case of instability	2 rounds in most cases 4 round in case of instability
Obsolete Views	Yes	No	Yes
Obsolete Proposals	No	Yes	No
Synchronization (Agreed round number)	Guaranteed	Not guaranteed in FA Guaranteed in SA	Guaranteed

Table 8.1: Comparison table between the membership algorithm in Enhanced Jgroup, Jgroup and the Client-Server models

From this table and from the performance results shown in chapter 7, Jgroup algorithm has been enhanced using *AlgorithmII*. The fast agreement algorithm was successfully able to terminate membership changes in two rounds while the slow agreement handles situations in four execution rounds when network instability or lack of symmetry among group members exist. Obviously the fast agreement algorithm is faster than the slow agreement algorithm. In addition, the overall performance of *AlgorithmII* has proven to be faster that the original Jgroup algorithm.

8.2 Future Work

Group membership maintenance is still an on going research area where lots of work could be done. As mentioned earlier in the group communication mechanisms, there must exist a message delivery semantic that handles the delivery of messages to client applications. These semantics complement the membership service by integrating the multicasted messages with view changes installed by the group

membership service. In Jgroup, *view synchrony* semantic is supported, since it acquires a more relaxed structure with flexible ordering semantics to allow systems to handle partitioning and re-merging more efficiently. A future work is to apply a system supported application to test the performance of message delivery latency to the client objects using the new group membership algorithms proposed.

Appendix A

Proof of Correctness of Algrotihm II

In this Appendix, we verify the correctness of the second algorithm of the thesis contribution. In addition, we show that the second algorithm still conforms to the PGMS properties stated earlier. Note, we are adding to the existing proof done by [BDM99] to illustrate the correctness of the PGMS algorithm.

In [BDM99], the proof started by proving that the algorithm terminates successfully, since the termination property is fundamental and it will be used in the proofs of the PGMS specifications. So, the proof is divided into two main parts:

(i) If a correct process enters S-Phase of a view *v*, it will eventually enter EE-Phase of *v* itself.

(ii) If a correct process enters EE-Phase of a view *v*, it will eventually install a view after *v*.

Proof of the second part is needed to proof the first part.

Now, since the algorithm has been modified to allow for installation of views during the S-Phase, the proof will slightly be modified to the following:

(i) **Lemma D.3:** If a correct process enters S-Phase of a view *v*, it will either:

1. Install a view after *v*, or
2. Enter EE-Phase of *v* itself.

(ii) **Lemma D.2:** If a correct process enters EE-Phase of a view *v*, it will eventually install a view after *v*.

Lemma D.3-1 *if a correct process enters S-Phase of a view v, it will install a view after v.*

PROOF: Suppose p installs a bounded number of views and v is the last view installed by p. Let <*SYNCHRONIZE*, V_p, V_q, *agreed, estimate*> be the synchronize message sent by p (this message must exist since p must m-send at least one *SYNCHRONIZE* message when starting the S-Phase to all processes reachable in its current estimate set.) If all processes in P_p (where P_p denotes *p.esitmate*) are not m-suspected and p is correct, then every message sent by p during v to P_p set will be m-received (see property A.1(a) – p. 26)

Now we need to prove the following:

Claim: *Each process* $r \in P_p$ *will enter S-Phase during* v_r

Suppose this claim is false by contradiction. This means that r will never receive a synchronous message from p or it remains at IDLE State. If r has joined the group and a network event occurred that caused a process to be added or removed, r will eventually detect the change, updates its estimate set and send a *SYNCHRONIZE* message to all processes perceived as reachable after starting the S-Phase, a contradiction. However, if r is already at S-phase, and both p and r are reachable and correct. Even though r doesn't have p in its estimate set, but p's synchronous message will be m-received as of (property A.1(e) – p. 26), a contradiction. And this concludes the claim.

Claim: *Each process* $r \in P_p$ *must be synchronized with p.*

Suppose this claim is false. Meaning that p could accept a *SYNCHRONIZE* message from r that is obsolete, i.e., it has an older agreed invocation number that is accepted for the installation of the new view. Note that for each process $r \in P_p$, the version number $V_p[r]$ & the agreed number $A_p[r]$ is generated when S-Phase starts.

Hence, if p receives *SYNCHRONIZE* message from r with p's latest version number, it doesn't add r to synchronized unless it first checks that p's agreed number in r's estimate *(r.estimate)* is equal to p's latest version number, $A_r[p] = V_p$. If this is so, p re-sends another *SYNCHRONIZE* message to r for updating its agreed invocation round with p to become synchronized, a contradiction.

Claim: *S-Phase always terminates*

Suppose this claim is false. This would imply that the agreement algorithm will be blocked and no installation of views will occur, by blocking all processes at S-Phase. Now, from the previous two claims, one can detect that processes that engage in the current estimate of view v_r, are all at S-Phase. In addition, at least one synchronous message will be sent to all processes in each process's estimate. Upon receiving the synchronous messages, processes are added to Synchronized ensuring that they all have the agreed invocation numbers. Hence, for each $r \in P_p$, if p sets them all to Synchronized, this implies that all processes in P_p have equal estimates, by construction. Hence, $P_r \subseteq P_p$ and vice versa. Where, P_r is the last view estimate for each process $r \in P_p$. Since the coordinator is part of the view estimates, the coordinator will observe such an agreement. Thus, the coordinator will m-send a view message to p, which will also m-receive it and install its new view, a contradiction, which concludes the claim. This also concludes the proof of lemma **D.3-1**

Another proof by contradiction of the third claim would be as follows. Suppose, as mentioned earlier, that all processes are blocked at S-phase. Process p receives a synchronize message from all processes in P_p. However, not all estimates in r conform to the estimate set in p. This implies that cases might be encountered where $r \in P_p$, *but* $p \notin P_r$. By construction, the algorithm calls EE-Phase, a

contradiction. This would lead us to the second part of **lemma D.3,** which has been successfully proven in [BDM99]

Corollary D.1 *if a correct process p enters agreement phase during a view v, then it will eventually install a new view after v.*

PROOF: the proof of this corollary is the combination of lemmas D.2 and D.3

Now, we need to prove that the new enhanced Jgroup algorithm follows the PGMS properties stated earlier. The first property to prove is View Accuracy.

In order to prove this theorem, we must prove the following lemma first.

Lemma D.4 *let v be a view installed by a process p, and let t be the time at which p entered s-phase of v. Let q be a process from which p accepts a message that is either*

(a) A <SYNCHRONIZE,-, P> message, where q's current estimate received is such that $p \in P$*, m-sent by q during a view* v_q*. Then q is at S-phase.* If message received such that $p \notin P$*, then q will enter ee-phase of* v_q *after t.*

(b) A message <Estimate, -, P> such that $p \in P$*, m-sent by q during a view* v_q*, then q has entered ee-phase of* v_q *after t.*

PROOF: At the beginning of the s-phase of *v*, *p* m-sends a *SYNCHRONIZE* message containing its current version number. There are two possibilities where *p* would eventually enter the ee-phase.

Consider (a), p will have received a *SYNCHRONIZE* message from each process not m-suspected after the initiation of the s-phase of *v*. By construction, all processes that m-sent the *SYNCHRONIZE* messages are already in S-phase. If all processes are synchronized at p but some processes $r \in P_p$ *have estimates such that* $s.estimate_r \neq s.estimate_p$. Thus, p must start the Estimate Exchange phase, which leads all processes $r \in P_p$ to eventually enter EE-Phase of their view after t.

As of case (b), p enters ee-phase of v after having m-received a *ESTIMATE* message from a process q_1 that is already at EE-Phase. Again q_1 enters EE-Phase after having its estimate set not matching one or more of the recipients estimates while at S-Phase though all are synchronized, or it also m-received an *ESTIMATE* from another process q_2 that has already entered ee-phase. By construction, proved in (BDM99), let t_i denote the time at which q_i m-sends the *ESTIMATE* message; we have the $estimate_{qi}(t_i)$ is contained in $estimate_{qi+1}(t_{i+1})$, for each $i = 1..n-1$. Thus, we have that $p \in estimate_{qi}(t_i)$, for each $i = 1...n$. Moreover, we have $version_{qi}[p](t_i)$ is equal to $version_{qi}+[p](t_{i+1})$, for each $i = 1...n-1$. This implies that $version_{qn}[p]$ is equal to p's version number, and that q has entered ee-phase of its view after t.

Theorem D.1 (View Accuracy) *If there is a time after which process q remains reachable from some correct process p, them eventually the current view of p will always include q.*

PROOF: Let p be a correct process and let q be always reachable from p after time t_0. We must prove that there is a time after which the current view of p always contains q. If p installs a bounded number of views, then the last installed view must contain q or otherwise by ***Corollary D.1,*** p would re-enter the agreement phase. Hence, suppose p installs an unbounded sequence of views, as stated in the paper there are three claims needed. Two of which matches both algorithms,

Claim *There is a time t_p after which no process distinct from p can directly exclude q from its view estimate without excluding p at the same time.*

Claim *There exists a time $t_4 \geq t_p$ after which p cannot directly exclude q (and vice versa)*

As of the third claim, few changes must be stated in the proof.

Claim *There is a time t after which all messages <VIEW,-,C> m-received and accepted by p are such that q belongs to C[p].estimate (and vice versa)*

Let P_p be those processes that participate with p in the agreement of an unbounded number of views. This implies that all processes $r \in P_p$ are correct and have already initiated the agreement phase with p. Suppose that p no longer suspect any of the processes in P_p, and p installs a view after a certain time t. By construction, each process r has entered S-Phase of their views after that time t. Thus all messages <*SYNCHRONIZE*, - ,P> m-received by p during v are such that p contains both or none of p & q where q is a process in P_p. This implies that p never removes q from its view estimate during v. Thus all messages m-received by the group coordinator are equal, where p's message has $q \in [p].estimate$ as well as q's message has $p \in [q].estimate$. So, when p receives its view message, q belongs to *C[p].estimate* and when q receives its view message while at S-Phase, p belongs to *C[q].estimate*, that concludes the claim, as well as part of the view accuracy theorem. A proof of this claim at EE-phase, is proved in the paper by which, View Accuracy theorem concludes.

Theorem D.2 (View Completeness) *If there is a time after which two correct servers are permanently and mutually unreachable, then eventually all views installed by one will exclude the other*

PROOF: we need to prove that there is a time after which p, which belongs to a set Θ, never contains q, which belongs to a set Π-Θ. If p has installed a bounded number of views, the last view installed can never contain q. Otherwise, p has to enter the agreement phase as of **Corollary D.1**. However, if p has installed an unbounded number of views, hence there is still time where p could m-suspect q.

This implies that there is a time t_1 after which p will permanently suspect q stated in (property A.1(a) – p.26). Also, there exist a time $t_2 \geq t_1$ after which p excludes q from its view estimate such that $q \notin s_p.estimate$. If this occurred at the beginning of S-Phase, this would imply that an m-suspect event occurred at p suspecting q. So, p will exclude q from its estimate set by which the new view installed by p after time t_2 will never contain q. The same would occur if p is at ee-Phase; a time will exist $t_2 \geq t_1$ after which all messages <*PROPOSE*, s_p> m-sent by p during a view are such that $q \notin s_p.estimate$. Thus all views installed by p after t_2 will never contain q. This concludes the proof.

Theorem D.3 (View Order) *The order in which processes install views is such that the successor relation is a partial order.*

The same proof, as mentioned, would hold since there have been no change in the way view id's are created. Each successor view w installed must have an Id that is greater than the last view installed by the process. The reverse would never be possible since all views are monotone increasing.

Theorem D.4 (View Integrity) *Every view installed by a process includes the process itself. Formally,*

PROOF: Every process must install its own local view containing only itself whenever the process joins as a new member. However, suppose a new view will be installed by p during S-phase. p can never install a new view unless it receives a view message from the group coordinator, by construction. By property A.1(a), for p to install this view message, it must belong to the estimate set perceived by the

coordinator which must also coincide with p's current estimate. So, if $C[p].estimate = s_p.estimate$, them p will install the new view. That concludes the proof.

Now we need to prove the last property which is View Coherency. We must first prove that a process will eventually enter agreement phase if a process in its current view v never installs v.

Lemma D.5 *If a correct process p installs a view v, then for every process* $q \in \overline{v}$ *either (i) q also install v, or (ii) p will eventually enter agreement phase during v.*

The proof given for this lemma conforms to both algorithms since the agreement phase includes both S-Phase and EE-phase

Theorem D.5 (View Coherency)

(i) ***If a correct process p installs view v, then either all processes in*** $\overline{v}$ ***also install v, or p eventually installs an immediate successor to v.***

The proof follows **Lemma D.2, D.3 and D.5**

(ii) ***If two processes p and q initially install the same view v and p later on installs an immediate successor to v, then eventually either q also installs an immediate successor to v, or q crashes.***

Following the same proof stated, minor changes will occur. In order to proof this theorem, we now must consider processes installing a successor view either during S-Phase or during EE-Phase. In both cases they must follow **Lemma D.3-1 or D.3-2.** Suppose a correct process p changes its view after installing view v. So, p will install a successor view w after v. If another correct process q that participates in view v with process p exist. Then q must

also install a successor view to v. By **Corollary D.1**, it is easy to prove that p will eventually enter agreement phase during v. However, when a time t_1 exist where p stops m-suspecting q, two possibilities might occur,

1. Either p installs a final view w in which q belongs to this view either during S-Phase as in **Lemma D.3-1** or during EE-Phase as in **Lemma D.3-2**, otherwise, both processes have to re-enter the agreement phase as in **Lemma D.5**. The lemma is proved.
2. Or, p still installs an unbounded number of views, where q must re-enter the agreement phase as stated in **Lemma D.5.** Lemma is proved.

(iii) ***When process p installs a view w as the immediate successor to view v, all processes that survive from view v to w along with p have previously installed v.***

This theorem should hold in case views are installed while at S-Phase or at EE-Phase. In which we must prove that if p installs an immediate successor to v where both views contain q, the new view w will only be installed after q has installed v. View $\overline{w}$ is either obtained from either a new m-suspect event while at synchronous phase in which the view estimate would change making the new view different from that of $\overline{v}$. Or if view $\overline{w}$ is different from $\overline{v}$ at EE-phase where *ESTIAMTE* messages are exchanged excluding those processes that are not part of the new view. Since q belongs to both $\overline{v}$ and $\overline{w}$, then the last view of q must be equal to v. That concludes the proof of View Coherency Theorem.

