System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems

Alberto Montresor

Technical Report UBLCS-2000-10

July 2000

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

Recent Titles from the UBLCS Technical Report Series

Dottorato di Ricerca in Informatica

Università di Bologna, Padova, Venezia

System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems

Alberto Montresor

January 2000

Coordinatore:	Tutore:
Prof. Özalp Babaoğlu	Prof. Özalp Babaoğlu

Redundancy is our main avenue of survival

Robert Silverberg, "Shadrach in the furnace"

Abstract

Distributed computing is one of the major trends in the computer industry. The growing interest in distributed applications, driven primarily by the popularization of the Internet, results in new challenges for software developers. As systems become more distributed, they also become more complex and more difficult to design. In order to deal with this complexity, new middleware systems for supporting the development of distributed applications are required.

The last few years have seen the emergence of several programming environments that greatly simplify the development of distributed applications. Among them, notable examples are CORBA and Java Remote Method Invocation (RMI). In order to abstract the complexity of the system and to promote modularity and reusability, these middleware platforms are based on object-oriented concepts like abstraction, encapsulation, inheritanceand polymorphism, and enable client/server interactions among distributed objects.

Existing object-oriented middleware environments focus their attention on improving portability, interoperability and reusability of distributed software components and applications. Unfortunately, none of them provides an adequate support for the development of dependable applications in the presence of partial failures. The main problem is the lack of "one-to-many" interaction primitives allowing clients to reliably invoke the same method on several objects at once.

In an effort to fill this void, the *object group* paradigm has been proposed. In this paradigm, functions of a distributed service are replicated among a collection of logically related server objects gathered together in an object group. A group constitutes a logical addressing facility: clients transparently interact with object groups by remotely invoking

methods on them, as if they were single, non-replicated remote objects.

In this thesis, we presents the design and the implementation of Jgroup, an extension of the Java distributed object model based on the group communication paradigm. Differently from other group-oriented extensions of existing distributed object models, Jgroup is expressly aimed at supporting the development of reliable and high-available distributed applications in *partitionable* environments. Jgroup enables the creation of groups of remote objects that cooperate towards some common goal using a partitionable group communication service. Remote object groups simulate the behavior of standard remote objects by implementing a set of remote interfaces and by enabling clients to remotely invoke the methods defined in these interfaces through the standard Java RMI mechanism. For the first time, communication among servers forming a group is based on group method invocations as well, offering a completely object-oriented framework for programmers. Finally, Jgroup is completed by a state merging service, which simplifies the development of state merging protocols needed to re-establish a consistent state after the disappearing of a partitioning.

Contents

List of Figures

Chapter 1

Introduction

Distributed computing is one of the major trends in the computer industry. The growing interest in distributed applications, driven primarily by the popularization of the Internet, results in new challenges for software developers. As systems become more distributed, they also become more complex and more difficult to design. In order to deal with this complexity, new middleware systems for supporting the development of distributed applications are required.

The last few years have seen the emergence of several programming environments that greatly simplify the development of distributed applications. Among them, notable examples are CORBA [37], Microsoft's DCOM [16, 71] and Java Remote Method Invocation (RMI) [52]. In order to abstract the complexity of the system and to promote modularity and reusability, these middleware platforms are based on object-oriented concepts like abstraction, encapsulation, inheritance and polymorphism, and enable client/server interactions among distributed objects: server objects encapsulate an internal state and make it accessible through a well-defined interface; client objects are allowed to access services provided by server objects by issuing remote method invocations on them. Remote method invocations are handled by local proxy objects called *stubs*, that deal with all low-level details of invocations.

Existing object-oriented middleware environments focus their attention on improving portability, interoperability and reusability of distributed software components and applications. Unfortunately, none of them provides an adequate support for the development of dependable applications in the presence of partial failures. The main problem is the lack of "one-to-many" interaction primitives allowing clients to reliably invoke the same method on several objects at once. This interaction style may greatly simplify the development of several types of applications with reliability and high-availability requirements. Its lack constitutes a major drawback for many modern industrial applications, for which these requirements are gaining increasing importance [36]. In the absence of any systematic support, building applications capable to deal with partial failures such as crashes and partitionings is an error-prone and time-consuming task.

In an effort to fill this void, the *object group* paradigm has been proposed [20]. In this paradigm, functions of a distributed service are replicated among a collection of logically related server objects gathered together in an object group. A group constitutes a logical addressing facility: clients transparently interact with object groups by remotely invoking methods on them, as if they were single, non-replicated remote objects. A method invocation on a group results in the method executed by one or more of the servers forming the group, depending on the invocation semantics. To distinguish these multi-peer invocations from standard "point-to-point" method invocations, we call them *group method invocations*. Servers forming a group cooperate in order to provide a more dependable version of the service to their clients. Cooperation among servers is achieved through a *group membership service* and a *reliable communication service* [14, 49, 76, 5, 9], that enable the creation of dynamic object groups and provide primitives for sending messages to all servers in a group, with various reliability and ordering guarantees.

1.1 Problem Statement

Modern communication networks may be very often characterized as *partitionable*: the system can break into several partitions, where the objects of a partition can communicate only with objects inside the partition itself. Partitionings tend to become more frequent and longer-lasting as the geographical extent of the network grows, or its connectivity weakens due to the presence of wireless links and mobile units. In addition to involuntary partitionings caused by failures, mobile computing systems have to deal with partitions when units are deliberately unplugged from the network.

When developing applications in partitionable systems, the main problem that has to be confronted is the possibility that the computation evolves in a inconsistent way in distinct partitions. One possible solution is to restrict services to a single (*primary*) partition and to block non-primary partitions until they merge back to the primary one. In this way, the computation is enabled to evolve in a single partition, thus excluding the possibility of inconsistencies. Most of existing object group systems [47, 30, 63, 12] follow this approach.

Although there is a wide class of applications that can benefit from the primary-partition model (in particular, those with strong consistency requirements), applications with high-availability requirements cannot be based on this approach. As an example, consider a distributed booking service that manages sets of "resources" (for example, tickets) that can be booked by clients. In case of partitioning, minority partitions are not allowed to book resources until communication is restored. Moreover, particular failure scenarios may cause the temporary loss of the primary partition, and consequently the blocking of the application. This outcome is not acceptable for a booking service.

The problem lies mainly on the fact that trying to mask partitions from applications restricts the possibilities of intervention of applications themselves. In other words, applications based on the primary-partition paradigm have no knowledge about partitions and thus cannot take advantage of their semantics in order to provide a more reliable and high-available service.

Apart from the availability issue, existing object group systems suffer also from another problem: they fail to maintain their promise to provide a *completely* object-oriented environment to software developers. On the contrary, remote method invocations are normally used only for interactions between external clients and object groups, while communication within a group of server objects is based on message exchanges. The use of different communication paradigms for programming service providers and service consumers compromises the advantages of the object-oriented environments associated with existing object group systems. Application developers are required to deal with low-level details as message marshaling, unmarshaling and dispatching.

1.2 Thesis contributions

In order to overcome these problems, we have designed Jgroup, an extension of Java RMI aimed at providing systematic support for the development of dependable applications in partitionable systems [9, 58, 57, 59]. In Jgroup, a *partitionable* group membership service monitors the system and keeps group members consistently informed about the current failure scenario. Applications may exploit the information obtained by the group membership service and consequently adapt their behavior in order to keep offering their services in multiple partitions of the system. Applications behaving in this manner are said *partition-aware*. Examples of applications that can benefit from a *partition-aware* approach can be found in areas such as mobile computing (distributed file systems with disconnected operations), weak consistency data sharing (directory services, resource allocation), computer-supported cooperative work (shared blackboards, distributed agendas), and time-intensive parallel computing (scientific computing, weather forecasting, ray tracing).

Changing the behavior of an application may lead to modifications to the quality of the offered services: in other words, partitionings may result in service reduction or degradation, but do not necessarily render application services completely unavailable. In the booking service example, it might be more desirable to distribute the resources among the servers implementing the application and allow clients in each partition to book the resources actually present in the partition itself (thus, reducing the quality of the service, but without blocking the service outside the primary partition).

Clearly, enabling servers in a group to carry on the computation independently may cause the state maintained by servers in different partitions to evolve in a inconsistent way. When two or more partition merge, servers need to establish a *state merging protocol* which reconciles, whenever possible, the divergence that may have taken place. Note that deciding which information has to be exchanged in this protocol is an application-dependent problem. Nevertheless, it is possible to support developers in designing protocols for diffusing this information. In Jgroup, this support is provided by the *state merging service*.

Finally, Jgroup differs from existing object group systems also for another characteristic: it offers an uniform communication interface entirely based on the remote method invocation paradigm. Servers included in a group interact with other servers by remotely invoking methods on the group itself, in the same way as clients do. In this manner, the benefits of object-oriented programming are extended also to internal communication among members.

The aim of this thesis is to discuss the programming model of Jgroup and to present its design and specification. A prototype implementation of Jgroup has been realized in the context of this thesis. The prototype is completely written in Java, and thus is fully portable in different computer architectures, including Linux, Solaris and Windows. We are currently using the Jgroup prototype for the development of demonstrative applications such as partition-aware automatic teller machines.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 recalls some background concepts about object-oriented distributed computing and group communication. This chapter is intended as an introduction to the basic notions needed in the rest of the thesis, while a detailed comparison between our work and existing object group systems is postponed to Chapter 7.

Chapter 3 presents an overview of the Jgroup distributed object model. This chapter constitutes also a tutorial for the use of Jgroup, and describes the main interfaces and classes needed by software developers to program applications based on Jgroup. The main components of the system are described, including the partitionable group membership service, the group method invocation service, and the state merging service.

A formal specification of Jgroup is given in Chapter 4. The chapter starts with a formal description of the distributed system for which Jgroup has been designed, and continues by presenting and discussing the specification of the components of Jgroup identified in Chapter 3. The presence of a formal specification allows software developers to reason about the correctness of their applications without knowing any detail of the Jgroup implementation.

Chapter 5 describes the internal architecture of Jgroup, together with a collection of psuedo-code algorithms implementing the main components of the Jgroup prototype.

Chapter 6 contains some examples of how Jgroup can be used to implement realistic applications. These examples include a replicated registry service (a repository facility included in the Jgroup distribution used by servers to advertise their availability to provide some service, and by clients to locate these services); a partitionable execution service for distributing parallel computations over a network of workstation; and finally, a partition-aware resource management service, which may be used to implement the distributed booking service described in this introduction.

As introduced above, Chapter 7 discuss how our work can be compared with other projects. In particular, three different aspects are taken in consideration: the relation between the specification of the group communication facility contained in Jgroup and other group communication specifications ; a comparison between the algorithm implemented by Jgroup and other algorithms discussed in the literature; and finally, the relation between the programming model associated with Jgroup and other object group systems.

The concluding chapter summarizes the main results of this thesis, and discusses the future of Jgroup. The thesis is concluded by two appendix sections, one containing a proof of the correctness of our algorithms, and the other presenting the application programming interface (API) of Jgroup.

Chapter 2

Background

The Jgroup distributed object model is based on the *object group* paradigm. This paradigm integrates two consolidated technologies for the development of distributed applications: *object-oriented distributed programming* to support distribution, reusability and interoperability, and *group communication* to support dependability. In this chapter, we briefly introduce the fundamental notions behind these technologies. The presentation of the object group model is postponed to the next chapter, while presenting the Jgroup distributed object model. A comparison between Jgroup and existing object group systems may be found in Chapter 7.

2.1 Object-Oriented Distributed Programming

Object-oriented programming is considered one of the today's best programming models to cope with complex systems while providing maintainability, extensibility and reusability. A model claiming such attributes is particular interesting for distributed systems, as these tend to become very complex.

The basis of object-oriented programming is the concept of *object*, which is an entity that encapsulates a state and makes it accessible through a well-defined *interface*. An object is an instance of a *class*. Behind the concept of object there are four basic notions:

- *Abstraction* is the process of identifying the characteristics that distinguish a collection of similar objects into a class. It is the concise representation of a more abstract idea.

- *Encapsulation* is the mechanism by which external aspects of an object are separated from its implementation: objects contain specific internal data, which is accessible only through the methods contained in its interface. By decoupling the interface and the implementation, encapsulation enables programmers to modify an object's implementation without affecting clients accessing the object.

- *Inheritance* is the mechanism by which an object interface may include the attributes and methods defined in another, more general interface. It is thus possible to reuse the code of existing implementations. The resulting class can be specialized by modifying inherited methods, or by adding new ones.

- Finally, *polymorphism* is the ability to substitute objects with matching interfaces for one another at run-time. With polymorphism, objects that have common descriptions can respond to the same request with different actions, depending on their type. Polymorphism makes it possible to design applications that can be easily extended.

Object-oriented, *distributed* programming (OODP) generalizes object-oriented programming for distributed systems, by extending the object model with remote client/server interactions between non-local objects. In the client/server model, servers provide clients with access to specific services through IPC mechanisms such as message passing or remote procedure call. In object-oriented distributed programming, a server object encapsulates an internal state and provides a service to a collection of *remote* client objects. As in pure object-oriented programming, this service is abstracted through an interface, i.e. a collection of methods that can be used to access the service. Clients residing on different machines may invoke these methods through *remote method invocations*.

The last few years have seen the emergence of several development environments based on the object-oriented distributed programming paradigm. Among them, the most notable examples are CORBA [37] and Java Remote Method Invocation (RMI) [52]. Here, we want to discuss the major characteristics of both these systems. Particular emphasis is laid on Java RMI, as it constitutes the basic building block for Jgroup.

2.1.1 CORBA

The Object Management Group (OMG) is an international industry consortium that promotes the theory and practice of object-oriented distributed programming. Their goal

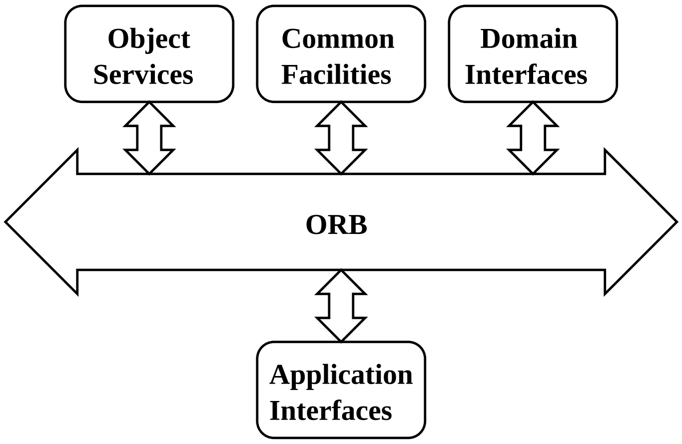

Figure 2.1: OMG's Object Management Architecture

is to provide a common architectural framework, across heterogenous hardware platforms and operating systems, for inter-communication among objects. The OMG was founded in 1989 by eight companies, and now boasts over 800 members, with almost every significant computer vendor represented [37]. The organization performs no development work, relying rather on existing technologies offered by member companies. Its approach is to issue *request for proposals* (RFPs) on all aspects of object technology, which solicit specifications of new components. Members may then propose a specification that is accompanied by an implementation proving the concepts detailed therein. A review and voting process is conducted and once a specification is accepted, any vendor that conforms to it may offer alternative implementations to market.

The OMG has developed a conceptual model, known as the *core object model*, and a reference architecture, called the *object management architecture* (OMA) upon which applications can be constructed. The OMG OMA attempts to define, at a high level of abstraction, the various facilities necessary for distributed object-oriented computing. The OMG OMA partitions the OMG problem space into practical, high-level architectural components. The five major parts of the OMA reference model are the following (Figure 2.1):

- The *Object Request Broker* (ORB, but also commercially known as CORBA) is a common communication bus for objects. It enables client objects to transparently and reliably invoke operations on remote objects and receive replies in a distributed environment. Integration of distributed objects is available across platforms, regardless of networking transports and operating systems.

- *Object Services* are general purpose components that are fundamental for developing useful CORBA applications. An object service is basically a set of CORBA objects that can be invoked through the ORB. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments, supporting basic functionalities useful for most applications. Several services have been designed and adopted as standards by the OMG [37], including event notifications, persistent storage of state, transaction processing interfaces, security and authentication control, etc.

- *Domain Interfaces* represent vertical areas that provide functionality of direct interest to end-users in specific application domains, such as finance or health care.

- *Common Facilities* provide end-user capabilities useful across many application domains, that can be configured to the specific requirements of a particular application. These are facilities that sit close to the user, such as printing, document management, and electronic mail facilities.

- *Application Interfaces* are specific to end-user applications. They represent component-based applications performing specific tasks for a user. An application is typically composed of a large number of objects, some of which are application-specific, and others part of object services, common facilities, or domain interfaces.

It is required that all OMG services be specified using the OMG *interface definition language* (IDL). This language is purely declarative, and provides no implementation details. It is programming language neutral and network neutral, and thus it is used as a means of describing data types. The syntax of OMG IDL is derived from C++, removing the constructs of an implementation language and adding a number of new keywords required to specify distributed systems.

2.1.2 Java Remote Method Invocation

Java Remote Method Invocation (RMI) is a distributed object model for the Java language that retains the semantics of the Java object model, making distributed objects easy to

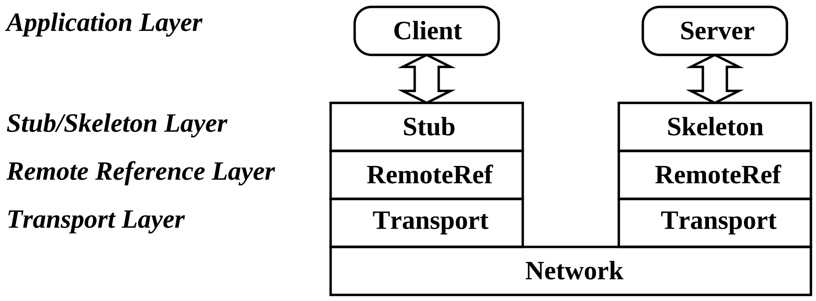

Figure 2.2: The Java RMI Architecture

implement and to use. In the standard Java RMI terminology, a *remote object* is one whose methods can be invoked from another Java virtual machine. An object of this type is described by one or more *remote interfaces*, which are Java interfaces that declare which methods of the remote object may be invoked remotely. *Remote method invocation* refers to the action of invoking a method of a remote interface on a remote object.