Appendix B

Implementation

Group.java in Algorithm I

```
/*
 * Copyright (c) 1998,1999 The Jgroup Team. All rights reserved.
 *
 */

package jgroup.daemon;

import java.net.InetAddress;
import java.util.Date;

import jgroup.gm.*;
import jgroup.mss.MssHost;
import jgroup.mss.Mss;
import jgroup.general.OrderedList;
import jgroup.general.Queue;
import jgroup.general.Util;
import jgroup.general.Types;
import debug.Debug;



/**
 * The <code> Group</code> class
 *
 * @author  Alberto Montresor
 * @since          Jgroup 0.1
 */

/**
 * The <code> Group</code> class
 * Modified Jgroup, Algorithm I
 * @author  Marwa Mansour
 * @since   Feb 20th, 2000
*/

final class             Group
extends                 OrderedList
implements              Tag
{

//////////////////////////////////////////////////////////////////////
// Fields
//////////////////////////////////////////////////////////////////////
```

```
// General fields
long        start, temp, end;
Mss         mss;          // Mss used to send
int         status;       // Current mode
MssHost[]   rset;         // Current reachable set
HostData    me;           // Information related to this host
int         myaddress;    // My 32 bit address
int         ninstalled;   // Number of views installed so far
int         ncoordinated;// Number of views coordinated so far
int         nsuspects;    // Number of suspect events
HostData    thosts;       // Complete list of hosts interested in
                          //the group
MemberData  tmembers;     // Complete list of members interested
                          //in the group
int         recvdprop;    // an integer variable used to detect
                          //the number of received proposals sent
                          //during S-Phase
boolean     flag;         //this flag is used to distinguish
                          //withit the proposals sent to the
                          //coordinator while at S_Synch or
                          //E_Estimate

// Current view information
long        cvid;         // Complete view identifier
int[]       cview;        // Complete view composition (host
                          //addresses)
long        pvid;         // Partial view identifier
PartialView pview;        // Partial view composition
int         hpos;         // This host position in
                          //<pview>.<hosts>
int         nmessages;    // Number of messages sent so far
int         equalEstimate;    //a variable used to count all
                              //equal estimates


// Agreement phase data
Queue       delayedSent;      // Messages whose sending is
                              // postponed
Queue       delayedRcvd;      // Messages whose delivering is
                              // postponed
Estimate    estimate;         // Hosts in the current estimate
int         nmembers;         // N. of members in the current
                              // estimate
int         nproposals;       // Number of proposals sent
                              //during this view
MsgProp     proposal;         // Current proposal
boolean     msgsStable;       // True if message buffer is
                              //consistent
boolean     viewStable;       // True if view delivered is
                              //consistent
private     int   noAgreement;//checks the number of loops
                              //where no agreement of the
                              //current view has been reached


// Group table links
Group       tnext;            //Next element in the general
                              //group table
Group       tprev;            //Previous element in the
                              //general group table
Group       hnext;            // Next element in the hash group
```

```
                                        // table
    Group       hprev;          // Previous element in the hash
                                //group table


//////////////////////////////////////////////////////////////////////////
// Constructors
//////////////////////////////////////////////////////////////////////////

    /**
     * Creates an empty group used in the GroupTable structure
     */
    Group()
    {
    }


    /**
     * Creates a group identified by the group identifier <code>
     * key</code>.
     *
     * @param   key   group identifier
     * @param   status      initial status of group
     * @param   vid   inital view identifier (both complete and
     * partial)
     * @param   hd          this host additional info
     * @param   md          joining member additional info
     */
    Group(Mss mss, int gid, int status, long vid, HostData hd,
MemberData md)
    {
        // Group identifier initialization
        this.key            = gid;

        // General fields
        this.mss      = mss;
        this.status   = status;
        rset          = new Host[1];
        rset[0]       = hd.host;
        me            = hd;
        myaddress     = hd.host.getAddress();
        ncoordinated = 0;
        ninstalled    = 0;
        nsuspects     = 0;
        recvdprop     = 0;
        flag          = true;
        thosts        = new HostData();
        tmembers      = new MemberData();
        thosts.insert(hd);
        tmembers.insert(md);
        start        =  0;
        temp         =  0;
        end          =  0;
        equalEstimate      =  0;

        // Initialization of the first view
        cvid          = vid;
        pvid          = vid;
        cview         = new int[1];
        cview[0]      = hd.key;
        pview         = new PartialView(hd, md);
```

```
            nmessages     = 0;
            hpos          = 0;

            // Initialization of agreement phase info
            delayedSent   = new Queue();
            delayedRcvd   = new Queue();
            estimate      = new Estimate();
            nmembers      = 1;
            proposal      = new MsgProp();
            msgsStable    = true;
            viewStable    = false;
            noAgreement   = 0;
      }


////////////////////////////////////////////////////////////////////////
// Group management
////////////////////////////////////////////////////////////////////////

      /**
       * Handles Process join when receiving a join request
       */
      void join(int nsuspects, int key, Host src, MssHost[] trset,
MssHost[] nrset, MssHost[] nuset)
      {
            HostData    sender;// Reference to additional sender info

            Debug.println(20, "Group.join: I'm interested");

                                    // DEBUG
            sender = (HostData) thosts.lookup(src.getAddress());
            if (sender == null) {
                  thosts.insert(new HostData(src));
                  src.grouplist.insert(this);
                  if (Util.in(trset, src))
                        if (this.nsuspects < nsuspects) {
                              this.nsuspects = nsuspects;
                              handleSuspect(trset, nrset, nuset);
                        }
            } else if (sender.leaving) {
                  sender.leaving = false;
                  if (status == S_IDLE && Util.in(rset, src)){
                  Debug.println(20, "Group.join: status is S_IDLE
calling synchronizationPhaseInit");
                        synchronizationPhaseInit();
                  }
            }
      }

      /**
       * A leave Method in case of receiving a leave message
       */
      void leave(HostData sender, Host src)
      {

            Debug.println(20, "Group.leave:  I'm leaving  " +
sender.host.getAddress());
            sender.leaving = true;
            switch (status) {                   // IDLE PHASE
                  case S_IDLE:
```

```
                        Debug.println(20, "Group.leave:  Group status
is IDLE going to call S_Phase");
                        synchronizationPhaseInit();
                        break;
                  case S_SYNCH:              // SYNCHRONIZATION PHASE
                        Debug.println(20, "Group.leave:  Group status
is S_SYNCH going to call EE_Phase");
                        estimate.remove(src);
                        if (estimate.hsize <= estimate.nsynchronized)
                              estimateExchangePhaseInit();
                        break;
                  case S_ESTIMATE:           // ESTIMATE EXCHANGE PHASE
                        Debug.println(20, "Group.leave:  Group status
is S_ESTIMATE going to call sendEstimate");
                        estimate.remove(src);
                        sendEstimate();
                        break;
            }

      }

      /**
       * Marwa Mansour on Feb 20, 2000
       * handle msuspect event
       * sending msend(<SYMMETRY, version, reachable>, (II-P)-
reachable)
       */
      void handleSuspect(MssHost[] trset, MssHost[] nrset, MssHost[]
nuset)
      {
            // process m-sends a SYMMETRY message to all reachable
processes since the previous m-suspect event
            Debug.println(20, "**Group.handleSuspect** " +
me.host.getName() + "sends an R_SYMM msg"+ " current version is: " +
me.version);

            mss.msend(R_SYMM, MsgSymm.encode(key, me.version, thosts,
nrset, rset), nrset);

            Debug.print(20,"Group.hanldeSuspect:  current reachable
set updated:");
            rset = trset;
            for (int i=0; i< rset.length; i++)
                  Debug.print(20,rset[i].getName()+" ");
            Debug.println(20, "");
            Debug.println(20, "daemon.Group: current status is: " +
status);
            switch (status) {

                  case S_IDLE:
// IDLE PHASE: Calling SynchronizationPhaseInit()
                        Debug.println(20,
"daemon.Group.HandleSuspect: status is IDLE and going to call
synchronizationPhaseInit");
                        synchronizationPhaseInit();
                        break;

                  case S_SYNCH:

                              // SYNCHRONIZATION PHASE
```

```
//everytime a process p suspects a process q, p removes q from its
view estimate (cannot be revoked during current view.)
// if a new network event occurs, the process that wants to join or
leave or is suspected to have failed will be removed from the current
estimate.  If all estimates are equal, then a proposal will be sent
without starting the EE-Phase.  If temporary inconsistencies occur,
the process will start the estimate exchange phase.

                        Debug.println(20, "Group.HandleSuspect:
status is S_SYNCH");

                        estimate.remove(nuset);
                        if (estimate.hsize <=
estimate.nsynchronized){
                              Debug.println(20, "Group.handleSuspect:
current estimate size: "+ estimate.hsize);
                              Debug.println(20, "Group.handleSuspect:
current estimate.nsynchronized: "+ estimate.nsynchronized);

                              if (estimate.hsize <= equalEstimate){
                                    Debug.println(20, "handlesuspect:
starting Fast Agreement");
                                    FastAgreement();
                              }
                              else{

      Debug.println(20,"handleSuspect:Starting EE_PHASE");
                                    estimateExchangePhaseInit();
                              }
                        }
                        break;

                  case S_ESTIMATE:
                  // ESTIMATE EXCHANGE PHASE:  starting estimate
exchange phase
                        Debug.println(20,
"daemon.Group.HandleSuspect: status is S_ESTIMATE and going to
sendEstimate");
                        estimate.remove(nuset);
                        sendEstimate();
                        break;
            }
      }
////////////////////////////////////////////////////////////////////
// Interactions with members
////////////////////////////////////////////////////////////////////

      void handleLocalDlvrAck(UpcallDlvrAck upcall)
      {
            HostData    sender;     // Original sender of the
                                    //acknowledged message
            HostData    hd;         // Original sender of the
                                    //acknowledged message
            boolean     carryon;    // Boolean flag to interrupt
                                    //loops
            int         i;          // Counter

            Debug.println(20, "Group.handleLocalDlvr: msg.pos " +
upcall.hpos + " " + upcall.mpos);// DEBUG
            sender = pview.hdata[upcall.hpos];
            sender.mack[upcall.mpos] = upcall.mid;
```

```
// Maybe the n. of dlvrd msgs has increased

            if (upcall.mid == sender.delivered+1) {
                  carryon = true;
                  for (i=0; i<sender.mack.length && carryon; i++)
                        carryon &= (pview.mdata[i].leaving ||
sender.mack[i] >= upcall.mid);
                  if (carryon) {
                        sender.delivered = upcall.mid;
                        for (i=0; i<pview.hsize && !carryon; i++) {
                              hd = pview.hdata[i];
                              carryon &= (hd.hack[hpos] ==
hd.delivered);
                        }
                        if (carryon) {
                              msgsStable = true;
                              if (status == S_ESTIMATE) {
                                    Debug.println(20,
"Group.handleLocalDlvrAck:  Group status is S_ESTIMATE going to
sendProposal");

      proposal.encodeMsgsProp(estimate.hdata, estimate.hsize);
                                    sendProposal();
                              }
                        }
                  }
            }
      }

      void handleLocalMcast(UpcallMcast upcall)
      {
            MsgMcast    msg;  // Message to send

            Debug.println(20, "daemon.Group: handleLocalMcast");

                                          // DEBUG
            msg = new MsgMcast(upcall.flag, upcall.getId(),
myaddress, upcall.dlen, upcall.data);
            if (status == S_IDLE){
                  Debug.println(20, "Group.handleLocalMcast: Group
status is S_IDLE going to accept the message");
                  acceptMessage(msg);
            }
            else if (!viewStable)
                  acceptLateMessage(msg);
            else
                  delayedSent.insert(msg);
      }

      void handleLocalPrepareAck(UpcallAck upcall)
      {
            Debug.println(20, "daemon.Group: handleLocalPrepareAck");

                                    // DEBUG

            // TO BE IMPLEMENTED...
      }


      void handleLocalViewAck(UpcallAck upcall)
      {
```

```
            Debug.println(20, "daemon.Group: handleLocalViewAck");

                                                // DEBUG
            pview.mdata[upcall.mpos].lastack = upcall.last;
            viewStable = true;
            for (int i=0; i < pview.msize && viewStable; i++)
                  viewStable = (pview.mdata[i].leaving ||
pview.mdata[i].lastack == ninstalled);
            if (status == S_ESTIMATE){
                  Debug.println(20, "Group.handleLocalViewAck:  Group
status is S_ESTIMATE, going to sendProposal");
                  sendProposal();
            }
      }

      void handleLocalJoin(UpcallGroup upcall)
      {
            Debug.println(20, "daemon.Group: handleLocalJoin");

                                    // DEBUG
            if (tmembers.insert(new MemberData(upcall.lid,
upcall.member))) {
                  nmembers++;
                  if (status == S_IDLE) {
                        Debug.println(20, "Group.handleLocalJoin:
Group status is S_IDLE, going to call synchronizationPhaseInit");
                        synchronizationPhaseInit();
                  } else if (status == S_ESTIMATE) {
                        Debug.println(20, "Group.handleLocalJoin:
Group status is S_ESTIMATE, going to sendProposal");
                        proposal.encodeMembProp(tmembers, nmembers);
                        sendProposal();
                  }
            }
      }

      void handleLocalLeave(UpcallGroup upcall)
      {
            MemberData  md;   // Sender member and additional data

            Debug.println(20, "daemon.Group: handleLocalLeave");

                                    // DEBUG
            md = (MemberData) tmembers.lookup(upcall.lid);
            if (!md.leaving) {
                  md.leaving = true;
                  nmembers--;
                  Debug.println(20, "Group.handleLocalLeave:
nmembers=  " + nmembers);
                  if (status == S_IDLE){
                        Debug.println(20, "Group.handleLocalLeave:
Group status is S_IDLE, going to call synchronizationPhaseInit");
                        synchronizationPhaseInit();

                  }
                  else if (status == S_ESTIMATE) {
                        Debug.println(20, "Group.handleLocalLeave:
Group status is S_ESTIMATE, going to sendProposal");
                        proposal.encodeMembProp(tmembers, nmembers);
                        sendProposal();

                  }
            }
```

```
    }

////////////////////////////////////////////////////////////////////////
// Interactions with remote hosts (group-related messages)
////////////////////////////////////////////////////////////////////////

    /**
     * Handles a mcast invocation received by a remote host; if the
     * host is in Estimate Exchange Phase, the message is buffered;
     * otherwise, if the view of the message corresponds to the
     * current one, the message is locally delivered and an ack
     * message is m-sent.
     *
     * @param   msg                 the m-received message.
     * @param   src                 the sender of the message.
     */
    void handleMsgMcast(MsgMcast msg, Host src)
    {
        msg.print();                // DEBUG
        Debug.println(20,"daemon.Group: handleMsgMcast");
        if (status == S_ESTIMATE) {
            Debug.println(20, "Group.handleMsgMcast(remote):
Group status is S_ESTIMATE, going to insert a delayed received
message");
            delayedRcvd.insert(msg);
        }
        else if (pvid == msg.vid) {
            localDeliver(msg);
            checkAck(pview.hdata[msg.hpos]);
            mss.msend(R_ACK, MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize), pview.hlist, pview.hsize);
        }
    }

    /**
     *
     *
     * @param   msg                 the m-received message.
     * @param   src                 the sender of the message.
     */
    void handleMsgAck(MsgAck msg, Host src)
    {
        HostData    hd;   // Reference to additional host data

        msg.print();      // DEBUG
        Debug.println(20,"daemon.Group: handleMsgAck");
        if    (pvid == msg.vid) {
            if (msg.ack.length == pview.hsize)
                for (int i=0; i<msg.ack.length; i++) {
                    hd = pview.hdata[i];
                    if (hd.hack[msg.hpos] < msg.ack[i]) {
                        hd.hack[msg.hpos] = msg.ack[i];
                        checkAck(hd);
                    }
                }
            else        // DEBUG
                Util.panic("handleMsgAck");
            }
    }
```

```
    /**
     *First of all, the method checks whether (i) the group is
     *known; (ii) the view associated to the message is equal to
     *the current one; (iii) the messge is unknown; (iv) the
     *forwarding sender is still in the estimate. If so, the
     *message is delivered and an ack message is sent. The message
     *is forwarded again to the remaining
     * members.
     * @param         msg                the m-received message.
     * @param         src                the sender of the message.
     */
    void handleMsgDlvrd(MsgMcast msg, Host src)
    {
          msg.print();

          Debug.println(20, "Group.handleMsgDlvrd: msg.hpos: " +
msg.hpos + "hpos" + hpos);
          // DEBUG
          Debug.println(20, "Group.handleMsgDlvrd: Group.pvid: " +
pvid + " msg.vid: " + msg.vid + " msg.mid " + msg.mid +     // DEBUG
                                                          " g: " +
pview.hdata[msg.hpos].hack[hpos]);
                      // DEBUG
          if( pvid == msg.vid && estimate.lookup(src.getAddress())
!= null && msg.mid > pview.hdata[msg.hpos].hack[hpos] ) {
                localDeliver(msg);
                checkAck(pview.hdata[msg.hpos]);
                mss.msend(R_ACK, MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize), pview.hlist, pview.hsize);

                if (status == S_ESTIMATE){
                      Debug.println(20, "Group.handleMsgDlvrd:
Group status is S_ESTIMATE, going to msend R_FORWARD message");
                      mss.msend(R_FORWARD, msg.data,
estimate.hlist, estimate.hsize);
                }
          }
    }

    /**
     *this method is invoked when a proces mreceives a mesage from
     *another process that has observed a change in the current
     * view. the process enters Agreement phase if the message is
     *not obsolete (Latest version)
     * the sender is believed reachable.
     *When a process p receives its last version number from
     *another process q, it adds q to the synchronized variable.
     *When p receives a reply to its
     *SYNCHRONIZE message from each process in the current
     *estimate, it adds those processes to synchronized and checks
     *their perceived estimate set.
     *If all processes have their current estimate matches the
     *process estimate set, a proposal message with flag set to
     *true is sent to the coordinator, by calling the fast
     *agreement method indicating the possibility of installing an
     *agreed view without entering the EE-Phase.
     *However, if estimates are not yet table,
     *the process will start EE-Phase to handle such instability.

     * @param   msg                the m-received message.
     * @param   src                the sender of the message.
```

```
     *  Marwa Mansour Feb 21, 2000
     *
     */

    void handleMsgSynch(MsgSynch msg, Host src)
    {
          HostData    sender;// Reference to HostData object of the
                             //sender

          msg.print();
          Debug.println(20,"daemon.Group: handleMsgSynch");
          if ((sender = (HostData) thosts.lookup(src.getAddress()))
!= null)

                switch(status) {

                case S_IDLE:      // IDLE PHASE
                      Debug.println(20, "Group.handleMsgSynch: IDLE
phase calling synchronizationPhaseInit"); // DEBUG

                      //updates the sender's version# with the
latest version number sent by the SYNCH message.
                      me.ctbl=null;
                      if (sender.version < msg.vsend) {
                            sender.version = msg.vsend;

                            //checks that the sender of the message
belongs to the current reachable set
                            if (Util.in(rset, sender.host))
                                  synchronizationPhaseInit();
                      }
                      break;

                case S_SYNCH:     // SYNCHRONIZATION PHASE
                      Debug.println(20, "Group.handleMsgSynch: S
phase " + sender.version + " " + msg.vsend);    // DEBUG

//it first updates my version with the latest version number sent by
SYNCH message
                      if (me.version == msg.vdest) {

//sets the status of sender's host data to inSynchronized and
increments the number of synchronized elements in estimate
                            Debug.println(20, "Adds " +
sender.host.getName() + " to Synchronized");
                            estimate.setSynchronized(sender);
                            if (sender.agreed < msg.vsend)
                                  sender.agreed = msg.vsend;


                            }
                      if (sender.version < msg.vsend) {
                            sender.version = msg.vsend;
                            Debug.println(20,
"Group.handleMsgSynch: Sending a R_SYCH message");
                            byte[] buf = MsgSynch.encode(key,
me.version, sender.version, estimate.hdata, estimate.hsize,
sender.host.symset);
                            mss.msend(R_SYNCH, buf, src);

                      }
```

```
                        if (sender.inSynchronized &&
estimate.checkEstimate(msg.hosts)){
                                equalEstimate++;
                                Debug.println(20, "sender in
synchronized and estimates are equal, estimate now is" +
equalEstimate);
                        }

                        if (estimate.hsize <=
estimate.nsynchronized){
                                Debug.println(20,
"Group.handleMsgSynch: current estimate size: "+ estimate.hsize);
                                Debug.println(20,
"Group.handleMsgSynch: current estimate.nsynchronized: "+
estimate.nsynchronized);

                                if (estimate.hsize <= equalEstimate){
                                        Debug.println(20,
"handleMsgSynch: Fast Agreement");
                                        FastAgreement();
                                }
                                else{
                                        Debug.println(20,
"handleMsgSynch: EE-Phase");
                                        estimateExchangePhaseInit();
                                }
                        }
                        break;

                case S_ESTIMATE:  // ESTIMATE EXCHANGE PHASE
                        Debug.println(20, "Group.handleMsgSynch: EE
phase ");   // DEBUG
                        if (sender.version < msg.vsend)
                                sender.version = msg.vsend;
                        Debug.println(20, "Group.hanldeMsgSynch:
sender.agreed " + sender.agreed + " msg.vsend " + msg.vsend ); //
DEBUG
                        if (sender.agreed < msg.vsend &&
estimate.lookup(src.getAddress()) != null) {
                                estimate.remove(msg.rset);
                                sendEstimate();
                        }

                } // END switch
        }


        /**
         * If a process p receives a symmetry message with its last
         *version number from a process q which belongs to the current
         *estimate set and runs under the same invocation round, p
         *removes q and all its corresponding
         *reachable set from its current estimate.

         * @param   msg                 the m-received message.
         * @param   src                 the sender of the message.
         */
        void handleMsgSymm(MsgSymm msg, Host src)
        {
              HostData    sender;       // Reference to sender host
                                        //information
```

```
            int   vdest;        // Version number of the destination
            int   pos;          // My position in the list of messages

            msg.print();


            // DEBUG
            Debug.println(20,"daemon.Group: handleMsgSymm");
            sender = estimate.lookup(src.getAddress());
            pos = Util.valueAt(msg.hosts, myaddress);
            if (pos >= 0) {
                  vdest = msg.version[pos];

                  if (status == S_SYNCH) {// SYNCHRONIZATION PHASE
                        Debug.println(20, "Group.handleMsgSymm: S
PHASE");
                        // DEBUG
                        if (me.version == vdest) {
                              Debug.println(20, "Group.handleMsgSymm:
accepted");
                              // DEBUG
                              estimate.remove(msg.rset);
                              if (estimate.hsize <=
estimate.nsynchronized)
                                    estimateExchangePhaseInit();
                        }

                  } else if (status == S_ESTIMATE) {   // ESTIMATE
EXCHANGE PHASE
                        Debug.println(20, "Group.handleMsgSymm: EE
PHASE");                // DEBUG
                        if (sender != null && me.agreed == vdest &&
sender.agreed == msg.vsend) {
                              Debug.println(20, "Group.handleMsgSymm:
accepted");
                              // DEBUG
                              estimate.remove(msg.rset);
                              sendEstimate();
                        }
                   }
             }
      }

      /**
       * if Status is S-Synch:
       *When a process p receives an Estimate message from a process
       *q that already knows p's current version number and is still
       *part of its current view estimate, p adds q to its
       *synchronized variable toguarantee the termination of the
       *synchronous phase.
       *if Status is S-Estim:
       *When p receives an Estimate message from a process q, while
       *at EE-Phase, it first checks that they both belong to the
       *same agreed phase by checking the agreed number.  If q's
       *estimate is not equal to p's
       *estimate, it intersects both estimates and resends another
       *estimate message announcing the new modification.
       *
       * @param   msg                 the m-received message.
       * @param   src                 the sender of the message.
       */
```

```
    void handleMsgEstim(MsgEstim msg, Host src)
    {
          HostData    sender;// Reference to sender information
          int         vdest;// Version number of destination

          msg.print();

          Debug.println(20,"daemon.Group: handleMsgEstim");
          try {
          sender = (HostData) thosts.lookup(src.getAddress());
          vdest = msg.agreed[Util.valueAt(msg.hosts, myaddress)];
          Debug.println(20, "vdest " + vdest);

          // DEBUG
          Debug.println(20, "me.version " + me.version);  // DEBUG

          if (status == S_SYNCH) {       // SYNCHRONIZATION PHASE
                Debug.println(20, "Group.handleMsgEstim: S_SYNCH");
                // DEBUG
                if (sender.version < msg.vsend)
                      sender.version = msg.vsend;

                if (!sender.inEstimate) {
                      byte[] buf = MsgSymm.encode(key, me.version,
sender, estimate.hlist, estimate.hsize);
                      mss.msend(R_SYMM, buf, src);

                } else if (me.version == vdest){
     Debug.println(20, "all in estimate and all in synchronized");
                      estimate.intersect(msg.hosts, msg.agreed);
                      estimateExchangePhaseInit();
                }

          } else if (status == S_ESTIMATE) {   // ESTIMATE EXCHANGE
PHASE
                Debug.println(20, "Group.handleMsgEstim: EE
PHASE");          // DEBUG
                if (sender.inEstimate && checkAgreed(msg.hosts,
msg.agreed)) {
                      estimate.intersect(msg.hosts, null);
                      sendEstimate();
                }
          }
          } catch (Exception e) { Util.panic("handleMsgEstim", e);
}     // DEBUG
      }


      /**
       *the coordinator role takes place
       *if the proposal flag is set to true and all proposals have
       *equal estimates, the view will be installed without sending
       *estimate messages. However, if the proposal received with a
       *flag set to false, this means
       *that processes have entered the EE-Phase, if all have equal
       *estiamtes, a new view will be installed.
       *
       * @param   msg                 the m-received message.
       * @param   src                 the sender of the message.
       */
      void handleMsgProp(MsgProp msg, Host src)
```

```
    {
          HostData    sender;
          // Reference to host information of the sender
          msg.print();      // DEBUG
          if ( status != S_IDLE && (sender = (HostData)
thosts.lookup(src.getAddress())) != null && sender.inEstimate) {
                sender.ctbl = msg;
                Debug.println(20, "msg.flag=  " + msg.flag);
                if (sender == me){
                      nproposals--;
                      if (nproposals<0)
                            nproposals = 0;
                      Debug.println(20, "Group.handleMsgProp:
number of proposals now "+ nproposals);
                }

                if (status == S_SYNCH && (sender != me ||
nproposals == 0) && checkAgreement() && msg.flag) {
                      Debug.println(20,"Group.handleMsgProp: group
status is S_Synch");
                      Createview();

                }

                if (status == S_ESTIMATE && (sender != me ||
nproposals == 0) && checkAgreement() && !msg.flag) {
                      Debug.println(20,"Group.handleMsgProp: group
status is S_ESTIMATE");
                      Createview();
                }
          }
     }

/**
 *creates and installs the new partial or complete view
 */
private void Createview(){
 // Variables containg the new view description (both for message and
local use)
          long        cvid;       // New complete view identifier
          int[]       cvcomp;     // New complete view composition
          int[][]     members;    // Decoded members
          int[]       pvids;      // Local view indexes
          int         pvid; // Index of subview id of this host

          // Variable used to build the view message information
          HostData    scan;       // Used to scan tcomp
          MsgProp[]   props;      // Array of proposals
          Host[]      dlist;      // Destination list of view
                                  // message
          int[][]     prev_vcomp; // Previous complete view
                                  // compositions
          long[]      prev_vid;   // Previous view id
          boolean     partial;    // True if partial views are
                                  //needed
          int         npartial;   // Number of different partial
                                  //views
          long[]      diff_vid;   // Distinct previous view id
          int         len;        // Short for new_vcomp.length
          int         i,j,k;      // Counters
```

```
and decodes the next view

                                                    ncoordinated);
                                                    Prop_hosts();


                                                    MsgProp[len];
                                                    long[len];
                                                    int [len][];
                                                    int [len][];
                                                    Host[len];

                                                    props and dlist
                                                    (HostData) thosts;
                                                    // DEBUG
                                                    println(20, "Group.handleMsgProp: current view
                                                    ");
                                                    (i=0; i < len; i++) {
                                                    scan = (HostData) scan.lookup(cvcomp[i]);

                                                    Debug.print(20, " " + (scan.host.getAddress() &


                                                    props[i] = scan.ctbl;
                                                    dlist[i] = scan.host;
                                                    }
                                                    } catch (Exception e) { Util.panic("Group.handleMsgProp:
                                                    Corrupted proposals", e); }                                    // DEBUG
                                                    println(20, "");

                                                                                    // DEBUG


                                                    // Decodes proposals
                                                    i = 0;
                                                    try {                    // DEBUG
                                                    for (i=0; i < len; i++) {
                                                    prev_vid[i]= props[i].cvid;
                                                    prev_vcomp[i] = props[i].decodeLastProp();
                                                    members[i]= props[i].decodeMembProp();
                                                    }
                                                    } catch (Exception e) {          // DEBUG
                                                    Util.panic("Group.handleMsgProp: Eccezione: len " +
len + ", i" + i + ", props[i]" + props[i], e);// DEBUG
                                                    }
                                                                // Checks whether partial views are needed
                                                    // ??? Si noti che questo puo' essere ottimizzato
                                                    // in modo da essere sicuri che che sia O(n^2)
                                                    partial = false;
                                                    for (i=0; i < len && !partial; i++)
                                                    for (j=0; j < prev_vcomp[i].length && !partial;
j++) {
                                                    k = Util.valueAt(cvcomp, prev_vcomp[i][j]);
                                                    partial = ( k >= 0 && prev_vid[i] !=
prev_vid[k] );
                                                    Debug.println(20, "Group.handleMsgProp:
Partial views needed?? " + partial);
                                                    }

                                                    // Constructs partial view index
                                                    pvids = new int[len];
```

```
            pvid = 0;
            if (partial) {
                  npartial = 0;
                  diff_vid = new long[len];
                  for (i=0; i < len; i++) {
                        for (j=0; j < npartial && prev_vid[i] !=
diff_vid[j]; j++);
                        if (j == npartial) {
                              diff_vid[j] = prev_vid[i];
                              npartial++;
                              Debug.println(20, "Group.jandleMsgProp:
number of partial views now are: " + npartial);
                        }
                        pvids[i] = j;
                        if (cvcomp[i] == myaddress) {
                              pvid = j;
                        }
                  }
                  ncoordinated += npartial;
                  Debug.println(20, "number of views coordinated so
far is " + ncoordinated);
            }

            // Reliable m-send the view message containing the
information
            mss.rmsend(R_VIEW, MsgView.encode(key, cvid, me.ctbl,
pvids, props), dlist);

            // Locally installs the view
            int[] incarns = me.ctbl.decodeHostProp_incarns();
            installView(cvid, cvcomp, incarns, members, pvids, pvid);
      }


      /**
       * Locates the position of this host in the complete view and
       * checks whether the version number in the message is equal to
       * the current one.
       * @param         msg                  the m-received message.
       * @param         src                  the sender of the message.
       */

      void handleMsgView(MsgView msg, Host src)
      {
            int   pos;        // Counter

            msg.print();      // DEBUG
            Debug.println(20,"daemon.Group:  handleMsgView");

            if ((status == S_ESTIMATE || status == S_SYNCH) &&
estimate.lookup(src.getAddress()) != null) {
                  Debug.println(20,"Group.handleMsgView: Group Status
is S_ESTIMATE");
                  pos = Util.valueAt(msg.cvcomp, myaddress);

                  if (pos >= 0 && msg.agreed[pos] == ninstalled+1){
                        installView(msg.cvid, msg.cvcomp,
msg.incarns, msg.members, msg.pvids, msg.pvids[pos]);
                  }
            }
      }
```

```
////////////////////////////////////////////////////////////////////
// Methods related to the total composition of the group
////////////////////////////////////////////////////////////////////

     /*
      * Removes MemberData item which have requested to exit and
      * have been excluded from the current view.
      */
     private void removeLeavingMembers(int gid, int last)
     {
           MemberData  scan; // Used to scan the list of members
           MemberData  prev; // Scan = prev.next

           Debug.println(20,"daemon.Group:  removeLeavingMembers-
calling StandardGM.notifyLeaveAck for member with Gid=  "+ gid);

           prev = (MemberData) tmembers;
           scan = (MemberData) prev.getFirst();
           while (scan != null) {
                 if (scan.leaving && scan.last != last) {
                       scan.member.notifyLeaveAck(gid);
                       prev.remove(scan.key);
                 } else
                       prev = scan;
                 scan = (MemberData) scan.getNext();
           }
     }


     /*
      *
      */

     private void removeLeavingHosts(int key, int last)
     {
           HostData    scan; // Used to scan the list of members
           HostData    prev; // scan = prev.next

           Debug.println(20,"daemon.Group: removeLeavingHosts
sending R_LEAVE MsgGroup");
           prev = (HostData) thosts;
           scan = (HostData) prev.getFirst();
           while (scan != null) {
                 if (scan.leaving && scan.last != last) {
                       mss.msend(Tag.R_LEAVE, MsgGroup.encode(key),
scan.host);
                       prev.remove(scan.key);
                 } else
                       prev = scan;
                 scan = (HostData) scan.getNext();
           }
     }

////////////////////////////////////////////////////////////////////
// Methods related to the current view estimate
////////////////////////////////////////////////////////////////////
```

```
    /*
     * Each process msends a synchronization message containing a
     * version number to those processes it perceives as being
     * reachable, then waits for responses.
     * the S_PHASE lasts until all processes in the vew estimate
     * have replied
     * to the SYNCHRONIZE message of p, or when p m-receives a
     * message from a process in its view estimate that has already
     * entered EE_PHASE
     */
    private void synchronizationPhaseInit()
    {
          HostData    hd;          // Optimization variable

          start = System.currentTimeMillis();
          System.out.println("started S-Phase@:  " + start);
          Debug.println(20, "daemon.Group:
synchronizationPhaseInit");
                                                  // DEBUG
          equalEstimate = 0;
          Debug.println(20, "Set recvd propsals counter from FA to:
"+ recvdprop);

//starts with intializing the current status the proposal flag
// initializes its current estimate set, version and agreed number

          status = S_SYNCH;
          flag = true;
          Debug.println(20, "daemon.Group: Current status now is
S_SYNCH");              estimate.init(thosts, rset);
          me.inSynchronized = true;
          me.version++;
          me.agreed++;
          equalEstimate++;

          Debug.println(20,"Group.S-Phase:My hostData info is:
hostname= " + me.host.getName() + " version= " + me.version + "
agreed= " + me.agreed);

          for (int i=0; i < estimate.hsize; i++)
//The synchronization phase starts by sending a synchronous message
//to all processes in it current estimate

               if ((hd = estimate.hdata[i]) != me) {
                    Debug.println(20, me.host.getName() + " going
to send a SYNCHRONIZE message announcing its new version#" );
                    byte[] buf = MsgSynch.encode(key, me.version,
hd.version, estimate.hdata, estimate.hsize, hd.host.symset);
                    mss.msend(R_SYNCH, buf, hd.host);
               }
          if (estimate.hsize <= estimate.nsynchronized){

// if all messages received are synchronized and equal to my current
// estimate, there will be no need to send estimate messages.
// the a proposal message will directly be sent with flag=true

               Debug.println(20, "Group.S_PHASE: current estimate
size: "+ estimate.hsize);
               Debug.println(20, "Group.S_PHASE: current
estimate.nsynchronized: "+ estimate.nsynchronized);
               FastAgreement();
```

```
        }
    }


    /**
     * This phase handles the modification and installation of the
     * view estimate.  View Estimates are modified when the sender
     * receives an estimate from a reachable process but different
     * from his own.
     * Consequently, it intersects its estimate with the one
     * received from the sender.  This continues until the group
     * coordinator receives equal
     * estimates from all participants; then the installation of
     * the view takes place.  Processes do not exit the agreement
     * phase unless a final agreed view is installed.
     */

    private void estimateExchangePhaseInit()
    {
          Queue       buffer;     // Temporary variable
          MsgMcast    msg;        // MsgDlvrd message
          HostData    hd;

          temp = System.currentTimeMillis();
          System.out.println("started EE-Phase@:  " + temp);

          Debug.println(20, "daemon.Group:
estimateExchangePhaseInit");        // DEBUG
          recvdprop = 0;

          Debug.println(20, "Set recvd propsals counter from FA to:
"+ recvdprop);
          status = S_ESTIMATE;

          flag = false;            //sets the proposal flag to false
          mss.msend(R_ESTIM, MsgEstim.encode(key, me.version,
estimate.hdata, estimate.hsize), estimate.hlist, estimate.hsize);
          proposal.encodeLastProp(cview, cview.length);
          proposal.encodeHostProp(estimate.hdata, estimate.hsize);
          proposal.encodeMembProp(tmembers, nmembers);
          if (msgsStable) {
                proposal.encodeMsgsProp(estimate.hdata,
estimate.hsize);
                sendProposal();
          }
          for (int i=0; i< pview.hsize; i++) {
                buffer = pview.hdata[i].buffer;
                while ( (msg = (MsgMcast) buffer.removeFirst()) !=
null )
                      mss.msend(R_FORWARD, msg.data, pview.hlist,
pview.hsize);
          }
    }


    /*
     * A method responsible for sending the proposal message to the
     * assigned coordinator
     */
    private void sendProposal()
    {
```

```
                Debug.println(20, "sending a proposal message with
flag"+ flag);
                mss.msend(R_PROP, proposal.encode(flag, key, cvid,
pvid), estimate.coord);
                if (estimate.coord == me.host){
                      nproposals++;
                      Debug.println(20, "current number of
proposals are: "+ nproposals);
                }
     }


     /*
      *This procedure is used to modify the view estimate and to
      *inform the other processes of the change.  Each process sends
      *a new estimate message to all processes in its current view
      *and a proposal message to the group coordinator.
      */

     private void sendEstimate()
     {
           HostData    hd;
           Debug.print(20, "daemon.Group: sending Estimate: ");
           // DEBUG
           for (int i=0; i<estimate.hsize; i++)

                // DEBUG
                Debug.print(20,
(estimate.hdata[i].host.getAddress() & 0xFF) + " ");  // DEBUG
           Debug.println(20, "");                     // DEBUG
           if (estimate.echanged) {
                Debug.println(20, "Group.sendEstimate:  estimate
has changed and no agreement has been reached.");
                byte[] buf = MsgEstim.encode(key, me.version,
estimate.hdata, estimate.hsize);
                mss.msend(R_ESTIM, buf, estimate.hlist,
estimate.hsize);
                proposal.encodeHostProp(estimate.hdata,
estimate.hsize);
                sendProposal();
                noAgreement++;
                Debug.println(20, "Group.SendEstimate: NoAgreement=
" + noAgreement);
           }
      }

      /*
       * This function is used by the Group coordinator when a
       * process m-receives a propose message.  It verifies whether
       * the proposals stored
       *in the coordinator table buffer are in agreement, i.e., all
       *estimates are equal.
       */

      private boolean checkAgreement()
      {
           MsgProp     pp;          // Proposal of this host
           MsgProp     pq;          // Proposal of compared host
           int         i,j;         // Counters
           Debug.println(20,"daemon.Group:  checkAgreement");
           if ((pp = me.ctbl) == null){
```

```
                Debug.println(20, "me.ctbl is null");
                return false;
          }
          for (i=0; i < estimate.hsize; i++) {
                pq = estimate.hdata[i].ctbl;
                if (!pp.checkEstimate(pq)){
                      Debug.println(20, " sender has an estimate
not equal to coordinator");
                      return false;
                }
                for (j=i+1; j < estimate.hsize; j++)
                      if
(!pq.checkMessages(estimate.hdata[j].ctbl)){
                            Debug.println(20, "sender has messages
not equal to coordinator");
                            return false;
                      }
          }
          return true;
     }

     /*
      * A method designed to allow sending a proposal message
      * without starting the estimate exchange phase.
      */

     private  void FastAgreement()
     {
          Queue       buffer;           // Temporary variable
          MsgMcast    msg;              // MsgDlvrd message

          Debug.println(20, "I am in Fast Agreement");
          proposal.encodeLastProp(cview, cview.length);
          proposal.encodeHostProp(estimate.hdata, estimate.hsize);
          proposal.encodeMembProp(tmembers, nmembers);
          if (msgsStable) {
               proposal.encodeMsgsProp(estimate.hdata,
estimate.hsize);
               sendProposal();
             }

          for (int i=0; i< pview.hsize; i++) {
               buffer = pview.hdata[i].buffer;
               while ( (msg = (MsgMcast) buffer.removeFirst()) !=
null )
                     mss.msend(R_FORWARD, msg.data, pview.hlist,
pview.hsize);
          }
     }

     /*
      *
      */
     private void checkAck(HostData hd)
     {
          int         min;          // New minimal value in hack
array
          int         i;            // Counter

          Debug.println(20,"daemon.Group: checkAck CHECK IT!!!");
          min = Integer.MAX_VALUE;
```

```
            for (i=0; i < hd.hack.length; i++) {
                  // System.out.print(" " + hd.hack[i]);
                  if (min > hd.hack[i])
                        min = hd.hack[i];
            }
            if (min > hd.hmin) {
                  for (i=hd.hmin; i < min; i++) {
                        // System.out.print("@");
                        hd.buffer.removeFirst();
                  }
                  hd.hmin = min;
            }
      }

      /*
       * compares the agreed list of the current estimate of both my
       *current estimate and the sender's estimate
       */
      boolean checkAgreed(int[] hosts, int[] agreed)
      {
            int   i=0, j=0;         // Counters

            Debug.println(20,"daemon.Group:  CheckAgreed");
            while (i < hosts.length && j < estimate.hsize)
                  if (hosts[i] == estimate.hdata[j].key) {
                        if (agreed[i] != estimate.hdata[j].agreed)
                              return false;
                        i++; j++;
                  }
                  else if (hosts[i] < estimate.hdata[j].key) {
                        i++;
                  }
                  else {
                        j++;
                  }

            return true;
       }


       /**
        *
        */
       private void localDeliver(MsgMcast msg)
       {
            MemberData  md;   // Used to scan <m_vcomp>
            HostData    hd;   // Additional info of the original
sender
            byte[]      data; // Temporary buffer

            Debug.println(20, "daemon.Group: localDeliver");

                              // DEBUG
            data = new byte[msg.len];
            System.arraycopy(msg.data, msg.start, data, 0, msg.len);
            for (int i=0; i<pview.msize; i++) {
                  md = pview.mdata[i];
                  if (!md.leaving){
                        Debug.println(20,"Group.localDeliver: member
is not leaving, sending a deliver notification via StandardGM");
```

```
                        md.member.notifyDlvr(key, msg.vid, msg.hpos,
msg.mid, msg.flag, data);
                }
            }
            hd = pview.hdata[msg.hpos];
            hd.hack[hpos] = msg.mid;
            hd.hack[msg.hpos] = msg.mid;
            hd.buffer.insert(msg);
            msgsStable = false;
        }


        /*
         *
         */
        private void acceptMessage(MsgMcast msg)
        {
            Debug.println(20, "daemon.Group: acceptMessage");

                            // DEBUG
            nmessages++;
            msg.complete(pvid, hpos, nmessages);
            localDeliver(msg);
            checkAck(me);
            // TODO: Mettere il checkack
            mss.msend(R_MCAST, msg.data, pview.hlist, pview.hsize);
        }

        private void acceptLateMessage(MsgMcast msg)
        {
            Debug.println(20, "Daemon: acceptLateMessage");

                                    // DEBUG
            nmessages++;
            msg.complete(pvid, hpos, nmessages);
            localDeliver(msg);
            // TODO: Mettere il checkack
            // TODO: Modificare per spedire solo agli elementi
            //rimasti in lista
            mss.msend(R_FORWARD, msg.data, pview.hlist, pview.hsize);
        }


        /**
         * This procedure handles the installation of the new view. It
         * creates the new view id and composes the new view based on
         * the current estimate
         */

        private void installView(long cvid, int[] chosts, int[]
incarns, int[][] cmembers, int[] pids, int pid)
        {
            int         plen;       // Partial view length
            InetAddress[]     pv_hosts;
            int[]       pv_vindex;
            ViewImpl    view;       // View to be passed to hosts
            HostData    hd;         // Used to scan thosts
            MemberData  md;         // Used to scan m_vcomp
            MsgMcast    scan;       // Used to scan delayed messages
            boolean     stable;     // True if the current view is
stable
```

```
            Debug.println(20, "daemon.Group: installView");

            // Stores complete view information
            this.cvid = cvid;
            cview = (int[]) cview.clone();

            // Computes host position in the current view and removes
info related to
            // hosts and members not participating in the next
(partial) view.
            plen = 0;
            for (int i=0; i < chosts.length; i++)
                  if (pids[i] == pid) {
                        chosts[plen] = chosts[i];
                        incarns[plen] = incarns[i];
                        cmembers[plen] = cmembers[i];
                        if (chosts[plen] == myaddress)
                              hpos = plen;
                        plen++;
                  }

            // Stores current view information
            ninstalled++;
            stable = pview.init(thosts, chosts, plen, ninstalled,
rset, tmembers, cmembers[hpos]);

            // Updates group flags
            msgsStable = true;
            viewStable = false;
            status = S_IDLE;

            // Build a new view
            pvid          = Types.computePartialVid(cvid, pid);
            pv_hosts      = new InetAddress[plen];
            pv_vindex     = new int[plen];
            for (int i=0; i < plen; i++) {
                  hd = (HostData) thosts.lookup(chosts[i]);
                  pv_hosts[i] = hd.host.getInetAddress();
                  pv_vindex[i]      = hd.host.getIndex();
            }
            view = new ViewImpl(key, pvid, pv_hosts, pv_vindex,
incarns, cmembers, hpos);
            Debug.println(20, "Group.installView: Stable view: " +
stable);
                                    // DEBUG
            Debug.print(20, "Group.installView: Vid: " + pvid);
                                                            // DEBUG

            Debug.println(20, " nviewInst: " + ninstalled);

            // DEBUG

            // Deliver the view to non-leaving members
            for (int i=0; i < pview.msize; i++)
                  if ((md = pview.mdata[i]) != null && !md.leaving)
                        md.member.notifyView(key, view, ninstalled,
hpos);

// Remove hosts and members that have request to leave
            removeLeavingHosts(key, ninstalled);
```

```
            removeLeavingMembers(key, ninstalled);

            if (!tmembers.isEmpty()) {
            // The group has not to be removed

                  // Updates HostData information
                  hd = (HostData) thosts.getNext();
                  while (hd != null) {
                        hd.ctbl = null;
                        hd.hack = new int[pview.hsize];
                        hd.mack = new int[pview.msize];
                        hd.delivered = 0;
                        hd.hmin = 0;
                        hd.buffer.clear();
                        hd = (HostData) hd.getNext();
                  }

              // Updates information about messages
                  nmessages = 0;
                  nproposals = 0;
                  while ((scan = (MsgMcast)
delayedSent.removeFirst()) != null)
                        acceptMessage(scan);
                  while ((scan = (MsgMcast)
delayedRcvd.removeFirst()) != null)
                        if (scan.vid == pvid)
                              localDeliver(scan);
                  if (nmessages > 0) {
                        byte[] buf = MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize);
                        mss.msend(R_ACK, buf, pview.hlist,
pview.hsize);
                  }

                  // Checks if the view is stable

                  if (!stable){
                        Debug.println(20, "Group.installView: view is
not stable, calling synchronizationPahseInit");
                        end = System.currentTimeMillis();
                        System.out.println("SA@:  " + (end-start));
                        Debug.println(20,"SA@:  " + (end-start));
                        synchronizationPhaseInit();
                  }
                  else{
                        end = System.currentTimeMillis();
                        System.out.println("FA@:  " + (end-start));
                        Debug.println(20,"FA@:  " + (end-start));
                  }

            }
      }

} // END Group
```

Group.java in Algorithm II

```
/*
 * Copyright (c) 1998,1999 The Jgroup Team. All rights reserved.
 *
 */

package jgroup.daemon;

import java.net.InetAddress;
import java.util.Date;

import jgroup.gm.*;
import jgroup.mss.MssHost;
import jgroup.mss.Mss;
import jgroup.general.OrderedList;
import jgroup.general.Queue;
import jgroup.general.Util;
import jgroup.general.Types;
import debug.Debug;


/**
 * The <code> Group</code> class
 *
 * @author  Alberto Montresor
 * @since   Jgroup 0.1
 */

/**
 * The <code> Group</code> class
 * Modified Jgroup, Algorithm II
 * @author  Marwa Mansour
 * @since   March 15th, 2000
 */

final class             Group
extends                 OrderedList
implements              Tag
{

//////////////////////////////////////////////////////////
// Fields
//////////////////////////////////////////////////////////

      // General fields
      Mss          mss;          // Mss used to send
      int          status;       // Current mode
      MssHost[]    rset; // Current reachable set
      HostData     me;    // Information related to this host
      int          myaddress;  // My 32 bit address
      int          ninstalled; // Number of views
                                 // installed so far
      int          ncoordinated;      // Number of views
                                      //coordinated so far
      int          nsuspects;  // Number of suspect events
      HostData     thosts;            // Complete list of hosts
                                      //interested in the group
      MemberData   tmembers;   // Complete list of members
```

```
                                    // interested in the group  //
                                    Current view information
     long        cvid;          // Complete view identifier
     int[] cview;               // Complete view
                                    //composition (host //addresses)
     long        pvid;          // Partial view identifier
     PartialView pview;         // Partial view composition
     int         hpos;          // This host position in
                                    //<pview>.<hosts>
     int         nmessages;     // Number of messages sent
                                    // so far
     int         equalEstimate;
//an integer variable that is incremented when the estimate received
from sender is equal to estimate of receiver while at Synchronous
Phase

     long        start;         // to initialize start time
     long        end;           // to initialize end time

     // Agreement phase data
     Queue       delayedSent;
// Messages whose sending is postponed

     Queue       delayedRcvd;
// Messages whose delivering is postponed

     Estimate    estimate;   // Hosts in the current estimate
     Estimate    lastEstimate;
// Hosts in the previous estimate

     int         nmembers;
// N. of members in the current estimate
     int         nproposals;
// Number of proposals sent during this view
     MsgProp     proposal;          // Current proposal
     byte[]      msg;
     boolean     msgsStable;
// True if message buffer is consistent
     boolean     viewStable;
// True if view delivered is consistent
     private int noAgreement;
//checks the number of loops where no agreement of the current view
has been reached

     MsgProp     ImmMsg;
// stores the coordinator's proposal Group table links

     Group tnext;
// Next element in the general group table

     Group tprev;
// Previous element in the general grouptable

     Group hnext;
// Next element in the hash group table

     Group hprev;
// Previous element in the hash group table

////////////////////////////////////////////////////////
```

```
// Constructors
//////////////////////////////////////////////////////

    /**
     * Creates an empty group used in the GroupTable structure
     */
    Group()
    {
    }


    /**
     * Creates a group identified by the group identifier <code>
     *key</code>.
     *
     * @param   key         group identifier
     * @param   status      initial status of group
     * @param   vid         inital view identifier (both complete
     *                      and partial)
     * @param   hd          this host additional info
     * @param   md          joining member additional info
     */
    Group(Mss mss, int gid, int status, long vid, HostData hd,
MemberData md)
    {
        // Group identifier initialization
        this.key     = gid;

        // General fields
        this.mss     = mss;
        this.status  = status;
        rset         = new Host[1];
        rset[0]      = hd.host;
        me           = hd;
        myaddress    = hd.host.getAddress();
        ncoordinated = 0;
        ninstalled   = 0;
        nsuspects    = 0;
        thosts       = new HostData();
        tmembers     = new MemberData();
        ImmMsg       = new MsgProp();

        thosts.insert(hd);
        tmembers.insert(md);

        // Initialization of the first view
        cvid         = vid;
        pvid         = vid;
        cview        = new int[1];
        cview[0]     = hd.key;
        pview        = new PartialView(hd, md);
        nmessages    = 0;
        hpos         = 0;
        equalEstimate     = 0;
        tworounds    = 0;
        fourounds    = 0;
        start        = 0;
        end          = 0;
        newRound     = true;

        // Initialization of agreement phase info
```

```
            delayedSent         = new Queue();
            delayedRcvd         = new Queue();
            estimate            = new Estimate();
            lastEstimate        = new Estimate();
            nmembers      = 1;
            proposal      = new MsgProp();
            msg           = new byte[0];
            msgsStable    = true;
            viewStable    = false;
            noAgreement   = 0;
      }




////////////////////////////////////////////////////////////////////////
// Group management
////////////////////////////////////////////////////////////////////////


      /**
       *
       */
      void join(int nsuspects, int key, Host src, MssHost[] trset,
MssHost[] nrset, MssHost[] nuset)
      {
            HostData    sender;           // Reference to additional
sender info

            Debug.println(20, "Group.join: I'm interested");
                        sender = (HostData)
thosts.lookup(src.getAddress());
            if (sender == null) {
                  thosts.insert(new HostData(src));
                  src.grouplist.insert(this);
                  if (Util.in(trset, src))
                        if (this.nsuspects < nsuspects) {
                              this.nsuspects = nsuspects;
                              handleSuspect(trset, nrset, nuset);
                        }
            } else if (sender.leaving) {
                  sender.leaving = false;
                  if (status == S_IDLE && Util.in(rset, src)){
//                      Debug.println(20, "Group.join: status is
S_IDLE calling synchronizationPhaseInit");
                        synchronizationPhaseInit();
                  }
            }
      }

      /**
       * Leave method in case of recieving a leave message
       */
      void leave(HostData sender, Host src)
      {

            Debug.println(20, "Group.leave:  I'm leaving  " +
sender.host.getAddress());
            sender.leaving = true;
```

```
        switch (status) {
              case S_IDLE:       // IDLE PHASE
                    synchronizationPhaseInit();
                    break;
              case S_SYNCH:      // SYNCHRONIZATION PHASE
                    estimate.remove(src);
                    if (estimate.hsize <= estimate.nsynchronized)
                          estimateExchangePhaseInit();
                    break;
              case S_ESTIMATE:  // ESTIMATE EXCHANGE PHASE
                    estimate.remove(src);
                    sendEstimate();
                    break;
        }

    }

    /*
     * ActivateGroups Method
     * handle msuspect event
     * sending msend(<SYMMETRY, version, reachable>, (II-P)-
     * reachable)
     */

    void handleSuspect(MssHost[] trset, MssHost[] nrset, MssHost[]
nuset)
    {
          // process m-sends a SYMMETRY message to all reachable
processes since the previous m-suspect event

          Debug.println(20, "daemon.Group: handleSuspect received
from Mss");       Debug.println(20, "**Group.handleSuspect** " +
me.host.getName() + "send R_SYMM msg"+ "my current version is: " +
me.version);

          mss.msend(R_SYMM, MsgSymm.encode(key, me.version, thosts,
nrset, rset), nrset);

          Debug.print(20,"Group.hanldeSuspect:  current reachable
set updated:");
          rset = trset;
          for (int i=0; i< rset.length; i++)
                Debug.print(20,rset[i].getName()+" ");
          Debug.println(20, "");
          Debug.println(20, "daemon.Group: the group current status
is (IDLE=1, S_SYNCH=2, S_ESTIMATE=3) " + status);
          switch (status) {

                case S_IDLE:       // IDLE PHASE
// if a network event occurred while the process is still at idle
state, the process will immediately start the synchronization phase

                      synchronizationPhaseInit();
                      break;

                case S_SYNCH:      // SYNCHRONIZATION PHASE
//everytime a process p suspects a process q, p removes q from its
view estimate (cannot be revoked during current view.)
// if a new network event occurs, the process that wants to join or
leave or is suspected to have failed will be removed from the current
estimate.  If all estimates are equal, the process will start the
```

```
Fast Agreement algorithm, by calling installImmediateView().  If
instability still persist, the Slow Agreement Algorithm will start,
EE-Phase

                        estimate.remove(nuset);
                        if (estimate.hsize <=
estimate.nsynchronized){
                              Debug.println(20, "Group.handleSuspect:
current estimate size: "+ estimate.hsize);
                              Debug.println(20, "Group.handleSuspect:
current estimate.nsynchronized: "+ estimate.nsynchronized);
                              if (estimate.hsize <= equalEstimate){
                                    Debug.println(20,"going to
install immediate view");
                                    installImmediateView();
                              }
                              else{
                                    Debug.println(20,"After handle
Suspect: starting ee-phase");
                                    estimateExchangePhaseInit();
                              }
                        }
                        break;

                  case S_ESTIMATE:
// ESTIMATE EXCHANGE PHASE:  will start EE-Phase, the slow agreement
algorithm
                        Debug.println(20,
"daemon.Group.HandleSuspect: status is S_ESTIMATE and going to
sendEstimate");
                        estimate.remove(nuset);
                        sendEstimate();
                        break;
            }
      }


////////////////////////////////////////////////////////////////////
// Interactions with members
////////////////////////////////////////////////////////////////////

      void handleLocalDlvrAck(UpcallDlvrAck upcall)
      {
            HostData    sender;
            // Original sender of the acknowledged message
            HostData    hd;
            // Original sender of the acknowledged message
            boolean     carryon;
            // Boolean flag to interrupt loops
            int         i;    // Counter

            Debug.println(20, "Group.handleLocalDlvr: msg.pos " +
upcall.hpos + " " + upcall.mpos);// DEBUG
            sender = pview.hdata[upcall.hpos];
            sender.mack[upcall.mpos] = upcall.mid;
            if (upcall.mid == sender.delivered+1) {
            // Maybe the n. of dlvrd msgs has increased
                  carryon = true;
                  for (i=0; i<sender.mack.length && carryon; i++)
                        carryon &= (pview.mdata[i].leaving ||
sender.mack[i] >= upcall.mid);
```

```
                if (carryon) {
                      sender.delivered = upcall.mid;
                      for (i=0; i<pview.hsize && !carryon; i++) {
                            hd = pview.hdata[i];
                            carryon &= (hd.hack[hpos] ==
hd.delivered);
                      }
                      if (carryon) {
                            msgsStable = true;
                            if (status == S_ESTIMATE) {
                                  Debug.println(20,
"Group.handleLocalDlvrAck:  Group status is S_ESTIMATE going to
sendProposal");

      proposal.encodeMsgsProp(estimate.hdata, estimate.hsize);
                                  sendProposal();
                            }
                      }
                }
          }
      }

      void handleLocalMcast(UpcallMcast upcall)
      {
            MsgMcast     msg;       // Message to send

            Debug.println(20, "daemon.Group: handleLocalMcast");

                                          // DEBUG
            msg = new MsgMcast(upcall.flag, upcall.getId(),
myaddress, upcall.dlen, upcall.data);
            if (status == S_IDLE){
                  Debug.println(20, "Group.handleLocalMcast: Group
status is S_IDLE going to accept the message");
                  acceptMessage(msg);
            }
            else if (!viewStable)
                  acceptLateMessage(msg);
            else
                  delayedSent.insert(msg);
      }

      void handleLocalPrepareAck(UpcallAck upcall)
      {
            Debug.println(20, "daemon.Group: handleLocalPrepareAck");

                                    // DEBUG
            // TO BE IMPLEMENTED...
      }


      void handleLocalViewAck(UpcallAck upcall)
      {
            Debug.println(20, "daemon.Group: handleLocalViewAck");

                                          // DEBUG
            pview.mdata[upcall.mpos].lastack = upcall.last;
            viewStable = true;
            for (int i=0; i < pview.msize && viewStable; i++)
                  viewStable = (pview.mdata[i].leaving ||
 pview.mdata[i].lastack == ninstalled);
```

```
            if (status == S_ESTIMATE){
                  Debug.println(20, "Group.handleLocalViewAck:  Group
status is S_ESTIMATE, going to sendProposal");
                  sendProposal();
            }
      }

      void handleLocalJoin(UpcallGroup upcall)
      {
            Debug.println(20, "daemon.Group: handleLocalJoin");

                                                      // DEBUG
            if (tmembers.insert(new MemberData(upcall.lid,
upcall.member))) {
                  nmembers++;
                  if (status == S_IDLE) {
                        Debug.println(20, "Group.handleLocalJoin:
Group status is S_IDLE, going to call synchronizationPhaseInit");
                        synchronizationPhaseInit();
                  } else if (status == S_ESTIMATE) {
                        Debug.println(20, "Group.handleLocalJoin:
Group status is S_ESTIMATE, going to sendProposal");
                        proposal.encodeMembProp(tmembers, nmembers);
                        sendProposal();
                  }
            }
      }

      void handleLocalLeave(UpcallGroup upcall)
      {
            MemberData  md;   // Sender member and additional data

            Debug.println(20, "daemon.Group: handleLocalLeave");

                                          // DEBUG
            md = (MemberData) tmembers.lookup(upcall.lid);
            if (!md.leaving) {
                  md.leaving = true;
                  nmembers--;
                  Debug.println(20, "Group.handleLocalLeave:
nmembers=  " + nmembers);
                  if (status == S_IDLE){
                        Debug.println(20, "Group.handleLocalLeave:
Group status is S_IDLE, going to call synchronizationPhaseInit");
                        synchronizationPhaseInit();
                  }
                  else if (status == S_ESTIMATE) {
                        Debug.println(20, "Group.handleLocalLeave:
Group status is S_ESTIMATE, going to sendProposal");
                        proposal.encodeMembProp(tmembers, nmembers);
                        sendProposal();
                  }
            }
      }


////////////////////////////////////////////////////////////////////
// Interactions with remote hosts (group-related messages)
////////////////////////////////////////////////////////////////////

      /**
```

```
     * Handles a mcast invocation received by a remote host; if the
     * host is in Estimate Exchange Phase, the message is buffered;
     *otherwise,if the view of the messagecorresponds to the
     *current one, the message is locally delivered and an ack
     *message is m-sent.
     *
     * @param   msg                 the m-received message.
     * @param   src                 the sender of the message.
     */
    void handleMsgMcast(MsgMcast msg, Host src)
    {
        msg.print();                // DEBUG
        Debug.println(20,"daemon.Group: handleMsgMcast");
        if (status == S_ESTIMATE) {
            Debug.println(20, "Group.handleMsgMcast(remote):
Group status is S_ESTIMATE, going to insert a delayed received
message");
            delayedRcvd.insert(msg);
        }
        else if (pvid == msg.vid) {
            localDeliver(msg);
            checkAck(pview.hdata[msg.hpos]);
            mss.msend(R_ACK, MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize), pview.hlist, pview.hsize);
        }
    }

    /**
     *
     *
     * @param   msg                 the m-received message.
     * @param   src                 the sender of the message.
     */
    void handleMsgAck(MsgAck msg, Host src)
    {
        HostData    hd;   // Reference to additional host data

        msg.print();


            // DEBUG
        Debug.println(20,"daemon.Group: handleMsgAck");
        if    (pvid == msg.vid) {
            if (msg.ack.length == pview.hsize)
                for (int i=0; i<msg.ack.length; i++) {
                    hd = pview.hdata[i];
                    if (hd.hack[msg.hpos] < msg.ack[i]) {
                        hd.hack[msg.hpos] = msg.ack[i];
                        checkAck(hd);
                    }
                }
            else

                    // DEBUG
                Util.panic("handleMsgAck");
                }
    }


    /**
```

```
        * First of all, the method checks whether (i) the group is
        * known; (ii) the view associated to the message is equal to
        * the current one; (iii) the messge is unknown; (iv) the
        * forwarding sender is still in the estimate. If so, the
        * message is delivered and an ack message is sent. The message
        * is forwarded again to the remaining members.
        * @param   msg                the m-received message.
        * @param   src                the sender of the message.
        */

      void handleMsgDlvrd(MsgMcast msg, Host src)
      {
            msg.print();                        // DEBUG
            Debug.println(20, "Group.handleMsgDlvrd: msg.hpos: " +
msg.hpos + "hpos" + hpos);
            // DEBUG
            Debug.println(20, "Group.handleMsgDlvrd: Group.pvid: " +
pvid + " msg.vid: " + msg.vid + " msg.mid " + msg.mid +     // DEBUG
                                                          " g: " +
pview.hdata[msg.hpos].hack[hpos]);        // DEBUG
            if( pvid == msg.vid && estimate.lookup(src.getAddress())
!= null && msg.mid > pview.hdata[msg.hpos].hack[hpos] ) {
                  localDeliver(msg);
                  checkAck(pview.hdata[msg.hpos]);
                  mss.msend(R_ACK, MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize), pview.hlist, pview.hsize);

                  if (status == S_ESTIMATE){
                        Debug.println(20, "msend R_FORWARD message");
                        mss.msend(R_FORWARD, msg.data,
estimate.hlist, estimate.hsize);
                  }
            }
      }

      /**
       * Different from the original algorithm, this phase terminates
       * in only two execution rounds while executing the fast
       * agreement algorithm.
       * A single SYNCHRONIZE message, during the fast agreement will
       * be sent where both the process version number and agreed
       * number will be updated(First Round).  Each process should
       * check the estimate set
       * stored in the SYNCHRONIZE message of the each sender.If
       * all participants have equal estimates, an immediate final
       * view will be sent by the group coordinator to all processes
       * for the installation of the agreed view (Second Round).
       * if the estimate set is not yet stable among processes, or a
       * change in symmetry occur, the slow agreement algorithm will
       * start, EE-Phase
       * @param   msg                the m-received message.
       * @param   src                the sender of the message.
       */
      void handleMsgSynch(MsgSynch msg, Host src)
      {
            HostData    sender;
            // Reference to HostData object of the sender
            HostData    scan; //used to scan the list of hosts

            msg.print();                  // DEBUG
            Debug.println(20,"daemon.Group: handleMsgSynch");
```

```
            if ((sender = (HostData) thosts.lookup(src.getAddress()))
!= null)

                  switch(status) {

                  case S_IDLE:      // IDLE PHASE
                        Debug.println(20, "Group.handleMsgSynch: IDLE
phase calling synchronizationPhaseInit"); // DEBUG

                        //updates the sender's version# with the
latest version number sent by the SYNCH message.
                        me.ctbl=null;
                        if (sender.version < msg.vsend) {
                              sender.version = msg.vsend;

                              //checks that the sender of the message
belongs to the current reachable set
                              if (Util.in(rset, sender.host))
                                    synchronizationPhaseInit();
                        }
                        break;

                  case S_SYNCH:     // SYNCHRONIZATION PHASE
                        Debug.println(20, "Group.handleMsgSynch: S
phase " + sender.version + " " + msg.vsend);    // DEBUG


//If the sender's version number is less than what the destination
process has in its current estimate buffer, it immediately updates
both the sender's version and agreed numbers.

                        if (sender.version < msg.vsend) {
                              Debug.println(20,"sender.version: "+
sender.version + "<" + "msg.vsend: " + msg.vsend);
                              sender.version = msg.vsend;

//If the sender's message contains the current version number of the
destination,  the destination process updates the sender's agreed
number and compares between both estimates

                              if (me.version == msg.vdest){
                                    Debug.println(20, "sender.agreed:
" + sender.agreed + " msg.vsend: " + msg.vsend);
                                    if (!NewAgreement(msg.hosts,
msg.agreed, me)){
                                          Debug.println(20, "Still in
the same round");

                                          if (sender.inEstimate &&
!sender.inSynchronized){

                                          estimate.setSynchronized(se
                                          nder);

//A Special Case where the more rounds will be added and a
SYNCHRONIZE message will have to be sent. If the destination
perceives its current version number updated by the sender, but its
agreed number has not been updated in the sender's estimate. Sender's
version and agreed round number is updated while sending another
SYNCHRONIZE message to avoid using obsolete messages.
```

```
msg.vsend){                                                    if (sender.agreed <

msg.vsend;                                                          sender.agreed =

      Debug.println(20, "Sender.agreed is now updated to:" +
sender.agreed);

                                                               }
                                                               Debug.println(20,
"Group.handleMsgSynch: Sending a R_SYCH message");
                                                               byte[] buf =
MsgSynch.encode(key, me.version, sender.version, estimate.hdata,
estimate.hsize, sender.host.symset);
                                                               mss.msend(R_SYNCH,
buf, src);

//if the estimate viewed by the sender's synchronous message is equal
to the process current estimate, equal estimate will be incremented

                                                               if
(estimate.checkEstimate(msg.hosts)){

      equalEstimate++;

      Debug.println(20, "estimates are equal and the equalEstimate
now is " + equalEstimate);
                                                               }
                                                         }
                                                   }
                                                   else{
                                                         Debug.println(20,
sender.host.getName() + "Has started a new round- removing sender and
starting EE-Phase");
                                                   }
                                             }
                                             else{
                                                   if (sender.inEstimate) {

                                       estimate.setSynchronized(sender);
                                                         if (sender.agreed <
msg.vsend){
                                                               sender.agreed =

msg.vsend;
                                                               Debug.println(20,
"Sender.agreed is now updated to:" + sender.agreed);
                                                         }

                                                         if
(estimate.checkEstimate(msg.hosts)){
                                                               equalEstimate++;
                                                               Debug.println(20,
"estimates are equal and the equalEstimate now is " + equalEstimate);
                                                         }

                                                   }
                                                   else{
                                                         Debug.println(20, "sender
not in Estimate-starting EE-phase");

      estimateExchangePhaseInit();
                                                   }
```

```
                        }
                    }

//if the sender's version number is already updated.  It will be
added to synchronized and its agreed number will be updated as well
                    else{
                        Debug.println(20,"sender.version: "+
sender.version + "=" + "msg.vsend: " + msg.vsend);
                        if (sender.inEstimate &&
!sender.inSynchronized){
                            if (sender.agreed < msg.vsend)
                                sender.agreed = msg.vsend;
                            estimate.setSynchronized(sender);
                            if
(estimate.checkEstimate(msg.hosts)){
                                equalEstimate++;
                                Debug.println(20,
"estimates are equal and the equalEstimate now is " + equalEstimate);
                            }
                        }
                    }

//if all messages are synchronized and all estimates are equal, an
immediate view will directly be installed, otherwise EE-Phase begins.

                    if (estimate.hsize <=
estimate.nsynchronized){
                        Debug.println(20,
"Group.handleMsgSynch: current estimate size: "+ estimate.hsize);
                        Debug.println(20,
"Group.handleMsgSynch: current estimate.nsynchronized: "+
estimate.nsynchronized);
                        if (estimate.hsize <= equalEstimate &&
status==S_SYNCH){
                            Debug.println(20,"Install

immediate view");
                            installImmediateView();
                        }
                        else
                            estimateExchangePhaseInit();
                    }
                    break;

                case S_ESTIMATE:  // ESTIMATE EXCHANGE PHASE
                    Debug.println(20, "Group.handleMsgSynch: EE
phase ");
// DEBUG
                    if (sender.version < msg.vsend)
                        sender.version = msg.vsend;
                    Debug.println(20, "Group.hanldeMsgSynch:
sender.agreed " + sender.agreed + " msg.vsend " + msg.vsend ); //
DEBUG
                    if (sender.agreed < msg.vsend &&
estimate.lookup(src.getAddress()) != null) {
                        estimate.remove(msg.rset);
                        sendEstimate();
                    }

                } // END switch
    }
```

```
    /**
      * If a process p receives a symmetry message with its last
      * version number
      *from a process q which belongs to the current estimate set and
      *runs under the same invocation round, p removes q and all its
      *corresponding reachable set from its current estimate.
       * @param   msg                the m-received message.
       * @param   src                the sender of the message.
       */

      void handleMsgSymm(MsgSymm msg, Host src)
      {
            HostData    sender;
            // Reference to sender host information

            int         vdest;
            // Version number of the destination

            int         pos;
            // My position in the list of messages

            msg.print();


            Debug.println(20,"daemon.Group: handleMsgSymm");
            sender = estimate.lookup(src.getAddress());
            pos = Util.valueAt(msg.hosts, myaddress);
            if (pos >= 0) {
                  vdest = msg.version[pos];

                  if (status == S_SYNCH) {
             // SYNCHRONIZATION PHASE
                        Debug.println(20, "Group.handleMsgSymm: S
PHASE");
                                          // DEBUG
                        if (me.version == vdest) {
                              Debug.println(20, "Group.handleMsgSymm:
accepted");
                                    // DEBUG
                              estimate.remove(msg.rset);
                              if (estimate.hsize <=
estimate.nsynchronized)
                                    estimateExchangePhaseInit();
                        }

                  } else if (status == S_ESTIMATE) {   // ESTIMATE
EXCHANGE PHASE
                        Debug.println(20, "Group.handleMsgSymm: EE
PHASE");           // DEBUG
                        if (sender != null && me.agreed == vdest &&
sender.agreed == msg.vsend) {
                              Debug.println(20, "Group.handleMsgSymm:
accepted");
                                    // DEBUG
                              estimate.remove(msg.rset);
                              sendEstimate();

                        }
                  }
            }
```

```
    }

    /**
     * A process receiving an estimate message, will either be at
     * Synchronous phase, by which it will be informed that an
     * instability happened and the process will immediately start
     * the slow agreement.  Or, a process
     * might receive an estimate message while at EE-Phase.
     * if Status is S-Synch:
     * When a process p receives an Estimate message from a process
     * q that already knows p's current version number and is still
     * part of its current view estimate, p adds q to its
     * synchronized variable to guarantee the termination of the
     * synchronous phase.
     * if Status is S-Estim:
     * When p receives an Estimate message from a process q, while
     * at EE-Phase, it first checks that they both belong to the
     * same agreed phase by
     * checking the agreed number.  If q's estimate is not equal to
     * p's estimate, it intersects both estimates and resends
     * another estimate message announcing the new modification.
     *
     * @param   msg                 the m-received message.
     * @param   src                 the sender of the message.
     */

    void handleMsgEstim(MsgEstim msg, Host src)
    {
          HostData    sender;
          // Reference to sender information

          int         vdest;
          // Version number of destination

          msg.print();


          Debug.println(20,"daemon.Group: handleMsgEstim");
          try {       // DEBUG
          sender = (HostData) thosts.lookup(src.getAddress());
          vdest = msg.agreed[Util.valueAt(msg.hosts, myaddress)];
          Debug.println(20, "vdest " + vdest);

                                  // DEBUG
          Debug.println(20, "me.version " + me.version);

          // DEBUG

          if (status == S_SYNCH) {       // SYNCHRONIZATION PHASE
                Debug.println(20, "Group.handleMsgEstim: S_SYNCH");

    // DEBUG
                if (sender.version < msg.vsend)
                      sender.version = msg.vsend;
                if (!sender.inEstimate) {
                      byte[] buf = MsgSymm.encode(key, me.version,
sender, estimate.hlist, estimate.hsize);
                      mss.msend(R_SYMM, buf, src);
                } else if (me.version == vdest) {

                      estimate.intersect(msg.hosts, msg.agreed);
```

```
                    estimateExchangePhaseInit();
                }

            } else if (status == S_ESTIMATE) {   // ESTIMATE EXCHANGE
PHASE
                Debug.println(20, "Group.handleMsgEstim: EE
PHASE");
// DEBUG
                if (sender.inEstimate && checkAgreed(msg.hosts,
msg.agreed)) {
                    estimate.intersect(msg.hosts, null);
                    sendEstimate();
                }
            }