In Java RMI, remote interfaces must satisfy the following requirements:

- A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote, which is a marker interface that defines no methods.
- Each method declaration in a remote interface must satisfy the requirements of a remote method declaration as follows:
  - A remote method declaration must include the exception java.rmi.RemoteException in its *throws* clause, in addition to any application-specific exceptions. Remote exceptions are thrown when a remote method invocation fails for some reason, such as communication failures (unreachable servers, servers refusing the connection, etc.) or failures happening during parameter marshaling or unmarshaling.
  - In a remote method declaration, a remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object) must be declared as the remote interface, not the implementation class of that interface.

The Java RMI architecture is illustrated in Figure 2.2. Java RMI uses a standard mechanism (derived from RPC) for communicating with remote objects: *stubs* and *skele-*

tons. A stub for a remote object acts as a client's local representative for the remote object. Clients never interact directly with remote objects, but only with stub objects which are responsible for carrying out the method invocation on their remote counterpart. A skeleton for a remote object is a server-side entity that contains a method which dispatches invocations to the actual remote object implementation. Stubs and skeletons are generated by the `rmic` compiler provided with the standard Java RMI distribution.

A stub for a remote object implements the same set of remote interfaces implemented by the remote object. When a stub's method is invoked, it does the following:

- initiates a connection with the remote Java virtual machine containing the remote object;
- marshals (writes and transmits) the parameters to the remote Java virtual machine;
- waits for the result of the method invocation;
- unmarshals (reads) the return value or exception returned;
- returns the value to the caller.

In the remote Java virtual machine, each remote object may have a corresponding skeleton (in JDK 1.2-only environments, skeletons are not required). The skeleton is responsible for dispatching the invocation to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshals (reads) the parameters for the remote method;
- invokes the method on the actual remote object implementation;
- marshals the result (return value or exception) to the invoker.

In JDK 1.2 an additional stub protocol was introduced that eliminates the need for skeletons. Instead, generic code is used to carry out the duties of skeletons.

Each stub contains a *remote reference*, that can be seen as an handle for the remote object and is responsible for the semantics of the invocation. The current version of Java RMI includes only two unicast (point-to-point) invocation mechanisms, one relative to servers always running on some machine, and one relative to servers that are activated only when one of their methods is invoked. No multicast invocation mechanism is provided.

Before a client may invoke the methods of a remote object, it must obtain a stub for it. For this reason, the Java RMI architecture includes a repository facility called *registry* that can be used to retrieve remote object stubs by simple names. Each registry maintains a set of bindings ⟨*name*, *remote object*⟩; new bindings can be added using method bind, while method lookup method is used to obtain the stub for a remote object registered under a certain name. Since registries are remote objects, the Java RMI architecture includes also a bootstrap mechanism to obtain registry stubs.

2.2 The Group Communication Paradigm

The group communication paradigm [13] allows the provision of reliable and high-available applications through replication. *Groups* are the key abstraction of group communication. A group consists of a collection of *members* (i.e., processes or objects) that share a common goal and actively cooperate in order to reach it.

During the last years, several experimental and commercial group communication systems have appeared [14, 76, 5, 61, 9, 3, 49]. Although the services provided by them present several differences, the key mechanisms underlying their architectures are the same: a *group membership service* integrated with a *reliable multicast service*. The task of the group membership service is to keep members consistently informed about changes in the current membership of a group through the *installation* of *views*. The membership of a group may dynamically vary due to voluntary requests to join or leave a group, or to accidental events such as member's crashes. Installed views consist of a collection of members and represent the perception of the group's membership that is shared by its members. In other words, there has to be agreement among members on the composition of a view before it can be installed. The task of a reliable multicast service is to enable members of a group to communicate by multicasting messages. Message deliveries are integrated with view installations as follows: two members that install the same pair of views in the same order deliver the same set of messages between the installations of these views. This delivery semantics, called *view synchrony*, enables members to reason about the state of other members using only local information such as the current view composition and the set of delivered messages.

2.2.1 Primary-Partition Group Communication

As noted in the introduction, two classes of group communication services have emerged: *primary-partition* [14, 49] and *partitionable* [76, 5, 61, 9, 3]. A primary-partition group communication service attempts to maintain a single agreed view of the current membership of a group. Members excluded from this primary view are not allowed to participate in the distributed computation. Primary-partition group communication services are suitable for non-partitionable systems, or for applications that need to maintain a unique state across the system. The most notable examples of primary-partition group communication services are Isis [14] and Phoenix [49].

Isis

The Isis toolkit [14], developed at Cornell University, was the first group communication system to support view synchrony. Isis is a library of procedural tools that are linked directly to the application program, providing it with functionalities for dynamically creating and joining process groups, and multicasting messages to process groups with various ordering guarantees. In particular, Isis provides primary-partition group membership and virtual synchrony with FIFO, causal and total-order multicast primitives.

Phoenix

Phoenix [49], developed at the Swiss Federal Institute of Technology at Lausanne, is a group communication toolkit that provides primary-partition group membership and view synchrony in asynchronous, large scale environments (those characterized by geographical distribution and a high number of member processes). Phoenix implements dynamic routing, datagram-oriented reliable communication channels, group communication primitives and view synchrony. It supports a high number of processes by assigning them different roles: *core members*, *clients* and *sinks*. Core members manage shared state and have the strongest reliability guarantees with respect to message delivery and membership changes. They communicate with each other using group multicasts. Clients interact with members by sending requests to them, and receiving replies and view changes. Finally, sinks only receive information diffused by the members of the group. The Phoenix API provides object-oriented abstractions for supporting object groups instead of process groups.

2.2.2 Partitionable Group Communication

In contrast, a partitionable group communication service allows multiple agreed views to co-exist in the system, each of them representing one of the partitions in which the network is subdivided. Members of a view are allowed to carry on the distributed computation separately from the members not included in the view. Partitionable systems are intended for applications that are able to take advantage of their knowledge about partitionings in order to make progress in multiple, concurrent partitions. As we have said in the introduction, applications with these characteristics are called *partition-aware* [10]. Examples of applications that can benefit from a partition-aware approach can be found in areas such as computer-supported cooperative work (CSCW), mobile systems, weak-consistency data sharing.

Totem

The Totem system [61], developed at the University of California, Santa Barbara, is a set of communication protocols for the construction of fault-tolerant distributed systems. Totem is intended for complex applications in which both reliability and real-time performance are critical. The Totem system handles process failures and recovery, network partitioning and remerging (with continued operations in all parts of a partitioned system). Furthermore, Totem provides several ordered multicast primitives to process groups, with high throughput and low, predictable latency. These protocols exploit the hardware multicast capabilities of local-area networks and the locality of process groups in order to provide soft real-time guarantees.

Horus

The Horus project [76] was originally launched as an effort to redesign the Isis toolkit in a set of small, clearly defined modular units. Horus provides a virtual synchrony model that lets an application make progress even in minority partitions, indicating to the application whether it is currently part of the primary partition or not. One of the most interesting features of Horus lies in its communication architecture that treats a protocol as an abstract data type. Protocol layers can be stacked on top of each other in a variety of ways at runtime. Horus provides a *protocol composition framework* that allows custom protocols to be built from composing existing ones; this architecture has the additional advantage that an application only pays for the properties it uses.

Ensemble

Recently, a new generation of the Horus system has appeared, called Ensemble [39]. Ensemble is a high-performance, reconfigurable network protocol architecture intended for adaptive applications. The basic functionality of Ensemble is to track the membership of groups and to provide communication support among group members, and from non-members to a group. Depending on the nature of the application, the user may work with Ensamble directly, as a tool supporting data replication, collaboration, or coordination. However, Ensemble can also be used to control other technologies, such as the endpoints of standard communication sockets layered over IP.

Transis

The Transis communication system [23, 24, 25, 26], developed at the Hebrew University of Jerusalem, supports process group communication with several forms of group multicast operations: FIFO ordered, causally ordered, ordered, totally ordered, and safely delivered. Transis contains a protocol for reliable message delivery that optimizes the performance for existing network hardware and tolerates network partitionings. Recently, the Transis group has developed protocols for scaling group membership in wide-area networks [5, 43].