            } catch (Exception e) { Util.panic("handleMsgEstim", e);
}
                                                        // DEBUG
      }




      /**
       * this message will only be handled if the process has already
       * entered the slow agreement algorithm
       * the coordinator role takes place
       * if the proposal flag is set to true and all proposals have
       * equal estimates, the view will be installed without sending
       * estimate messages
       * However, if the proposal received with a flag set to false,
       * this means that processes have entered the EE-Phase, if all
       * have equal estimate, a new view will be installed.
       *
       * @param   msg               the m-received message.
       * @param   src               the sender of the message.
       */

      void handleMsgProp(MsgProp msg, Host src)
      {
       // Variables containg the new view description (both for
message and local use)
            long        cvid;       // New complete view identifier
            int[]       cvcomp;     // New complete view composition
            int[][]     members;    // Decoded members
            int[]       pvids;      // Local view indexes
            int         pvid; // Index of subview id of this host

            // Variable used to build the view message information
            HostData    sender;     // Reference to host information
                                    //of the sender
            HostData    scan;       // Used to scan tcomp
            MsgProp[]   props;      // Array of proposals
            Host[]      dlist;      // Destination list of view
                                    //message
            int[][]     prev_vcomp; // Previous complete view
                                    //compositions
            long[]      prev_vid;   // Previous view id
            boolean     partial;    // True if partial views are
                                    // needed
            int         npartial;   // Number of different partial
```

```
                                    // views
            long[]      diff_vid;   // Distinct previous view id
            int         len;        // Short for new_vcomp.length
            int         i,j,k;      // Counters


            msg.print();            // DEBUG


            if ( status != S_IDLE && (sender = (HostData)
thosts.lookup(src.getAddress())) != null && sender.inEstimate ) {
                  sender.ctbl = msg;
                  if (sender == me){
                        nproposals--;
                        Debug.println(20, "Number of proposals now= "
+ nproposals);
                  }
                  if (status == S_ESTIMATE && (sender != me ||
nproposals == 0) && checkAgreement()) {
//                      Debug.println(20,"Group.handleMsgProp: group
status is S_ESTIMATE");
                        // Creates a new complete view identifier and
decodes the next view composition
                        cvid    = Types.createsVid(myaddress,
++ncoordinated);
                        cvcomp = me.ctbl.decodeHostProp_hosts();

                        // Allocates arrays
                        len = cvcomp.length;
                        props        = new MsgProp[len];
                        prev_vid     = new long[len];
                        prev_vcomp   = new int [len][];
                        members      = new int [len][];
                        dlist        = new Host[len];

                        // Initializes props and dlist
                        scan = (HostData) thosts;
                        try {


                              // DEBUG
                        Debug.println(20, "current view composition
is:  ");
                        for (i=0; i < len; i++) {
                              scan = (HostData)
scan.lookup(cvcomp[i]);
                              Debug.print(20, " " +
(scan.host.getAddress() & 0xFF));
            // DEBUG
                              props[i] = scan.ctbl;
                              dlist[i] = scan.host;
                        }
                        } catch (Exception e) {
Util.panic("Group.handleMsgProp: Corrupted proposals", e); }
                        // DEBUG
                        Debug.println(20, "");  // DEBUG

                        // Decodes proposals
                        i = 0;
                        try {       // DEBUG
                        for (i=0; i < len; i++) {
```

```
                              prev_vid[i]= props[i].cvid;
                              prev_vcomp[i] =
props[i].decodeLastProp();
                              members[i]= props[i].decodeMembProp();
                        }
                        } catch (Exception e) {

                                                                  // DEBUG
                              Util.panic("Group.handleMsgProp:
Eccezione: len " + len + ", i" + i + ", props[i]" + props[i], e);//
DEBUG
                        }     // DEBUG


                  // Checks whether partial views are needed
                  // ??? Si noti che questo puo' essere ottimizzato
                  // in modo da essere sicuri che che sia O(n^2)
                        partial = false;
                        for (i=0; i < len && !partial; i++)
                              for (j=0; j < prev_vcomp[i].length &&
!partial; j++) {
                                    k = Util.valueAt(cvcomp,
prev_vcomp[i][j]);
                                    partial = ( k >= 0 && prev_vid[i]
!= prev_vid[k] );
//                                  Debug.println(20,
"Group.handleMsgProp: Partial views needed?? " + partial);
                              }

                        // Constructs partial view index
                        pvids = new int[len];
                        pvid = 0;
                        if (partial) {
                              npartial = 0;
                              diff_vid = new long[len];
                              for (i=0; i < len; i++) {
                                    for (j=0; j < npartial &&
prev_vid[i] != diff_vid[j]; j++);
                                    if (j == npartial) {
                                          diff_vid[j] = prev_vid[i];
                                          npartial++;
                                          Debug.println(20,
"Group.jandleMsgProp: number of partial views now are: " + npartial);
                                    }
                                    pvids[i] = j;
                                    if (cvcomp[i] == myaddress) {
                                          pvid = j;
                                    }
                              }
                              ncoordinated += npartial;
                              Debug.println(20, "number of views
coordinated so far is " + ncoordinated);
                        }

                        // Reliable m-send the view message
containing the information
                        mss.rmsend(R_VIEW, MsgView.encode(key, cvid,
me.ctbl, pvids, props), dlist);

                        // Locally installs the view
```

```
                        int[] incarns =
me.ctbl.decodeHostProp_incarns();
                        installView(cvid, cvcomp, incarns, members,
pvids, pvid);
                  }
            }
      }


      /**
       * Locates the position of this host in the complete view and
       * checks whether theversion number in the message is equal to
       * the cuurent one then it installs the agreed upon view
       *
       * @param   msg                the m-received message.
       * @param   src                the sender of the message.
       */

      void handleMsgView(MsgView msg, Host src)
      {
            int         pos;        // Counter
            HostData    sender;

            msg.print();      // DEBUG
            sender = (HostData) thosts;
            Debug.println(20,"handleMsgView");

            if (status == S_SYNCH &&
estimate.lookup(src.getAddress()) != null){
                  for (int i =0; i<msg.cvcomp.length; i++){
                        sender = (HostData)
sender.lookup(msg.cvcomp[i]);
                        if (sender.agreed != msg.agreed[i] &&
sender.host != me.host){
                              sender.version = msg.agreed[i];
                              sender.agreed = msg.agreed[i];
                        }
                  }
                  pos = Util.valueAt(msg.cvcomp, myaddress);
                  if (pos >= 0 && msg.agreed[pos] == ninstalled+1)
                        installView(msg.cvid, msg.cvcomp,
msg.incarns, msg.members, msg.pvids, msg.pvids[pos]);

            }
            if (status == S_ESTIMATE &&
estimate.lookup(src.getAddress()) != null) {
                  pos = Util.valueAt(msg.cvcomp, myaddress);
                  if (pos >= 0 && msg.agreed[pos] == ninstalled+1)
                        installView(msg.cvid, msg.cvcomp,
msg.incarns, msg.members, msg.pvids, msg.pvids[pos]);
            }
      }


////////////////////////////////////////////////////////////////////
// Methods related to the total composition of the group
////////////////////////////////////////////////////////////////////

      /*
       * Removes MemberData item which have requested to exit and
       * have been excluded from the current view.
```

```
     */

    private void removeLeavingMembers(int gid, int last)
    {
          MemberData  scan;       // Used to scan the list of
members
          MemberData  prev;       // Scan = prev.next

          Debug.println(20,"daemon.Group:  removeLeavingMembers-
calling StandardGM.notifyLeaveAck for member with Gid=  "+ gid);

          prev = (MemberData) tmembers;
          scan = (MemberData) prev.getFirst();
          while (scan != null) {
                if (scan.leaving && scan.last != last) {
                      scan.member.notifyLeaveAck(gid);
                      prev.remove(scan.key);
                } else
                      prev = scan;
                scan = (MemberData) scan.getNext();
          }
    }


    /*
     *
     */
    private void removeLeavingHosts(int key, int last)
    {
          HostData    scan; // Used to scan the list of members
          HostData    prev; // scan = prev.next

          Debug.println(20,"daemon.Group: removeLeavingHosts
sending R_LEAVE MsgGroup");
          prev = (HostData) thosts;
          scan = (HostData) prev.getFirst();
          while (scan != null) {
                if (scan.leaving && scan.last != last) {
                      mss.msend(Tag.R_LEAVE, MsgGroup.encode(key),
scan.host);
                      prev.remove(scan.key);
                } else
                      prev = scan;
                scan = (HostData) scan.getNext();
          }
    }


////////////////////////////////////////////////////////////////////////
// Methods related to the current view estimate
////////////////////////////////////////////////////////////////////////

    /*
     * Each process msends a synchronization message containing a
     * version number to those
     * processes it perceives as being reachable, then waits for
     *responses.
     * the S_PHASE lasts until all processes in the view estimate
     * have replied, nstalled a final agreed view and terminates
     * during fast agreement
     * or when p m-receives a message from a process in its view
```

```
     * estimate that has already entered EE_PHASE
     */

    private void synchronizationPhaseInit()
    {
          HostData    hd;         // Optimization variable

          Debug.println(20, "synchronizationPhaseInit");

          // DEBUG
          //sets the start time, and initializes equal estimate,
the status
           the current estimate, version# and agreed number
          start = System.currentTimeMillis();
          System.out.println("Start: "+ start);
          equalEstimate = 0;
          status = S_SYNCH;

                // DEBUG
          Debug.print(20, "LastEstimate ");

          estimate.init(thosts, rset);
          me.inSynchronized = true;
          me.version++;
          me.agreed++;
          equalEstimate++;
          Debug.println(20, "current equalEstimate is now: " +
equalEstimate);
          Debug.println(20,"Group.S-Phase:My hostData info is:
hostname= " + me.host.getName() + " version= " + me.version + "
agreed= " + me.agreed);
          Debug.println(20, "estimate.coord: " +
estimate.coord.getName() + "me.host: " + me.host.getName());

          for (int i=0; i < estimate.hsize; i++)
                if ((hd = estimate.hdata[i]) != me) {
                      Debug.println(20, me.host.getName() + "
sending a SYNCHRONIZE message" );
                      byte[] buf = MsgSynch.encode(key, me.version,
hd.version, estimate.hdata, estimate.hsize, hd.host.symset);
                      mss.msend(R_SYNCH, buf, hd.host);
                }

//if all estimates are equal, the fast agreement will handle this
part and an installation of an immediate view will be issued.
                if (estimate.hsize <= estimate.nsynchronized){
                Debug.println(20, "current estimate size: "+
estimate.hsize);
                Debug.println(20, "current estimate.nsynchronized:
"+ estimate.nsynchronized);
                if (estimate.hsize <= equalEstimate){
                      Debug.println(20,"going to install immediate
view");
                      installImmediateView();
                }
                else{
                      estimateExchangePhaseInit();
                }

          }
    }
```

```
      /**
       *The Slow Agreement Algorithm
       * This phase handles the modification and installation of the
       * view estimate.  View Estimates are modified when the sender
       * receives an estimate from a reachable process but different
       * from his own.
       * Consequently, it intersects its estimate with the one
       * received from the sender.
       * This continues until the group coordinator receives equal
       * estimates from all participants; then the installation of
       * the view takes place.  Processes do not exit the agreement
       * phase unless a final agreed view is installed.
       */


      private void estimateExchangePhaseInit()
      {
            Queue       buffer;           // Temporary variable
            MsgMcast    msg;        // MsgDlvrd message

            Debug.println(20, "estimateExchangePhaseInit");

            // DEBUG
            status = S_ESTIMATE;
            mss.msend(R_ESTIM, MsgEstim.encode(key, me.version,
estimate.hdata, estimate.hsize), estimate.hlist, estimate.hsize);
            proposal.encodeLastProp(cview, cview.length);
            proposal.encodeHostProp(estimate.hdata, estimate.hsize);
            proposal.encodeMembProp(tmembers, nmembers);
            if (msgsStable) {
                  proposal.encodeMsgsProp(estimate.hdata,
estimate.hsize);
                  sendProposal();
            }
            for (int i=0; i< pview.hsize; i++) {
                  buffer = pview.hdata[i].buffer;
                  while ( (msg = (MsgMcast) buffer.removeFirst()) !=
null )
                        mss.msend(R_FORWARD, msg.data, pview.hlist,
pview.hsize);
            }
      }


      /*
       * A method responsible for sending the proposal message to the
       * assigned coordinator
       */

      private void sendProposal()
      {
            if (msgsStable && viewStable) {
                  mss.msend(R_PROP, proposal.encode(key, cvid, pvid),
estimate.coord);
                  if (estimate.coord == me.host){
                        nproposals++;
                        Debug.println(20, "current number of
proposals are: "+ nproposals);
                  }
```

```
        }
    }


    /*
     *This procedure is used to modify the view estimate and to
     *inform the other processes of the change.  Each process sends
     *a new estimate message to all processes in its current view
     *and a proposal message to the group coordinator.
     */

    private void sendEstimate()
    {
        Debug.print(20, "sending Estimate: ");

                                                    // DEBUG
        for (int i=0; i<estimate.hsize; i++)

                                    // DEBUG
            Debug.print(20,
(estimate.hdata[i].host.getAddress() & 0xFF) + " ");              //
DEBUG
        Debug.println(20, "");

                                                          // DEBUG
        if (estimate.echanged) {
            byte[] buf = MsgEstim.encode(key, me.version,
estimate.hdata, estimate.hsize);
            mss.msend(R_ESTIM, buf, estimate.hlist,
estimate.hsize);
            proposal.encodeHostProp(estimate.hdata,
estimate.hsize);
            sendProposal();
            noAgreement++;
            Debug.println(20, "NoAgreement= " + noAgreement);

        }
    }

    /*
     * This function is used by the Group coordinator when a
     * process m-receives a propose message.  It verifies whether
     * the proposals stored in the coordinator table buffer are in
     * agreement, i.e., all estimates are equal.
     */

    private boolean checkAgreement()
    {
        MsgProp     pp;             // Proposal of this host
        MsgProp     pq;             // Proposal of compared host
        int         i,j;            // Counters



        Debug.println(20,"checkAgreement");
        if ((pp = me.ctbl) == null){
            return false;
        }
        for (i=0; i < estimate.hsize; i++) {
            pq = estimate.hdata[i].ctbl;
            if (!pp.checkEstimate(pq))
```

```
                        return false;
                  for (j=i+1; j < estimate.hsize; j++)
                        if
(!pq.checkMessages(estimate.hdata[j].ctbl))
                              return false;
            }
            return true;
      }


      /*
       *
       */
      private void checkAck(HostData hd)
      {
            int         min;  // New minimal value in hack array
            int         i;    // Counter

            Debug.println(20,"daemon.Group: checkAck CHECK IT!!!");
            min = Integer.MAX_VALUE;
            for (i=0; i < hd.hack.length; i++) {
                  // System.out.print(" " + hd.hack[i]);
                  if (min > hd.hack[i])
                        min = hd.hack[i];
            }
            if (min > hd.hmin) {
                  for (i=hd.hmin; i < min; i++) {
                        // System.out.print("@");
                        hd.buffer.removeFirst();
                  }
                  hd.hmin = min;
            }
      }

      /*
       * compares the agreed list of the current estimate of both my
       *current estimate and the sender's estimate
       */

      boolean checkAgreed(int[] hosts, int[] agreed)
      {
            int   i=0, j=0;         // Counters

            Debug.println(20,"CheckAgreed");
            while (i < hosts.length && j < estimate.hsize)
                  if (hosts[i] == estimate.hdata[j].key) {
                        if (agreed[i] != estimate.hdata[j].agreed)
                              return false;
                        i++; j++;
                  }
                  else if (hosts[i] < estimate.hdata[j].key) {
                        i++;
                  }
                  else {
                        j++;
                  }

            return true;
      }

      /**
```

```
        *A method used to check the agreed list of the sender's
        *synchronize message whether my agreed round number is
        * already updated in the sender
        * or the process have to re-send a synchronize message
        * (Special Case)
        */

      boolean NewAgreement(int[] hosts, int[] agreed, HostData src)
      {
            int   i=0;          // Counters

            Debug.println(20,"Check NewAgreement");
            for (i=0; i<hosts.length; i++)
                  if (hosts[i] == src.key) {
                        if (agreed[i] < src.version){
                              Debug.println(20, "me.agreed is not
updated");
                              return false;
                        }

                  }
            return true;

      }

      /**
       *
       */
      private void localDeliver(MsgMcast msg)
      {
            MemberData  md;          // Used to scan <m_vcomp>
            HostData    hd;          // Additional info of the
                                     //original sender
            byte[]      data;        // Temporary buffer

            Debug.println(20, "daemon.Group: localDeliver");

            data = new byte[msg.len];
            System.arraycopy(msg.data, msg.start, data, 0, msg.len);
            for (int i=0; i<pview.msize; i++) {
                  md = pview.mdata[i];
                  if (!md.leaving){
                        md.member.notifyDlvr(key, msg.vid, msg.hpos,
msg.mid, msg.flag, data);
                  }
            }
            hd = pview.hdata[msg.hpos];
            hd.hack[hpos] = msg.mid;
            hd.hack[msg.hpos] = msg.mid;
            hd.buffer.insert(msg);
            msgsStable = false;
      }


      /*
       *
       */
      private void acceptMessage(MsgMcast msg)
      {
```

```
            Debug.println(20, "daemon.Group: acceptMessage");

                                    // DEBUG
            nmessages++;
            msg.complete(pvid, hpos, nmessages);
            localDeliver(msg);
            checkAck(me);
            // TODO: Mettere il checkack
            mss.msend(R_MCAST, msg.data, pview.hlist, pview.hsize);
      }

      private void acceptLateMessage(MsgMcast msg)
      {
            Debug.println(20, "Daemon: acceptLateMessage");

                                          // DEBUG
            nmessages++;
            msg.complete(pvid, hpos, nmessages);
            localDeliver(msg);
            // TODO: Mettere il checkack
            // TODO: Modificare per spedire solo agli elementi
rimasti in lista
            mss.msend(R_FORWARD, msg.data, pview.hlist, pview.hsize);
      }
      /**
       * Going to install immedaite view if view is stable at first
       * round while at Fast Agreement.  The coordinator proposed
       * estimate will be encoded, creating the new view composition
       * and id the coordinator then installs this view immediately
       * and sends a view message to all processes participating in
       * this view
       */

      void installImmediateView()
      {
       // Variables containg the new view description (both for
       // message and local use)
            long        cvid;       // New complete view identifier
            int[]       cvcomp;     // New complete view composition
            int[]       cvc;
            int[][]     members;    // Decoded members
            int[]       pvids;      // Local view indexes
            int         pvid;       // Index of subview id of this
                                    // host

            // Variable used to build the view message information
            HostData    sender;     // Reference to host information
                                    // of the sender
            HostData    scan;       // Used to scan tcomp
            MsgProp[]   props;      // Array of proposals
            Host[]      dlist;      // Destination list of view
                                    // message
            int[][]     prev_vcomp; // Previous complete view
                                    //compositions
            long[]      prev_vid;   // Previous view id
            boolean     partial;    // True if partial views are
                                    // needed
            int         npartial;   // Number of different partial
                                    // views
            long[]      diff_vid;   // Distinct previous view id
            int         len;        // Short for new_vcomp.length
```

```
            int         i,j,k;      // Counters
            int[]       trial;
            byte[]      msg;
            MsgProp     prop;
            HostData    hd;

            // Creates a new complete view identifier for the
            // coordinator and decodes the next view composition


            if (estimate.coord == me.host){
                  msg = encodeImmediateProposal();
                  prop = new MsgProp(msg);
                  prop.print();
                  for (i=0; i<estimate.hsize;i++){
                        sender = estimate.hdata[i];
                        sender.ctbl = prop;
                  }

                        cvid   = Types.createsVid(myaddress,
++ncoordinated);
                        cvcomp = me.ctbl.decodeHostProp_hosts();

                        // Allocates arrays
                        len = cvcomp.length;
                        props        = new MsgProp[len];
                        prev_vid     = new long[len];
                        prev_vcomp   = new int [len][];
                        members            = new int [len][];
                        dlist        = new Host[len];

                        // Initializes props and dlist
                        scan = (HostData) thosts;
                        try {

                        Debug.println(20,
"Group.installImmediateView: current view composition is:  ");
                        for (i=0; i < len; i++) {
                              scan = (HostData)
scan.lookup(cvcomp[i]);
                              Debug.print(20, " " +
(scan.host.getAddress() & 0xFF));
            // DEBUG
                              props[i] = scan.ctbl;
                              dlist[i] = scan.host;
                        }
                        } catch (Exception e) {
Util.panic("Group.installImmediateView: Corrupted proposals", e); }
                              // DEBUG
                        Debug.println(20, "");

                        // Decodes proposals
                        i = 0;
                        try {


                              // DEBUG
                        for (i=0; i < len; i++) {
                              prev_vid[i]= props[i].cvid;
                              prev_vcomp[i] =
props[i].decodeLastProp();
```

```
                        members[i]= props[i].decodeMembProp();
                    }
                    } catch (Exception e) {

                                                        // DEBUG
                        Util.panic("Group.handleMsgProp:
Eccezione: len " + len + ", i" + i + ", props[i]" + props[i], e);//
DEBUG
                    }

                    // Checks whether partial views are needed
                    // ??? Si noti che questo puo' essere
ottimizzato
                    // in modo da essere sicuri che che sia
O(n^2)
                    partial = false;
                    for (i=0; i < len && !partial; i++)
                        for (j=0; j < prev_vcomp[i].length &&
!partial; j++) {
                            k = Util.valueAt(cvcomp,
prev_vcomp[i][j]);
                            partial = ( k >= 0 && prev_vid[i]
!= prev_vid[k] );
                        }

                    // Constructs partial view index
                    pvids = new int[len];
                    pvid = 0;
                    if (partial) {
                        npartial = 0;
                        diff_vid = new long[len];
                        for (i=0; i < len; i++) {
                            for (j=0; j < npartial &&
prev_vid[i] != diff_vid[j]; j++);
                            if (j == npartial) {
                                diff_vid[j] = prev_vid[i];
                                npartial++;
                            }
                            pvids[i] = j;
                            if (cvcomp[i] == myaddress) {
                                pvid = j;
                            }
                        }
                        ncoordinated += npartial;
                        Debug.println(20, "number of views
//
coordinated so far is " + ncoordinated);
                    }

                    // Reliable m-send the view message
containing the information
                    mss.rmsend(R_VIEW, MsgView.encode(key, cvid,
me.ctbl, pvids, props), dlist);

                    // Locally installs the view
                    int[] incarns =
me.ctbl.decodeHostProp_incarns();
                    installView(cvid, cvcomp, incarns, members,
pvids, pvid);


            }
```

```
    }

    /**
     * This procedure handles the installation of the new view.It
     * creates the new view id and composes the new view based on
     * the current estimate
     */

    private void installView(long cvid, int[] chosts, int[]
incarns, int[][] cmembers, int[] pids, int pid)
    {
          int         plen;       // Partial view length
          InetAddress[]     pv_hosts;
          int[]       pv_vindex;
          ViewImpl    view;       // View to be passed to hosts
          HostData    hd;         // Used to scan thosts
          MemberData  md;         // Used to scan m_vcomp
          MsgMcast    scan;       // Used to scan delayed messages
          boolean     stable;           // True if the current view
is stable

          // Stores complete view information
          this.cvid = cvid;
          cview = (int[]) cview.clone();

// Computes host position in the current view and removes info
related to
// hosts and members not participating in the next (partial) view.
          plen = 0;
          for (int i=0; i < chosts.length; i++)
                if (pids[i] == pid) {
                      chosts[plen] = chosts[i];
                      incarns[plen] = incarns[i];
                      cmembers[plen] = cmembers[i];
                      if (chosts[plen] == myaddress)
                            hpos = plen;
                      plen++;
                }

          // Stores current view information
          ninstalled++;
          stable = pview.init(thosts, chosts, plen, ninstalled,
rset, tmembers, cmembers[hpos]);

          // Updates group flags
          msgsStable = true;
          viewStable = false;
          status = S_IDLE;

          // Build a new view
          pvid          = Types.computePartialVid(cvid, pid);
          pv_hosts      = new InetAddress[plen];
          pv_vindex     = new int[plen];
          for (int i=0; i < plen; i++) {
                hd = (HostData) thosts.lookup(chosts[i]);
                pv_hosts[i] = hd.host.getInetAddress();
                pv_vindex[i]      = hd.host.getIndex();
          }
          view = new ViewImpl(key, pvid, pv_hosts, pv_vindex,
incarns, cmembers, hpos);
```

```
            Debug.println(20, "Stable view: " + stable);

                  // DEBUG
            Debug.print(20, "Vid: " + pvid);

                                          // DEBUG
            Debug.println(20, " nviewInst: " + ninstalled);

            // DEBUG

            // Deliver the view to non-leaving members
            for (int i=0; i < pview.msize; i++)
                  if ((md = pview.mdata[i]) != null && !md.leaving)
                        md.member.notifyView(key, view, ninstalled,
hpos);

            // Remove hosts and members that have request to leave
            removeLeavingHosts(key, ninstalled);
            removeLeavingMembers(key, ninstalled);

            if (!tmembers.isEmpty()) {          // The group has not
to be removed

                  // Updates HostData information
                  hd = (HostData) thosts.getNext();
                  while (hd != null) {
//                      Debug.println(20, hd.host.getName() + "has
been re-initialized");
                        hd.ctbl = null;
                        hd.hack = new int[pview.hsize];
                        hd.mack = new int[pview.msize];
                        hd.delivered = 0;
                        hd.hmin = 0;
                        hd.buffer.clear();
                        hd = (HostData) hd.getNext();
                  }

               // Updates information about messages
                     nmessages = 0;
                     nproposals = 0;
                     while ((scan = (MsgMcast)
delayedSent.removeFirst()) != null)
                        acceptMessage(scan);
                     while ((scan = (MsgMcast)
delayedRcvd.removeFirst()) != null)
                        if (scan.vid == pvid)
                              localDeliver(scan);
                     if (nmessages > 0) {
                        byte[] buf = MsgAck.encode(key, pvid, hpos,
pview.hdata, pview.hsize);
                        mss.msend(R_ACK, buf, pview.hlist,
pview.hsize);
                     }

                     // Checks if the view is stable
                     if (!stable){
                        Debug.println(20, "View is not stable");
                        end = System.currentTimeMillis();
                        System.out.println("SA End Time: " + (end-
start));
```

```
                        Debug.println(20, "SA EndTime:" + (end-
start));
                        synchronizationPhaseInit();
                  }
                  else{
                        end = System.currentTimeMillis();
                        System.out.println("FA End Time: " + (end-
start));
                        Debug.println(20, "FA EndTime:" + (end-
start));
                  }

            }
      }
/**
  * this method is responsible to decode the proposed view of the
coordinator
  * that will be used to install the immediate view
  */

private byte[] encodeImmediateProposal()
{
      byte[]  data;
      data  = new byte[100];
      int size;

      ImmMsg.encodeLastProp(cview, cview.length);
      ImmMsg.encodeHostProp(estimate.hdata, estimate.hsize);
      ImmMsg.encodeMembProp(tmembers, nmembers);
      if (msgsStable && viewStable) {
            Debug.println(20, "message is stable and view is
stable");
            ImmMsg.encodeMsgsProp(estimate.hdata, estimate.hsize);
            data =  ImmMsg.encode(key, cvid, pvid);
      }
      return data;
}

} // END Group
```

Estimate.java in Both Algorithms

```
/*
 * Copyright (c) 1998,1999 The Jgroup Team. All rights reserved.
 *
 */

package jgroup.daemon;

import jgroup.mss.MssHost;
import jgroup.mss.Mss;
import jgroup.general.*;
import debug.Debug;


/**
 * The <code> HostData</code> class
 *
 * @author  Alberto Montresor
 * @since   Jgroup 0.3
 */

/**
 * The <code> HostData</code> class
 * modified Estiamte used for both algorithms I & II
 * @author  Marwa Mansour
 */

final class Estimate
{

////////////////////////////////////////////////////////////////////////
// Fields
////////////////////////////////////////////////////////////////////////

      Host[]      hlist;
      // Array of hosts contained in the estimate

      HostData[]  hdata;
      // Additional data associated to hosts in <hlist>

      int   hsize;
      // N. of hosts in hlist/hdata (hsize <= hlist.length)

      int   nsynchronized;
      // Number of synchronized elements in estimate

      boolean     echanged;   // True if estimate has changed
      Host        coord;      // Coordinator


////////////////////////////////////////////////////////////////////////
// Constructors
////////////////////////////////////////////////////////////////////////

      Estimate()
      {
            hdata = new HostData[0];
            hlist = new Host[0];
      }
```

```
////////////////////////////////////////////////////////////////////////
// Methods
////////////////////////////////////////////////////////////////////////

      /*
       * Creates a new estimate by intersecting the total list of
       * hosts contained in
       * <thosts> and the list of reachable host contained in <rset>.
       *
       * @param thosts                the total list of hosts
       * @param rset                        the reachable set
       */
      void init(HostData thosts, MssHost[] rset)
      {
            HostData    scan;       // Used to scan total
            int         i=0;        // Used to scan rset

            if (hdata.length < rset.length) {
                  hdata = new HostData[rset.length];
                  hlist = new Host[rset.length];
            }

            Debug.print(20, "Estimate: ");            // DEBUG
            scan = (HostData) thosts.getFirst();
            hsize = 0;
            while (scan != null) {
                  if (i == rset.length || scan.key <
rset[i].getAddress() || scan.leaving) {
                        scan.inEstimate = false;
                        scan = (HostData) scan.getNext();
                  }
                  else if (scan.key == rset[i].getAddress() &&
!scan.leaving) {
                        Debug.print(20, " " + scan.host.getName());
      // DEBUG
                        hdata[hsize] = scan;
                        hlist[hsize] = scan.host;
                        scan.inEstimate = true;
                        scan.inSynchronized = false;
                        scan = (HostData) scan.getNext();
                        hsize++;
                        i++;
                  }
                  else {
                        i++;
                  }
            }
            Debug.println(20, " " +hsize);

                        // DEBUG


            coord = hlist[0];
            echanged = true;
            nsynchronized = 1;
            Debug.println(20,"The current Group coordinator is: " +
coord.getName());
      }
```

```
/*
 * Computes the intersection between the current estimate and
 * the set <hosts>
 *
 * @param hosts   the hosts to be intersected with the current
 * estimate
 * @param agreed  the agreed version numbers received with the
 * hosts
 */

void intersect(int[] hosts, int[] agreed)
{
      int                                 nsize=0;
                              // New hsize of the estimate
      int                                 i=0, j=0;
                              // Counters

      while (i < hsize && j < hosts.length) {
            if (hdata[i].key == hosts[j]) {
                  // Do not remove hdata[i]/hlist[i]
                  hdata[nsize] = hdata[i];
                  hlist[nsize] = hlist[i];
                  if (agreed != null)
                        hdata[nsize].agreed = agreed[j];
                  nsize++;
                  i++; j++;
            } else if (hdata[i].key < hosts[j]) {
                  // Remove hdata[i]/hlist[i]
                  hdata[i].inEstimate = false;
                  if (hdata[i].inSynchronized)
      // If synchronized, it has to be removed from the count
                        nsynchronized--;
                  i++;
            } else {
                  j++;
            }
      }

      echanged = (nsize < hsize);
      hsize              = nsize;
      coord        = hlist[0];
}


/*
 * Remove the host described by <host> from the current
 * estimate.
 *
 * @param host          the host to be removed
 */
void remove(MssHost host)
{
      int   i=0, j=0;          // Counters

      while (i < hsize) {
            if (hlist[i] == host) {
                  // Remove hdata[i]/hlist[i]
                  Debug.println(20, "Estimate.remove: removing
host " +  host.getName() + " from current view estimate");
                  echanged = true;
```

```
                        hsize--;
                        hdata[i].inEstimate = false;
                        if (hdata[i].inSynchronized)
                              // If synchronized, it has to be
removed from the count
                              nsynchronized--;
                  } else {
                        // Do not remove hdata[i]/hlist[i]
                        hdata[j] = hdata[i];
                        hlist[j] = hlist[i];
                        j++;
                  }
                  i++;
            }
            coord = hlist[0];
            Debug.println(20, "Estimate.remove: The group coordinator
now is: " +  coord.getName());
      }


      /*
       * Remove from the estimate the hosts contained in the <uset>
array.
       */
      void remove(MssHost[] uset)
      {
            int                              nsize=0;
                              // New hsize of the estimate
            int                              i=0, j=0;
                              // Counters

            while (i < hsize) {
                  if (j == uset.length || hdata[i].key <
uset[j].getAddress()) {
                        // Do not remove hdata[i]/hlist[i]
                        hdata[nsize] = hdata[i];
                        hlist[nsize] = hlist[i];
                        nsize++;
                        i++;
                  } else if (hdata[i].key == uset[j].getAddress()) {
                        // Remove hdata[i]/hlist[i]
                        hdata[i].inEstimate = false;
                        if (hdata[i].inSynchronized)
                              // If synchronized, it has to be
removed from the count
                              nsynchronized--;
                        i++; j++;
                  } else
                        j++;
            }

            echanged = (nsize < hsize);
            hsize              = nsize;
            coord        = hlist[0];
      }
```

```
      /*
       * Remove from the estimate the hosts whose addresses are
       * contained in the <uset>
       * array.
       */
      void remove(int[] uset)
      {
            int                                 nsize=0;
                              // New hsize of the estimate
            int                                 i=0, j=0;
                              // Counters

            nsize = 0;
            while (i < hsize) {
                  if (j == uset.length || hdata[i].key < uset[j]) {
                                                // heVcomp[i].key is not
removed
                        // Do not remove hdata[i]/hlist[i]
                        hdata[nsize] = hdata[i];
                        hlist[nsize] = hlist[i];
                        nsize++;
                        i++;
                  } else if (hdata[i].key == uset[j]) {
                                                      // heVcomp[i].key
must be removed
                        // Remove hdata[i]/hlist[i]
                        hdata[i].inEstimate = false;
                        if (hdata[i].inSynchronized)
                              // If synchronized, it has to be
removed from the count
                              nsynchronized--;
                        i++; j++;
                  } else {
                        j++;
                  }
            }

            echanged = (nsize < hsize);
            hsize        = nsize;
            coord        = hlist[0];
      }


      /*
       * Return true if the host identified by <address> is contained
       * in the estimate.
       */

      HostData lookup(int address)
      {
            int   i;              // Counter

            for (i=0; i < hsize && hdata[i].key < address; i++);
            return ((i<hsize && hdata[i].key == address) ? hdata[i] :
null);
      }

      /**
       * If hosts contained in <hi> belongs to the estimate and is
       * not synchronized, change its <inSynchronized> field and
       * update the counter of synchronized hosts.
```

```
      */

      void setSynchronized(HostData hd)
      {
      .     if (hd.inEstimate && !hd.inSynchronized) {
                  Debug.println(20, hd.host.getName()+" is added to
synchronized");
                  hd.inSynchronized = true;
                  nsynchronized++;
            }
      }

      /**
       * compares between two estimate sets during the S-Phase
       *returns true if the estimate sent is equal to my current
       * estimate
       */
      boolean checkEstimate(int[] hosts)
      {
            if (hsize != hosts.length)
                  return false;
            for(int i=0; i<hsize; i++)
                  if (hdata[i].key != hosts[i])
                        return false;
            return true;
      }


} // END Estimate
```