Chapter 3

The Jgroup Distributed Object Model

As we have seen in the previous chapter, existing middleware environments as Java RMI and CORBA focus their attention on improving portability, interoperability and reusability of distributed software components. Unfortunately, none of them adequately support the development of *reliable* and *high-available* distributed applications in the presence of partial failures. The main problem with which developers have to confront is the lack of reliable "one-to-many" interaction primitives needed to support replication (and thus dependability) in distributed systems. These primitives may be used by clients to reliably invoke the same method on several servers at once, and by servers to reliably communicate with other servers in order to coordinate their activities. The lack of this interaction style constitutes a major drawback for many modern industrial applications, for which requirements such as reliability and high-availability are gaining increasing importance. This fact has been recognized also by the OMG, that has recently issued a *request for proposals* (RFP) on fault tolerance [36]. In the absence of any systematic support, building applications capable to deal with partial failures is an error-prone and time-consuming task. In particular, developing dependable applications capable to deal with partitionings is even more difficult, as programmers have to take into consideration the possibility that the states of servers included in distinct partitions evolve in inconsistent ways. The aim of Jgroup is to fill this void by supporting the development of dependable applications in partitionable systems and to provide a comprehensive object-oriented environment to compose high-quality distributed applications out of reusable software components.

This chapter provides an overview of the Jgroup distributed object model. We start out with a brief description of the underlying system model, needed to understand the framework in which Jgroup operates. The complete formal specification of the system model may be found in Section 4.1. Then, we introduce the object group paradigm, that constitutes the basis of the Jgroup distributed object model. Finally, we present the Jgroup architecture, and we conclude by illustrating the main elements composing it.

3.1 Underlying System Model

We consider an asynchronous distributed system consisting of a collection of *Java virtual machines* (JVM) that communicate through message-passing. Being asynchronous, there are no constraints on the relative speed at which JVMs make progress and on the message transmission delays. Furthermore, neither an exact synchronization of the local clocks nor a reasoning based on "global time" is possible. Practical distributed systems often have to be considered as being asynchronous since transient failures, unknown scheduling strategies and variable loads on the computing and communication resources make it impossible to bound delays.

Each JVM hosts a collection of objects, an object being an instance of a class, and a class being an implementation of an abstract data type. An application can be composed of a considerable number of objects, but only few of them are actually *active*, i.e. involved in communication with other objects residing in remote JVMs. Since we are only interested in describing the distributed behavior of our system, in the rest of this work we restrict our attention on these objects. In order to simplify what follows, we distinguish between two main roles for objects. A *server object* (or *server* for brevity) is one whose methods may be accessed by other objects residing in remote JVMs through *remote method invocations*. A *client object* (or *client* for brevity) is one that accesses the methods of another object residing in a remote JVM. It is important to note that, as objects invoke methods on each other in an unrestricted fashion, client or server are only roles that objects play at particular moments of their existence. In other words, it is possible that a server object acts also as a client of a different service.

Neither the computing nor the communication infrastructure are reliable. A JVM may fail by *crashing*, together with the objects hosted in it, for example due to the crash of the computer node in which it is running. Furthermore, communication failures (e.g., link crashes, buffer overflows or incorrect/inconsistent routing tables) may disable communica-

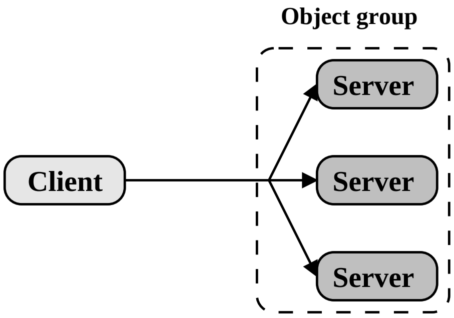

Figure 3.1: A client interacting with an object group.

tion between JVMs, leading to messages being dropped. Particular communication failure scenarios called *partitionings* may involve multiple sets of JVMs for extended periods of time: a partitioning disables communication among different *partitions*, each containing a set of JVMs. JVMs within a given partition can communicate among themselves, but cannot communicate with JVMs outside the partition. When communication between two or more partitions is reestablished, we say that they *merge*.

3.2 The Object Group Paradigm

In an effort to overcome the lack of support for the development of object-oriented applications with dependability requirements, the *object group paradigm* has been proposed [20, 48]. In this paradigm, functions of a distributed service are replicated among a collection of logically related server objects gathered together in an *object group* (Figure 3.1). Server objects forming a group implement the same set of remote interfaces and coordinate their activities in order to appear as a single server.

For clients, a group constitutes a logical addressing facility: the services offered by a group may be accessed as if it were a single, non-replicated entity, without having to know the number, location or identity of the individual servers. Communication between clients and groups takes the form of remote method invocations. A method invocation on a group results in the method executed by one or more of the servers forming the group, depending on the invocation semantics. In order to preserve transparency, only one result value is selected and returned to the client performing the invocation. To distinguish these multi-peer invocations from standard "point-to-point" method invocations, we call them *group method invocations* (GMI). For clients, group method invocations cannot be distinguished from standard RMI interactions: clients obtain a representative object called stub which acts as a proxy for a group, and perform invocations on it; stubs take care of all low-level details of group method invocations, such as locating the servers composing the group, contacting them and selecting a return value for the invoker.

In the last few years, the object group paradigm has been the subject of intense investigation, and several object group systems have appeared [47, 46, 31, 64, 12]. In general, existing object group systems may be described as composed of:

- A *primary-partition group membership service*, whose aim is to enable servers to create, join and manage groups of server objects. Servers included in the primary partition install a *primary view* and are allowed to carry on the computation. Servers not belonging to the primary view are logically blocked until their partitions merge the primary one.

- A *primary-partition reliable communication service*, generally composed of a *group method invocation service* and a *reliable multicast service*. The former is used by clients to perform group method invocations on the servers included in the primary partition, while the latter is used among servers for communication internal to the group. Both these service are generally based on *virtual synchrony*, which guarantees that messages exchanged among servers included in the primary view are delivered following a total order and are synchronized with respect to view changes [15].

- A *state transfer service*: Since the members of a group usually maintain a shared state, new members of a group and servers that have been excluded from the primary view need to receive the current state of the computation when dynamically joining it. This is achieved by the state transfer service that transmits the state from a server currently included in the primary view to a new member. The state transfer is generally driven by the object group system, that calls back to the application to obtain the state. Thus, group members must provide methods for "getting" and "setting" their current state.

Unfortunately, existing object group systems do not provide an adequate support for the development of applications with high-availability requirements to be deployed in partitionable systems. In the presence of partitionings, the main problem that must be

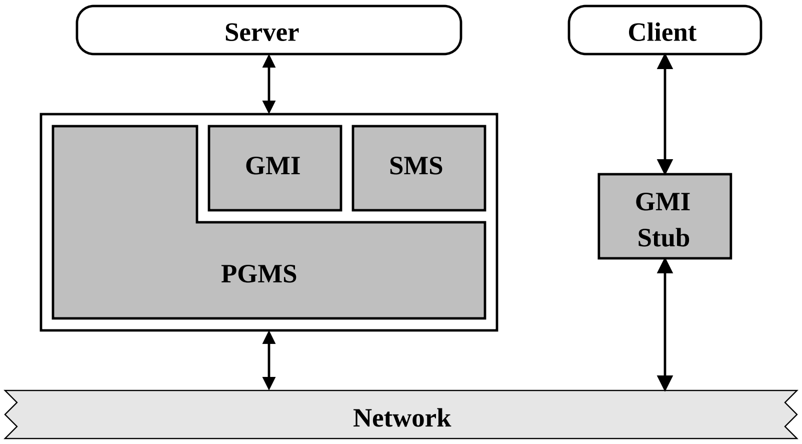

Figure 3.2: The Jgroup architecture

confronted by developers is the possibility that distinct partitions maintain inconsistent states. In the primary-partition approach, this problem is avoided by guaranteeing that computation is carried on in at most one partition. When consistency is the primary requirement of an application, primary-partition object group systems are the right choice. For applications with high-availability requirements, however, blocking the service in partitions different from the primary is not acceptable. Another problem arises when the primary partition disappears (for example, due to the impossibility of defining a majority partition among those present in the system), leading to the complete blocking of the application[49].

3.3 Jgroup: a Partition-Aware Approach to Object Groups

The Jgroup distributed object model designed in the context of this thesis is an attempt to extend the object group paradigm in order to support the development of partition-aware applications. In order to meet this goal, several issues have to be confronted:

- When developing partition-aware applications, servers included in a view must be aware of the presence of partitionings in the system. Primary-partition group membership services are not suitable, as servers included in partitions different from the primary stop being notified whenever the current failure scenario change. On the contrary, all servers participating in a partition-aware computation need to be continuously informed about the current partitioning scenario, so as to reconfigure themselves and adjust their behavior consequently.

- In an object group system (and in general in every group communication system), the behavior of the reliable communication service strongly depends on the group membership service adopted. With the primary-partition approach, all communications (both group method invocations coming from clients and message multicast exchanged among servers) are addressed to the servers included in the primary view. Application design is simplified, as there is a single communication stream in the system. Partition-aware applications, however, may have different requirements. Each server must be able to communicate with servers included in its view, and clients must be able to contact at least the servers included in their partition. Each communication involving the group should follow the *view synchrony* semantics: two servers that install the same pair of consecutive views in the same order deliver the same set of communication events in the period occurring between the two installations.

- In the partition-aware approach to group communication, there is the possibility that servers in concurrent partitions continue serving external operation independently and become inconsistent. When the conditions leading to the partition are repaired, an application-specific state merging protocol has to be executed to define a new global state that reconciles the divergence that may have taken place. In general, state merging protocols are more complex than the corresponding primary-partition state transfer protocols: the former are based on a bi-directional exchange of information between two merging partitions, while the latter are based on a transfer of information from servers already belonging to the primary view to servers dynamically joining it.

Having different requirements, partition-aware applications cannot be based on traditional object group systems based on the primary-partition approach. For this reason, Jgroup provides the *partitionable* counterparts of the primary-partition services listed in the previous section.

- *Partitionable Group Membership Service* (PGMS): The task of PGMS is to keep members continuously informed about the current failure scenario, including both crashes and partitionings. Partition-aware applications are programmed so as to reconfigure themselves and adjust their behavior based on the composition of installed views. In the absence of partitionings, every correct server should install the same view, and this view should include exactly those servers that have not crashed. If a partitioning occurs, our partitionable group membership service guarantees that under certain stability conditions, correct servers within the same partition install identical views and that their composition correspond to the composition of the partition itself.

- *Group Method Invocation Service* (or GMIS for brevity): The communication service included in Jgroup is strictly integrated with PGMS, and thus offer a partition-aware service. Clients are enabled to interact with servers belonging to their current partition, and servers included in a view are enabled to coordinate their behavior using the facilities of the group communication services. Differently from existing object group systems, Jgroup provides only a group method invocation service, that manages all communications involving members of a group. In other words, not only clients interact with server groups by remotely invoking methods on it, but also interactions internal to the group are based on the same paradigm. As they are characterized by different properties, we call *external* group method invocations those performed by clients, and *internal* group method invocations those performed by servers. The multi-invocation/multi-reply invocation service included in Jgroup has many advantages with respect to multicast message-based group communication. The development of complex partition-aware consistency protocols is simplified since operations like marshaling, unmarshaling and dispatching of invocations are handled automatically by GMIS. Applications programmers may take advantage of the object-oriented programming paradigm suggested by internal invocations and Jgroup; in particular, mechanisms such as encapsulation, inheritance and polymorphism may be particularly useful when developing complex applications. Finally, the unified communication model allows a more compact specification with respect to object group systems that mix message multicast and remote method invocation.

- *State Merging Service* (SMS): Task of SMS is to support developers in the design and the implementation of state merging protocols for their applications. In a similar way to state transfer facilities used in a primary-partition object group system, SMS takes care of calling back to the servers in order to obtain information about the current state of the computation and diffusing them to other partitions. Group members must provide operations for "getting" the current state of the computation from a partition, and "merging" it with the state of other partitions.

The partitionable group membership service, the group method invocation service and the state merging service constitutes the basic components of the Jgroup architecture, illustrated in Figure 3.2. The aim of the Jgroup model is to comprise a *minimal set of requirements* which a toolkit for object-oriented programming of partition-aware applications should fulfill. The Jgroup toolkit, however, may be easily extended with further layers providing additional facilities that may be extremely useful for programmers. As an example, consider an invocation ordering service, that enables objects to perform invocations following causal or total order.

In the next sections, we examine closely the characteristics of the components described above by summarizing their specifications and by analyzing their programming interface. Unless differently specified, all interfaces and classes described in the rest of this chapter belong to the `jgroup` package of the Jgroup distribution. For those interested in the complete specification of our services, please refer to Section 4; the complete application programming interface is included in Appendix B.

3.4 The Partitionable Group Membership Service

The partitionable group membership service included in Jgroup is based on our previous work on group communication [9]. Its specification has benefited from extensive reflection based on actual experience with programming realistic applications and has gone through numerous refinements over the last several years [7, 8, 9]. Here, we discuss briefly its main characteristics, while the complete specification is contained in Section 4.2. Note that our partitionable group membership service may be exploited separately from the group method invocation service described in the next section. In other words, it is possible to make use of our PGMS together with other communication facilities, as well as alone as a sophisticated failure detection facility. For this reason, in this section the discussion is limited to those features that are independent from the communication service used.

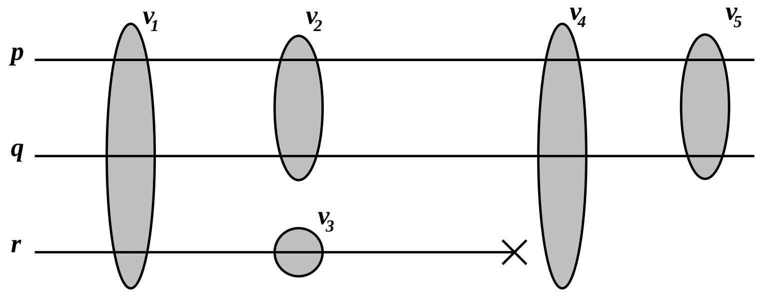

Figure 3.3: The group membership service at work.

As we have seen, groups are collections of servers that cooperate towards some common goal. For increased flexibility, the group composition is allowed to vary dynamically as new servers are added and existing ones removed. Servers willing to participate in a distributed computation have to become *member* of a group by *joining* it. Later on, a member may decide to terminate its collaboration by *leaving* the group. At a certain point in time, the *membership* of a group comprises those servers that have joined but have not left the group. Due to the problems caused by the asynchrony of the systems and given the possibility of failures, each member may have a different perception of the current membership of a group. The task of a partitionable group membership service is to track voluntary variations in the membership, as well as involuntary events corresponding to failures or repairs occurring at servers or communication links. All these variations are reported to members through the *installation* of *views*. Installed views consist of a collection of members and an unique view identifier, and correspond to the group's current membership as perceived by the members included in it. The PGMS specification requires that members install only views to which they belong, and only after having agreed upon their composition with other members in the view. Two views installed by two different members must be installed in the same order. Being partitionable, our PGMS admits the co-existence of concurrent views. Unlike a primary-partition group membership service, our system is *live*: it never stops to track changes in group membership, even under failure scenarios where no primary partition exists.

Figure 3.3 illustrates an example scenario where three servers are member of the same group and react to different failure events. Initially, servers p, q and r share the same view v_1 containing all of them. Suddenly, a partitioning splits the network in two partitions, one containing servers p and q, the other containing only server r. The group membership

```
import jgroup.*;

public class Monitor implements Member {

  private final static int MONITOR_GID = 111;
  private GroupManager gm;

  public Monitor(DistributedSystem ds) {
    gm = GMFactory.getGroupManager(ds,this);
    gm.join(null, MONITOR_GID);
  }

  public void vChange(Object obj, View v) {
    System.out.println(v);
  }

}
```

Figure 3.4: The Monitor class

service reacts by installing two concurrent views v_2 and v_3, each of them corresponding to one of the partitions. After a certain period of time, the partitioning disappears and the partitions merge back. The group membership service reacts by creating a new common view v_4. Unfortunately, server p crashes before being able to install this view. Server q and r react to the crash by installing a new view v_5 excluding r.

In order to explain how PGMS can be exploited to develop partition-aware applications, a simple example is given (Figure 3.4): a Monitor server whose only task is to print the current view composition whenever a new view is installed. To further simplify the example, exception handling has been removed. Note that this application does not provide any service to external clients, nor requires extra communication among servers. Its only purpose is to illustrate the basic features of our group membership service.

The programming interface of PGMS is contained in the jgroup package, that is imported at the beginning of Figure 3.4. In order to become a member of a group and to be capable to receive view change notifications, each monitor server must implement the Member interface (Figure 3.5). All interactions between member objects and the group membership service are handled by *group managers*, whose interface is enclosed in Figure 3.6. A member that needs to join a group must first obtain a GroupManager object for the group using the getGroupManager method of the GMFactory class (Figure 3.6). The first argument of getGroupManager is a DistributedSystem object, i.e. an high-level

```
package jgroup;

public interface Member {

  public void vChange(Object obj, View v);

  public void prepare(Object obj);

  /* ... */

}
```

Figure 3.5: The jgroup.Member interface

description of the collection of hosts in which servers are expected to be executed. The second argument is the Member object that will be notified at every view change. In the example, the Monitor constructor uses the distributed system received as argument and this to indicate that view change events have to be notified to the monitor object itself.

After having obtained a group manager, the second step is to invoke the join(Object gdata, int gid) method on it, in order to join the group identified by gid. Argument gdata may contain an application-defined data structure that will be associated with the group (this may be useful when an object is member of more than one group; see Appendix B for details). Group identifiers should be interpreted similarly to the concept of *port number* in protocols like TCP and UDP: a mean for remote entities to collect themselves together under the same communication endpoint. When designing a dependable applications, developers choose a group identifier for it and make all servers join the group with the same group identifier. In the example, each monitor joins the group using MONITOR_GID as identifier. Obviously, group identifiers are used only internally to the service implementation; clients make use of a completely different identification scheme, based on a naming service. This scheme will be introduced in the next section.