MsgSynch.java in both Algorithms

```
/*
 * Copyright (c) 1998,1999 The Jgroup Team. All rights reserved.
 *
 */

package jgroup.daemon;

import jgroup.mss.MssHost;
import jgroup.general.Types;
import debug.Debug;


/**
 * The <code> MsgSynch</code> class
 *
 * @author  Alberto Montresor
 * @since          Jgroup 0.1
 */

/**
 * The <code> MsgSynch</code> class
 * Modified MsgSynch class
 * @author  Marwa Mansour, 2000
 *
*/

final class MsgSynch
{

////////////////////////////////////////////////////////////////////////
// Position constants
////////////////////////////////////////////////////////////////////////

//The estimate set and agreed list where added to the MsgSynch code.
This required to the size of the data buffer dynamically since, we
cannot predict the size of the estimate set received

      private static final int P_GID      = 0;         // Gid field
      private static final int P_VSND = P_GID    + Types.GID_SIZE;
            // Sender version number
      private static final int P_VDST = P_VSND + Types.VER_SIZE;
      // Dest. version number
      private static final int P_VLEN = P_VDST + Types.VER_SIZE;
      // Size of the view estimate
      private static final int P_VSET = P_VLEN + Types.GRP_SIZE;
      // Estimate set/version n.


////////////////////////////////////////////////////////////////////////
// Unique message instance
////////////////////////////////////////////////////////////////////////

      private static MsgSynch msg;

      static {
```

```
            msg = new MsgSynch();
      }

////////////////////////////////////////////////////////////////////////
// Fields
////////////////////////////////////////////////////////////////////////

      int         vsend;      // Sender version number
      int         vdest;      // Destination version number
      int[]       rset;       // Reachable set (32 bit adressess)
      int[]       agreed;     // Agreed version number table
      int[]       hosts;      // View estimate set



////////////////////////////////////////////////////////////////////////
// Decoding methods
////////////////////////////////////////////////////////////////////////

      /**
       * Decodes a <code> MsgSynch</code> message starting from a m-
       * received sequence of
       * bytes.
       */

      static MsgSynch decode(byte[] data)
      {
            int   len;        // Length of the rset field
            int   vlen;       // Version length

            int   pos;        // Position; used to scan array data

            // Read data
            msg.vsend = Types.byte2ver(data, P_VSND);
            msg.vdest = Types.byte2ver(data, P_VDST);
            len = Types.byte2grp(data, P_VLEN);
            msg.agreed   = new int[len];
            msg.hosts    = new int[len];
            pos = P_VSET;

            //dynamic allocation of the agreed list
            for (int i=0; i<len; i++) {
                  msg.agreed[i] = Types.byte2ver(data, pos);
                  pos += Types.VER_SIZE;
            }

            //dynamic allocation of the estimate set
            for (int i=0; i<len; i++) {
                  msg.hosts[i]      = Types.byte2hid(data, pos);
                  pos += Types.HID_SIZE;
            }

            //Read rset array
            len = Types.byte2grp(data, pos);
            msg.rset    = new int[len];
            pos += Types.GRP_SIZE;
            for (int i=0; i < len; i++) {
                  msg.rset[i] = Types.byte2hid(data, pos);
                  pos += Types.HID_SIZE;
            }
```

```
        return msg;

    }


////////////////////////////////////////////////////////////////////////
// Encoding methods
////////////////////////////////////////////////////////////////////////

    /**
     * Computes the encoding of a <code> MsgSynch</code> message.
     *
     * @param   gid         group identifier.
     * @param   vsend       version number of the sender host
     * @param   vdest       version number of the destination host
     * @param   rset        reachable set
     * @param   heVcomp           estimate set
     * @param   elen        estimate length
     */
    static byte[] encode(int gid, int vsend, int vdest, HostData[]
heVcomp, int elen, MssHost[] rset)
    {
        int         size;       // Contents length
        byte[]      data;       // Mss message for sending
        int         pos;        // Position; used to scan array
                                // data
        int         count;

        // Allocates data buffer

        size = Types.GID_SIZE + 2*Types.VER_SIZE + Types.GRP_SIZE
+ (Types.VER_SIZE + Types.HID_SIZE) * elen + Types.GRP_SIZE +
rset.length*Types.HID_SIZE;

        Debug.println(20, "initial size" + size);
        data = new byte[size];

        // Writes data
        Types.gid2byte(data, P_GID , gid );
        Types.ver2byte(data, P_VSND, vsend);
        Types.ver2byte(data, P_VDST, vdest);
        Types.grp2byte(data, P_VLEN, elen);

        pos = P_VSET;
        for (int i=0; i < elen; i++)
            pos = Types.ver2byte(data, pos, heVcomp[i].agreed);

        for (int i=0; i < elen; i++)
            pos = Types.hid2byte(data, pos, heVcomp[i].key);

        //Writes rset Array
        Types.grp2byte (data, pos, rset.length);
        pos += Types.GRP_SIZE;
        for (int i=0; i < rset.length; i++)
            pos = Types.hid2byte(data, pos,
rset[i].getAddress());

        return data;

    }
```

```
// DBEGIN
      void print()
      {
            int   i;           //counter
            Debug.print(20, "Vsend: "+ vsend);
            Debug.print(20, "| Vdest: "+ vdest);
            Debug.print(20, "|Estimate: ");
            for (i=0; i < hosts.length; i++)
                  Debug.print(20, (hosts[i] & 0xFF) + " ");
            Debug.print(20, "");
            Debug.print(20, "| Agreed: ");
            for (i=0; i < agreed.length; i++)
                  Debug.print(20, agreed[i] + " ");
            Debug.print(20, "");
            Debug.print(20, "| Rset:");
            for (i=0; i < rset.length; i++)
            Debug.print(20, " " + (rset[i] & 0xFF));
            Debug.println(20, "");

      }
// DEND

} // END MsgSynch
```