After having joined the group, the object will start to receive view change notification from the group manager through its vChange(Object gdata, View v) method inherited from the Member interface. View v contains information about the current membership of the group, while gdata is a reference to the application-defined data structure associated with the group through the join method. The implementation of method vChange has to be provided by application developers, depending on the desired behavior of the application in the case of new failures or repairs. In our example, vChange prints the view composition.

```
package jgroup;

public interface GroupManager {

  public void join(Object gdata, int gid)
    throws GMHaltedException, JgroupException;

  public void leave()
    throws GMHaltedException, JgroupException;

  /* ... */
}

public class GMFactory {

  public static GroupManager getGroupManager
    (DistributedSystem ds, Member member)
      throws JgroupException, java.io.IOException:

  /* ... */
}
```

Figure 3.6: The jgroup.GroupManager interface and the jgroup.GMFactory class

3.5 The Group Method Invocation Service

As noted in Section 3.3, Jgroup differs from existing object models for its uniform communication interface entirely based on the group method invocation paradigm. For the first time, both clients and servers interact with groups by remotely invoking methods on them. In this way, the benefits of object-oriented programming are extended also to internal communication among servers. Although they share the same intercommunication paradigm, we distinguish among *internal* invocations performed by servers and *external* invocations performed by clients. There are several reasons behind this distinction:

- *Accessibility*: in general, methods invoked by servers should not be invocable by clients. In fact, clients should be able to access only the "public" interface of the service, while methods invoked by servers should be considered "private" to the implementation.

- *Transparency*: the aim of Jgroup is to provide an invocation mechanism for clients completely transparent with respect to standard RMI interactions: once obtained a stub for the group, clients invoking a method are blocked until they obtain a single return value. This means that clients are not required to be aware of the fact that the method they are invoking will be executed by a group of servers, instead of a single one. Servers, on the other hand, may have different requirements; for example, a server may need to obtain a result value from each of the servers included in the current view.

- *Efficiency*: Satisfying the same requirements for external and internal method invocations would have required that clients became members of the group. To avoid this, the specification of external invocations is slightly weaker with respect to internal ones. Recognition of differing invocation semantics allows higher costs of full group membership to be limited to servers that are typically far fewer in number than clients.

The distinction between internal methods that can be invoked by servers and external methods that may be invoked by clients is made when designing the application. Internal methods are collected in *internal remote interfaces*, while external methods are collected in *external remote interfaces*. Each server class must implement one external remote interface and one internal remote interface.

An external interface satisfy the following requirements:

- It must at least extend, either directly or indirectly, the interface GmiRemote. This interface extends java.rmi.Remote, and contains no methods. It is used as a marker to identify external interfaces.

- Each method declaration in an external remote interface must satisfy the requirements of standard remote method invocations, i.e. it must include the exception java.rmi.RemoteException (or one of its superclasses) in its *throws* clause, in addition to any application specific exception.

Internal interfaces must extend, directly or indirectly, the interface Member. Method declarations contained in them do not have to satisfy particular requirements. Note that internal and external interfaces must be clearly separated: an internal interface cannot extend GmiRemote, and an external interface cannot extend Member.

After having defined the external and internal interfaces of a service, the next step is to obtain the appropriate proxy classes for invoking external and internal methods on the group itself. Proxies for external method invocations are called *stubs* (as in standard

```
package jgroup;

import jgroup.registry.DependableRegistry;

public interface GMIGroupManager {

  public void join(Object gdata, int gid, String name)
    throws GMHaltedException, JgroupException;

  public void join(Object gdata, int gid, String name, DependableRegistry drs)
    throws GMHaltedException, JgroupException;

  public IID getIdentifier();

}
```

Figure 3.7: The jgroup.GMIGroupManager class.

Java RMI), while the task of serving internal method invocations on a group is delegated to *GMI group managers*, i.e. group managers opportunely extended to manage groups of server objects that communicate through group method invocations. Stubs implement the same external interface for which they act as a proxy; on the contrary, a GMI group manager implements a *multi-response interface* derived from the internal interface, and the GMIGroupManager interface illustrated in Figure 3.7. Multi-response interfaces differs from the corresponding internal interface due to the fact that internal invocations return an array of results, instead of a single one. Stubs, multi-response interfaces and group managers are generated by a tool called `gmic` (GMI compiler), included in the Jgroup distribution. The compiler is invoked with the package-qualified interface names of the external and the internal interface, that must have been previously compiled successfully. It generates and compiles the required classes and interfaces, that are inserted in the package of the internal interface. Being derived from the standard `rmic` compiler included in the distribution of Java RMI, `gmic` supports all command line arguments that can be used with `rmic`.

Servers that need to perform internal method invocations may use the group manager they have obtained from the GMFactory class. This group manager is a GMI group manager and implements the multi-response interface corresponding to the internal interface implemented by the server.

On the contrary, clients that need to interact with a group must obtain a stub from a *dependable registry service* (see Section 3.7 for details). A dependable registry service

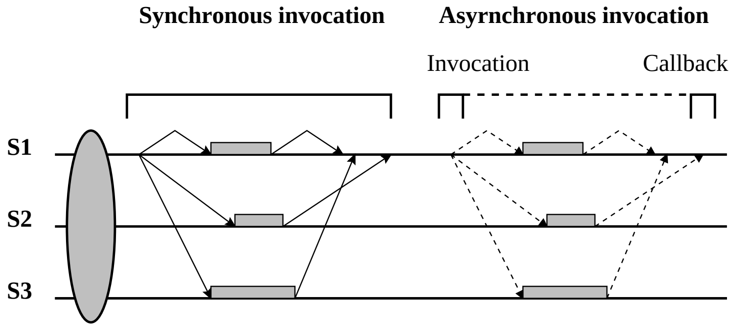

Figure 3.8: Internal method invocations with synchronous and asynchronous semantics

enables servers to register themselves under a group name represented by a string. Clients may look up the registry by name and obtain the desired stub. The act of registering a server under a group name is performed by invoking one of the join methods declared in GMIGroupManager (Figure 3.7), which require the specification of a name for the group. Using the first version of join, the server is bound on the dependable registry service currently running in the distributed systems described by the DistributedSystem object used to obtain a group manager. The second version of join binds the server on the dependable registry service specified by drs.

In the following sections, we illustrate how internal and external invocations work, and how internal invocations have substituted message multicasting as communication paradigm. In particular, we describe how reliability guarantees have been opportunely adapted to cope with method invocations instead of messages. This has requested also the introduction of new terminology. We say that an object (clients or server) *performs* a method invocation when it invokes a method on a group; we say that a server *executes* an invocation when it executes the method associated with it. Method invocations are uniquely identified; in this way, it is possible to establish an association between the act of performing a method invocation and the act of executing it.

3.5.1 Internal Group Method Invocations

Differently from Java remote method invocations defined in the next section, internal method invocations return an array of results instead of a single value. This array contains a value for each of the servers that have executed the method invocation. The multi-request/multi-reply interaction style suggested by internal invocations may significantly

```
package jgroup;

public class MultipleException {

  public Exception[] getExceptions();

  public Object[] getValues();

}
```

Figure 3.9: The jgroup.MultipleException class.

reduce the cost of development of distributed protocols. Consider for example a protocol in which a coordinator wants to obtain some information from every member of a group. In a message-based system, this operation requires a multicast message from the coordinator, and a reply message from each group member. Reply messages have to be sent only to the coordinator, in order to avoid the costs of a multicast message from every member. Moreover, the coordinator must be able to collect replies coming at different times, possibly associated with concurrent request messages. In Jgroup, GMIS takes care of all these low-level details.

If one or more of the servers executing an invocation throws an exception, GMIS throws a MultipleException (see Figure 3.9). Method getExceptions declared in this special exception returns an array containing the exceptions that have been thrown during the execution of the method, while method getReturnValues enables servers to obtain the regular return values. MultipleException is thrown also when one or more of the servers in the current view are not able to execute the method invocation, for example due to a crash or a partitioning. In this case, the exception array returned by getExceptions contains an UnreachableException in correspondence of the unreachable servers.

Internal invocations described so far follow the *synchronous* semantics of standard method invocations: the invoking thread remains blocked until the control is returned to it. Unfortunately, there are programming scenarios where the synchronous nature of this blocking model of interaction may be too costly, as it requires an application-level response from every member of the group. Furthermore, it also requires programmers to consider other issues related to deadlock caused by circular invocations. For this reason, Jgroup provides also the possibility of performing *asynchronous* internal invocations. In order for an invocation to be asynchronous, the invoker must specify a *callback* object, that

```
package jgroup;

public interface Callback {

  public void returnValues(Object[] values);

  public void returnException(MultipleException e);

}
```

Figure 3.10: The jgroup.Callback interface.

will be notified when a reply array is ready to be delivered. The invocation terminates immediately, without returning any value. A callback object must implement the Callback interface (see Figure 3.10), by defining the returnValues and returnException methods, which are invoked to notify the application that either a reply array or a MultipleException are ready to be delivered. If the method returns *void*, the invoker has two possibilities: it can specify a callback object, that in the absence of return values is used to acknowledge the termination of the invocation; or it can specify null, meaning that the invoker is not interested in knowing when the invocation terminates. Examples of synchronous and asynchronous invocations are depicted in Figure 3.8.

As internal invocations return arrays of replies, the GMI group manager that acts as a proxy for the group does not implement the internal remote interface, but the multi-response interface derived from it. As an example, consider the internal interface Foo shown in Figure 3.11. It contains a method getValue returning an integer and a method setValue returning void. The corresponding multi-response interface `FooGM` obtained through `gmic` is shown in the same figure. Note that FooGM contains two method for each of the methods declared in Foo, one for synchronous invocations and one for asynchronous ones. The synchronous version of getValue returns an array of integers, while the synchronous version of setValue returns void as the original one. On the contrary, methods for asynchronous invocations always return void.

After having described the basic notions about how internal methods are declared and invoked, it is opportune to describe what guarantees are provided by our group method invocation service. The complete specification of the properties regarding internal invocations is contained in Section 4.3.3. Here, we provide a brief introduction to the main properties that characterize our specification. First of all, our communication service is

```
public class Foo {

  public int getValue();

  public void setValue(int value);

}

public class FooGM {

  public int[] getValue()
    throws jgroup.MultipleException;

  public int[] getValue(jgroup.Callback cb)
    throws jgroup.MultipleException;

  public void setValue(int value);
    throws jgroup.MultipleException;

  public void setValue(int value, jgroup.Callback cb);
    throws jgroup.MultipleException;

}
```

Figure 3.11: An example of internal class and its corresponding multi-response interface.

live: the execution of a method invocation is guaranteed to terminate with at least one return value, or an application-defined exception. Furthermore, if a correct server executes an internal invocation in a view, all servers included in that view will execute the same invocation, or the server will install a new view. Since installed views represent the current failure scenario as perceived by servers, this guarantees that the method invocation will be executed by every server in the partition of the invoker. Internal invocations satisfy also fundamental integrity requirements such as that a method invocation is executed at most once, and only if some server has previously performed it. Finally, we guarantee that each internal invocation is executed in at most one view; in other words, different servers executing an invocation, execute it in the same view. In this way, result values contained in a reply array are guaranteed to have been computed during the same view.

The execution of internal method invocations by the servers forming a group follow the view synchrony semantics. View synchrony requires that servers surviving from a view to another one execute the same set of method invocations during the first of the two views. In other words, before a new view is installed, all servers included in both the previous and the new view have to agree on the set of method invocations they have

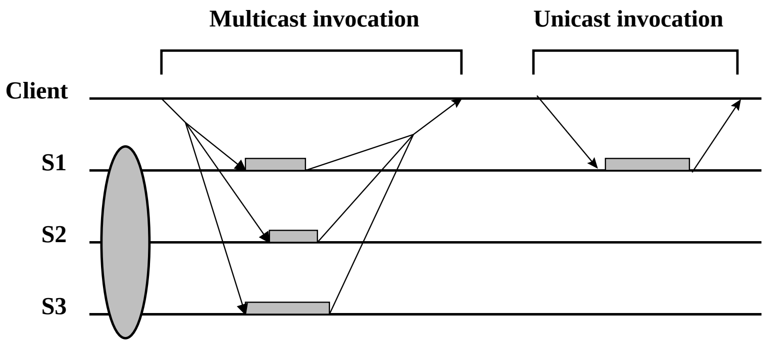

Figure 3.12: External method invocations with multicast and anycast semantics

executed during the previous view. This enables servers to reason about the state of other servers using only local information such the history of installed views and the set of method invocations executed. Clearly, depending on the application semantics, it is possible that servers need to agree on the *sequence* of executed method invocation. For example, a distributed blackboard may be implemented using totally-ordered group method invocations. In Jgroup, ordering semantics may be easily enforced by adding additional layers which consider the set of invocations executed in a view and order them accordingly to the chosen semantics.

3.5.2 External Group Method Invocations

External invocations are completely transparent with respect to standard remote method invocations; so, they return a single value instead of an array. When designing the external remote interface for a service, an application developer must choose between two different invocation semantics:

- the *anycast invocation semantics* guarantees that a method invocation performed by a client on a group of servers will be executed by invoking the same method on at least one of the servers, unless the client is completely partitioned from the object group. The anycast semantics is suitable for methods whose execution can be performed on a single server, without involving the entire group; for example, for read-only methods that interrogate on a replicated database.

- the *multicast invocation semantics* guarantees that a method invocation performed by a client on a group of servers will be executed by invoking the same method on every server contained in the client's partition. The multicast semantics is suitable for methods whose execution must affect every server of a group; for example, for write methods that modify the state of a replicated database.

The choice of the semantics used for a method invocation is done when designing the external remote interface, by including the exception McastRemoteException in the *throws* clause of methods that need to be invoked with multicast semantics. When generating the GMI stub for the interface, `gmic` will analyze the throw clause using reflection and will adapt the stub code opportunely.

As for internal invocations, GMIS guarantees that external invocations are live: if at least one of the servers is not partitioned from the invoking client, the service will eventually return a reply value to the client. Furthermore, external invocations are executed at most once, and only if some client has previously performed it. These properties are common to invocations with both anycast and multicast semantics. External invocations with multicast semantics satisfy also view synchrony: two servers surviving from a view to another one execute the same set of external methods in the period occurring between the installations of those two views.

As the reader can see, specifications of external and internal method invocations present several similarities. Nevertheless, there is an important difference among them: external invocations are not guaranteed to be executed in at most one view. The problem lies on the fact that, whenever a client's stub does not obtain an answer from a server that have been contacted in order to have a method invocation executed, there is the possibility that the server has actually executed the method, but a partitioning has occurred before having the possibility to deliver the reply to the client. Thus, in order to eventually obtain a reply, the stub has to contact another server, and this may cause the method invocation to be executed in concurrent views. We analyze this problem more extensively in Section 5.7, where we prove that the only solution for it is to force clients to become members of a group before performing a method invocation on it. Obviously, a similar solution is too costly, as groups would become very large.

In the absence of this property, application developers have to consider the problems that may be caused by the execution of the same method by members included in different partitions. Consider for example a merging protocol (possibly based on the state merging service described in the next section). When two views merge in a new one, servers contained in them must exchange information about updates derived from external invocations. If the same update has been applied in two different views, the state merging protocol may cause the duplicated application of the same update. In Jgroup, the problem has been avoided by enabling servers to obtain a method invocation identifier for each of the external method invocations they execute. Each identifier is unique; when exchanging information during merging protocols, servers may associate an identifier with each of the updates, and thus discard duplicated updates. The method identifier for an external invocation may be obtained by invoking method getIdentifier on the GMI group manager associated with the group.

As noted before, one of the requirements of our group method invocation service is that external invocations performed by clients are completely transparent with respect to standard Java RMI interactions. Thus, issues like the possibility of having a return value from each of the servers executing an external method, or the possibility of performing asynchronous external invocations on a group have not been considered when designing Jgroup. The choice of including transparency among the requirements of our group method invocation service is motivated by the fact that providing other non-transparent invocation semantics would have complicated enormously the design of the external group communication service included in Jgroup, and consequently its utilization from application developers. In fact, client and server designers would have had to confront with three different semantic choices (anycast vs multicast, synchronous vs asynchronous, one-response vs multi-response), leading to eight different invocation semantics. For this reason, we prefer to postpone the design and the implementation of these semantics in future extension of the Jgroup distributed object model.

3.6 The State Merging Service

The fact that our partitionable group membership service allows the co-existence of multiple, concurrent views must be taken into particular consideration when developing dependable applications. During a partitioning, the state of servers belonging to distinct partitions may become inconsistent, as each partition may carry on the computation independently from the others. When the conditions leading to the partitioning are repaired and a new view is installed, an application-specific merging protocol may be needed.

Servers participating in the protocol try to define a new global state, which reconciles, whenever possible, the divergence that may have taken place. As an example, consider the implementation of a shared blackboard for supporting cooperative work sessions among groups of users. During a partitioning, the contents of blackboards maintained by servers belonging to different partitions may evolve differently. When partitions merge and a new view is installed, servers that belonged to different partitions exchange the changes in the blackboard observed since the last time they have been together in the same view.

State merging protocols are based on the exchange of information between servers currently included in a common view, but that have been partitioned in the past. Determining what information need to be exchanged, and how it should be used to obtain a new global state are problems that are completely application-dependent, and thus are left to application programmers. If we consider the problem of distributing this information, on the contrary, the situation is different. Based on our experience in building applications in partitionable systems [10, 57], we have identified a general scheme of organizing state merging protocols for partition-aware applications. In this scheme, when two or more partitions merge in a new one, a coordinator is elected among the servers in each of the merging partitions; each coordinator acts on the behalf of its partition and is responsible to select the information needed to update servers not included in its partition. The state merging service takes care of diffusing this information among the servers composing the group. This one-round diffusion scheme has proven to be extremely useful when developing partition-aware applications, as testified by the examples contained in Chapter 6.