MsgProp.java in both Algorithms

```
/*
 * Copyright (c) 1998,1999 The Jgroup Team. All rights reserved.
 *
 */

package jgroup.daemon;

import jgroup.general.*;
import debug.Debug;



/**
 * The <code> MsgProp</code> class
 *
 * @author  Alberto Montresor
 * @since          Jgroup 0.1
 */

/**
 * The <code> MsgProp</code> class
 * Modified jgroup Algorithm I
 * @author  Marwa Mansour
*/

final class MsgProp
{

////////////////////////////////////////////////////////////////////////
// Position constants
////////////////////////////////////////////////////////////////////////

      private static final int P_GID      = 0;        // gid field
      private static final int P_FLAG =  P_GID + Types.GID_SIZE;

      //a flag to distinguish between proposals sent while at S_Synch
and EE phases
      private static final int P_CVID = P_FLAG + Types.FLG_SIZE;
      // cvid field
      private static final int P_PVID = P_CVID + Types.VID_SIZE;
      // pvid field
      private static final int P_CLEN = P_PVID + Types.VID_SIZE;
      // clen field
      private static final int P_HLEN = P_CLEN + Types.LEN_SIZE;
      // hlen field
      private static final int P_MLEN = P_HLEN + Types.LEN_SIZE;
      // mlen field
      private static final int P_DLEN = P_MLEN + Types.LEN_SIZE;
      // dlen field
      private static final int P_HCHK = P_DLEN + Types.LEN_SIZE;
      // hchecksum field
      private static final int P_DCHK = P_HCHK + Types.CHK_SIZE;
      // dchecksum field
      private static final int P_DATA = P_DCHK + Types.CHK_SIZE;
      // Comparing data
```

```
////////////////////////////////////////////////////////////////////////
// Fields
////////////////////////////////////////////////////////////////////////

      boolean     flag;        // the flag of the proposal message
      long        cvid;        // Last complete view identifier
      long        pvid;        // Last partial view identifier
      int         last_plen;   // last_prop field length (in bytes)
      int   host_plen;   // host_prop field length (in bytes)
      int   memb_plen;   // memb_prop field length (in bytes)
      int   msgs_plen;   // msgs_prop field length (in bytes)
      int   last_start;  // last_prop field start position
      int   host_start;  // host_prop field start position
      int   memb_start;  // memb_prop field start position
      int   msgs_start;  // msgs_prop field start position
      int   host_pchk;         // checksum of host_prop
      int   msgs_pchk;         // checksum of msgs_prop
      byte[]      data;        // Message byte image
      byte[]      last_prop;   // Proposal section: last complete view
      byte[]      host_prop;   // Proposal section: view composition
                               // (hosts)
      byte[]      memb_prop;   // Proposal section: view composition
                               // (members)
      byte[]      msgs_prop;   // Proposal section: delivered messages


////////////////////////////////////////////////////////////////////////
// Constructors
////////////////////////////////////////////////////////////////////////

      /**
        * Creates a new instance of <code> MessageProp</code>
        * starting from a m-received message.
       *
       * @param               data        the m-received message.
       */

      MsgProp(byte[] data)
      {
            this.data = data;

            // Read single fields
            flag  = Types.byte2flg(data, P_FLAG);
            cvid  = Types.byte2vid(data, P_CVID);
            pvid  = Types.byte2vid(data, P_PVID);
            last_plen = Types.byte2len(data, P_CLEN);
            host_plen = Types.byte2len(data, P_HLEN);
            memb_plen = Types.byte2len(data, P_MLEN);
            msgs_plen = Types.byte2len(data, P_DLEN);
            host_pchk = Types.byte2chk(data, P_HCHK);
            msgs_pchk = Types.byte2chk(data, P_DCHK);

            // Computes starting positions
            last_start = P_DATA;
            host_start = last_start + last_plen;
            memb_start = host_start + host_plen;
            msgs_start = memb_start + memb_plen;
      }

      /**
```

```
     * Creates a new instance of <code> MsgProp</code> acting as a
     * container for a proposal.
     *
     * @param          data        the m-received message.
     */

    MsgProp()
    {
          last_prop          = new byte[0];
          host_prop          = new byte[0];
          memb_prop          = new byte[0];
          msgs_prop          = new byte[0];
    }


////////////////////////////////////////////////////////////////////////
// Encoding methods
////////////////////////////////////////////////////////////////////////

    /*
     * Encodes the last complete view and stores it into the
     * specified group.
     *
     * @param                cview       last complete view
     * @param                length      actual length of cview
     */

    void encodeLastProp(int[] cview, int length)
    {
          int   size;        // New byte array length
          int   pos;         // Position of data

          size = Types.GRP_SIZE + Types.HID_SIZE * length;
          if (size > last_prop.length)
                last_prop = new byte[size];
          last_plen = size;

          Types.grp2byte(last_prop, 0, length);
          pos = Types.GRP_SIZE;
          for (int i = 0; i < length; i++)
                pos = Types.hid2byte(last_prop, pos, cview[i]);
    }

    /**
     * Encodes the next view estimate (hosts) and stores it into
     * the specified group.
     *
     * @param   hdata       estimate array
     * @param   hsize       actuale length of the estimate array
     */

    void encodeHostProp(HostData[] hdata, int hsize)
    {
          int   size;        // New byte array length
          int   pos;         // Position of data

          size = Types.GRP_SIZE + (Types.HID_SIZE + Types.INC_SIZE
+ Types.VER_SIZE) * hsize;
          if (size > host_prop.length)
                host_prop = new byte[size];
          host_plen = size;
```

```
            Types.grp2byte(host_prop, 0, hsize);
            pos = Types.GRP_SIZE;
            for (int i = 0; i < hsize; i++)
                  pos = Types.hid2byte(host_prop, pos, hdata[i].key);
            for (int i = 0; i < hsize; i++)
                  pos = Types.inc2byte(host_prop, pos,
hdata[i].host.getIncarnation());
            for (int i = 0; i < hsize; i++)
                  pos = Types.hid2byte(host_prop, pos,
hdata[i].agreed);
            host_pchk = Types.computeChecksum(host_prop, 0, size);
      }


      /**
       * Encodes the next view estimate (members) and stores it into
       * the specified group.
       *
       * @param tmember list of members
       * @param msize   number of members in the current estimate
       */

      void encodeMembProp(MemberData tmembers, int nmembers)
      {
            int   size;       // New byte array length
            int   pos;        // Position of data
            MemberData  scan; // Used to scan <group.tmembers>

            size = Types.GRP_SIZE + Types.GRP_SIZE * nmembers;
            if (size > memb_prop.length)
                  memb_prop = new byte[size];
            memb_plen = size;

            Types.grp2byte(memb_prop, 0, nmembers);
            pos = Types.GRP_SIZE;
            scan = (MemberData) tmembers.getFirst();
            while (scan != null) {
                  if (!scan.leaving)
                        pos = Types.grp2byte(memb_prop, pos,
scan.key);
                  scan = (MemberData) scan.getNext();
            }
      }

      /**
       * Encodes the information about delivered messages and stores
       * it into the specified group.
       *
       * @param   hdata       estimate array
       * @param   hsize       actuale length of the estimate array
       */

      void encodeMsgsProp(HostData[] hdata, int hsize)
      {
            int   size;       // New byte array length
            int   pos;        // Position of data

            size = Types.MID_SIZE * hsize;
            if (size > msgs_prop.length)
                  msgs_prop = new byte[size];
```

```
            msgs_plen = size;

            pos = 0;
            for (int i = 0; i < hsize; i++)
                  pos = Types.mid2byte(msgs_prop, pos,
hdata[i].delivered);
            msgs_pchk = Types.computeChecksum(msgs_prop, 0, size);
      }


      /**
       * Encodes a MsgProp message  starting from raw data.
       *
       * @param   group object from which raw data are
       * extracted.
       */

      byte[] encode(boolean flag,int gid, long cvid, long pvid)
      {
            int   size;       // Contents length
            byte[]      data; // Mss message for sending
            int         pos;  // Position; used to scan array data
            int         i;    // Counter

// Allocates data buffer, which has been modified to allow for the
new flag variable

            size = Types.FLG_SIZE + Types.GID_SIZE + 2*Types.VID_SIZE
+ Types.LEN_SIZE * 4 + Types.CHK_SIZE*2 +
                              last_plen + host_plen + memb_plen +
msgs_plen;
            data = new byte[size];
            Debug.println(20, "Proposal flag at encode: " + flag);

            // Writes data into data buffer
            Types.flg2byte(data, P_FLAG, flag);
            Types.gid2byte(data, P_GID,    gid);
            Types.vid2byte(data, P_CVID, cvid);
            Types.vid2byte(data, P_PVID, pvid);
            Types.len2byte(data, P_CLEN, last_plen);
            Types.len2byte(data, P_HLEN, host_plen);
            Types.len2byte(data, P_MLEN, memb_plen);
            Types.len2byte(data, P_DLEN, msgs_plen);
            Types.chk2byte(data, P_HCHK, host_pchk);
            Types.chk2byte(data, P_DCHK, msgs_pchk);
            pos = P_DATA;
            System.arraycopy(last_prop, 0, data, pos, last_plen);
            pos += last_plen;
            System.arraycopy(host_prop, 0, data, pos, host_plen);
            pos += host_plen;
            System.arraycopy(memb_prop, 0, data, pos, memb_plen);
            pos += memb_plen;
            System.arraycopy(msgs_prop, 0, data, pos, msgs_plen);

            return data;
      }
```

```
////////////////////////////////////////////////////////////////////////
// Equality methods
////////////////////////////////////////////////////////////////////////

      /**
       * Compares the estimates part of two instances of MsgProp
       *
       * @param   msg   the reference MsgProp message with which to
       * compare.
       */

      boolean checkEstimate(MsgProp msg)
      {
            if (msg == null || host_plen != msg.host_plen ||
host_pchk != msg.host_pchk)
                  return false;
            for (int i=0; i<host_plen; i++)
                  if (data[host_start+i] !=
msg.data[msg.host_start+i])
                        return false;
            return true;
      }

      /**
       * Compares the messages part of two instances of MsgProp
       *
       * @param   msg   the reference MsgProp message with which to
       * compare.
       */

      boolean checkMessages(MsgProp msg)
      {
            if (msg == null)
                  return false;
            if (pvid == msg.pvid) {
                  if (msgs_plen != msg.msgs_plen || msgs_pchk !=
msg.msgs_pchk)
                        return false;
                  for (int i=0; i<msgs_plen; i++)
                        if (data[msgs_start+i] !=
msg.data[msg.msgs_start+i])
                              return false;
            }
            return true;
      }

////////////////////////////////////////////////////////////////////////
// Decoding methods
////////////////////////////////////////////////////////////////////////

      /**
       * Decodes the last complete view.
       */
      int[] decodeLastProp()
      {
            int[] ret;                        // Return value
            int   len;                        // Return value length
            int   pos;                        // Data position

            len = Types.byte2grp(data, last_start);
            ret = new int[len];
```

```
        pos = last_start + Types.GRP_SIZE;
        for (int i=0; i < len; i++) {
            ret[i] = Types.byte2hid(data, pos);
            pos += Types.HID_SIZE;
        }
        return ret;
    }

    /**
     * Decodes the estimate for the next complete view.
     */
    int[] decodeHostProp_hosts()
    {
        int[] ret;                      // Return value
        int   len;                      // Return value length
        int   pos;                      // Data position

        len = Types.byte2grp(data, host_start);
        ret = new int[len];
        pos = host_start + Types.GRP_SIZE;
        for (int i=0; i < len; i++) {
            ret[i] = Types.byte2hid(data, pos);
            pos += Types.HID_SIZE;
        }
        return ret;
    }


    /**
     * Decodes the agreed version numbers for the next complete
     * view.
     */

    int[] decodeHostProp_incarns()
    {
        int[] ret;                  // Return value
        int   len;                  // Return value length
        int   pos;                  // Data position

        len = Types.byte2grp(data, host_start);
        ret = new int[len];
        pos = host_start + Types.GRP_SIZE + len * Types.HID_SIZE;
        for (int i=0; i < len; i++) {
            ret[i] = Types.byte2inc(data, pos);
            pos += Types.INC_SIZE;
        }
        return ret;
    }


    /**
     * Decodes the agreed version numbers for the next complete
     * view.
     */

    int[] decodeHostProp_agreed()
    {
        int[] ret;              // Return value
        int   len;              // Return value length
        int   pos;              // Data position
```

```
            len = Types.byte2grp(data, host_start);
            ret = new int[len];
            pos = host_start + Types.GRP_SIZE + len * (Types.HID_SIZE
* Types.INC_SIZE);
            for (int i=0; i < len; i++) {
                  ret[i] = Types.byte2ver(data, pos);
                  pos += Types.VER_SIZE;
            }
            return ret;
      }

      /**
       * Decodes the list of members contained in the
proposal.
       */

      int[] decodeMembProp()
      {
            int[] ret;        // Return value
            int   len;        // Return value length
            int   pos;        // Data position

            len = Types.byte2grp(data, memb_start);
            ret = new int[len];
            pos = memb_start + Types.GRP_SIZE;
            for (int i=0; i < len; i++) {
                  ret[i] = Types.byte2grp(data, pos);
                  pos += Types.GRP_SIZE;
            }
            return ret;
      }

// DBEGIN
      void print()
      {
            Debug.print(20, "Msg: ");
            for (int i=0; i < data.length; i++)
                  Debug.print(20, " " + data[i]);
            Debug.println(20, "");
            Debug.println(20, "Proposal flag decoded: " +  flag);
      }
// DEND

} // END MsgProp
```

Bibliography

[Ami95] Y. Amir, **Replication Using Group Communication Over a Partitioned Network**. Ph.D. thesis, The Hebrew University of Jerusalem, 1995

[AMMAC95] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, P. Ciarfella, **The Totem Single-Ring Ordering and Membership Protocol.** University of California, Santa Barbara, 1995. Also available in ACM Trans. Computer Systems vol. 13, pp. 311-342, November 1995

[ACT95] E. Anceaume, B. Charron-Bost, and S. Toueg, **On the Formal Specification of Group Membership Services**. Technical Report TR95-1534, Department of Computer Science, Cornell University, August 1995

[BDM98] O. Babaoglu, R. Davoli, and A. Montresor, **Group Communication in Partitionable Systems: Specification and Algorithms.** Technical Report UBLCS98-01, Department of Computer Science, University of Bologna, April 1998

[BDM99] O. Babaoglu, R. Davoli, and A. Montresor, **Group Communication in Partitionable Systems: Specification and Algorithms.** Technical Report UBLCS98-01, Department of Computer Science, University of Bologna, April 1998, revised May 1999

[BDGB95] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker, ***RELACS*: A Communication Infrastructure For Constructing Reliable Applications in Large-Scale Distributed Systems.** Technical Report

UBLCS94-15, Department of Computer Science, University of Bologna, January 1995

[BCG91] K. Birman, R. Cooper and B. Gleeson. **Design Alternatives for Process Group Membership and Multicast**. Technical Report TR91-1185, Computer Science Department, Cornell University, December 1991

[CHT96] T., Channdra, V. Hadzilacos, and S. Toueg. **On the Impossibility of Group Membership**. Computer Science Department, Cornell University, May 1996. Also available in Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, pages 322-330, May 1996

[DBM96] R. Davoli, O. Babaglu, and A. Mostresor, **Group Membership and View Synchrony in Partitionable Asynchronus Distributed Systems: Specification**. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, September 1996

[DBM97] R. Davoli, O. Babaglu, and A. Mostresor, **Partitionable Group Membership: Specification and Algorithms**. Technical Report UBLCS-97-1, Department of Computer Science, University of Bologna, May 1997

[DMS95-1] D. Dolev, D. Malki, and R. Strong, **An Asynchronous Membership Protocol that Tolerates Partitions**. Institute of Computer Science, The Hebrew University of Jerusalem, 1995

[DMS95-2] D. Dolev, D. Malki, and R. Strong, **Framework for Partitionable Membership Service**. Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995. Also available in Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, May 1996

[DM95] D. Dolev, D. Malki, **The Design of the Transis System**. Institute of Computer Science, The Hebrew University of Jerusalem, 1995

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava, ***Newtop:* A Fault-Tolerant Group Communication Protocol.** Department of Computer Science, University of Newcastle, June 1995. Also available in Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, Canada, June 1995

[KSMD98] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev, **A Client-Server Oriented Algorithm for Virtually Synchronous Group Emmbership in WANs.** Department of Computer Science and Engineering, University of California, San Diego, 1998

[KSMD99] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev, **A Client-Server Oriented Algorithm for Virtually Synchronous Group Emmbership in WANs.** Proceedings of the 20th Department of Computer Science and Engineering, University of California, San Diego, June1999. Also available in Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)

[iBus] **iBus Toolkit**, http://www.softwired-inc.com

[Maffeis] Dr. S. Maffeis, VP SoftWired Inc. maffeis@softwired-inc.com

[Maf95] S. Maffeis, **Run-Time Support for Object Oriented Distributed Programming**, Ph.D. Thesis, University of Zuirch, February 1995

[Mont99] A. Montresor, **The Jgroup Reliable Distributed Model.** Technical Report UBLCS-98-12. Department of Computer Science, University of Bologna, March 1999. Also available in Proceedings of the 2nd IFIP

International Working Conference on Distributed Applications and Systems (DAIS'99), Helsinki, June 1999

[RBG92] R. Renesse, K. Birman, B. Glade et all. **HORUS: A Flexible Group Communication System**. Department of Computer Science, Cornell University, 1992.

[Ric93] A. Riccardi, **The Asynchronous Memberhsip Problem**. Ph.D. Thesis, Cornell University, January 1993

[RSB93] A. Riccardi, A. Schiper and K. Birman, **Understanding Partitions & the "No Partition" Assumption,** Espirit Basic Research Project 6360, Broadcast Technical Report, August 1993. Also available in Proceedings 4th IEEE Workshops on Future Trends of Distributed Systems, Lisboa, September 1993

[RB93] A. Riccardi, and K. Birman, **Process Membership in Asynchronous Environments**. Technical Report TR93-1328. Department of Computer Science, Cornell University, February 1993