The aim of the *state merging service* (SMS) provided with Jgroup is to support the development of state merging protocols based on this diffusion scheme. SMS is similar to the state transfer service described in Section 3.2, as it drives the state merging protocol by calling back to the servers for “getting” and “merging” information about their state. The process of selecting a coordinator and diffusing the information are under the responsibility of the state merging service as well.

In order to use SMS when designing state merging protocols for partition-aware applications, servers must satisfy the following requirements:

- Each server must be able to act as a coordinator; in other words, every server has to maintain the entire shared state and be ready to respond with information about its state when requested by SMS.

- A server is required to be capable to receive information from servers belonging to different partitions and merge it with its current state. This assumption restricts the applicability of SMS; for example, applications implementing replicated servers with high-consistency requirements may not be capable to reconcile inconsistent updates on the same record. Note, however, that this is a limitation of the partition-aware approach, and not of our state merging service.

- To guarantee efficiency, each server should respond with a minimal quantity of information when SMS elects it as coordinator. A technique that may be applied to decrease the information exchanged is *semantic compression*: for example, if a portion of the shared state has been updated several times during the partitioning, in some cases it is possible to transmit only the last value of this portion. Returning to the shared blackboard example, if a user draws a box and moves it around the screen, when performing a state merging protocol only the last position is relevant. Another technique could be to maintain information about the updates that have been applied by other servers. Clearly, in this case there is a trade-off between the amount of information that need to be stored and the amount of information exchanged.

In our experience, it is not difficult to design state merging protocol satisfying these requirements for a large class of partition-aware applications. Note, however, that the distribution scheme supported by SMS is not the only possible; for example, other techniques [6] are more indicated for developing complex state merging protocols involving several communication rounds.

As said above, a coordinator is required to act on the behalf of other servers. In order to select a coordinator, SMS requires information about "who can act on the behalf of whom". This notion will be formalized in Section 4.4; here, we limit ourselves to an informal definition. We say that a server p is *up-to-date with respect to* a server q if all information known by q are known also by p; alternatively, we say that q belongs to the *update set* of p if p is up-to-date with respect to q. A server p may act as a coordinator on the behalf of a server q if p is up-to-date with respect to q. Initially, a server is up-to-date only with respect to itself. After having received information from other servers through the execution of its "merging" callback method, it is certain to be up-to-date with respect

```
package jgroup;

public interface MergingMember extends Member {

  public Object vChange(Object gdata, View view, MemberID[] dests);

  public void mergeState(Object gdata, Object state, MemberID[] sources);

}
```

Figure 3.13: The jgroup.MergingMember interface.

to these servers. On the other hand, a server becomes not up-to-date with respect to other servers due to the installation of a new view excluding them. Consider for example a server p installing a view v not containing a server q; since the state of q may be evolving inconsistently with respect to p, SMS declares p as not up-to-date with respect to q.

The programming interface of the state merging service provided with Jgroup is represented by the MergingMember interface illustrated in Figure 3.13. Members needing to exploit SMS must implement MergingMember, which extends the Member interface defined in Section 3.4. Two new methods are declared: a new version of the vChange method declared in Member, and a new method called mergeState. These methods are the callbacks invoked by SMS in order to implement the state merging protocol. As before, the invocation of a vChange method notifies the server that a new view has been installed. The invocation of the vChange version declared in MergingMember has the additional meaning that the server has been elected coordinator and is required to return an object containing the information needed to update the servers included in the dests array. Once obtained the information, SMS transmit it to servers in dests, where it invokes mergeState. Argument state contains the update information, while array sources contains the servers that already know this information. In both methods, argument gdata represents the user-defined data structure useful for servers belonging to multiple groups.

The complete specification of SMS is contained in Section 4.4; here, we briefly outline its basic properties. The main requirement satisfied by SMS is *termination*: if there is a time after which two servers install only views including each other, then eventually each of them will become up-to-date with respect to the other (directly or indirectly, meaning that a different server may be elected coordinator and provide information on behalf of one of the two servers). Another important property is *agreement*: servers surviving from

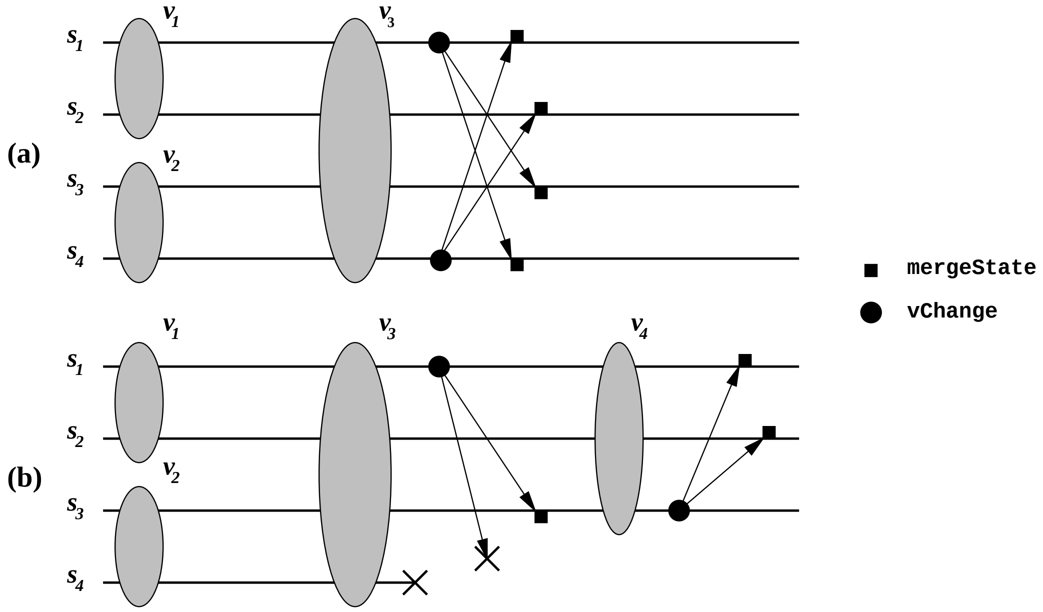

Figure 3.14: The functioning of the state merging service.

a view to the next one are guaranteed to execute the same set of mergeState invocations in the period occurring between the installations of the two views. This property is similar to view synchrony, and like view synchrony may be used to maintain information about the updates applied by other servers. Finally, an integrity property is defined, requiring that the special version of vChange and method mergeState are not invoked without reason (if servers are already up-to-date).

Meaningful examples of application programmed using SMS may be found in Chapter 6; for example, the dependable registry service described in the next section is based on it. For this reason, instead of presenting some trivial example, it is preferable to discuss some scenarios that illustrates the functioning of SMS. In Figure 3.14(a), two partitions (identified by view v_1 and v_2 in the figure) merge and a new view v_3 is installed. Assume that servers in each of v_1 and v_2 are up-to-date with respect to each other; in other words, previous merging protocols executed among them have been completed successfully. SMS elects s_1 coordinator of view v_1, and s_4 coordinator of view v_2, and calls them back to obtain information about the current state maintained in v_1 and v_2, respectively. The information obtained from view v_1 is disseminated to servers included in v_2, and vice versa. Once successfully merged the information coming from other partitions, each server is declared to be up-to-date with respect to servers belonging to the other partition.

A more complex scenario is illustrated in Figure 3.14(b). Initially, the scenario is similar to the previous one. Two partitions merge in a single one and a new view v_3 is installed. SMS elects s_1 coordinator for v_1, and s_4 coordinator for v_3, and calls them back to obtain information about updates. The updates obtained from s_1 are successfully merged to the state of s_3 and s_4, that are declared to be up-to-date with respect to s_1 and s_2. On the contrary, s_4 crashes before having the possibility of responding to the callback. The partitionable group membership service reacts by installing a new view v_4. Note that servers in v_1 are not up-to-date with respect to s_3. In order to solve this divergence, SMS starts a new state merging protocol. Here, only s_3 is selected as coordinator; the state merging protocol terminates when each of the servers included in v_1 is updated with the information coming from s_4.

3.7 The Dependable Registry Service

When introducing the object group paradigm, we have said that method invocations performed by clients on a group cannot be distinguished from standard remote method invocations on non-replicated servers. In Java RMI, this implies that before invoking a method, a client must obtain a stub for the group implementing the method itself. So far, we have not specified how these stubs are obtained in Jgroup. As illustrated in Chapter 2, Java RMI solves this problem by providing a registry service, a repository facility used by servers to advertise their availability to provide certain services, and by clients to retrieve stubs for remote objects. The `bind` method enables the registration of servers under a certain name, while the `lookup` method is used to retrieve servers by name. Unfortunately, the registry implementation included in the current JDK is not suitable for a distributed object model based on object groups. First of all, the registry introduces a single point of failure; if a registry fail, all objects bound in it are not accessible any more. Furthermore, the registry service has not been designed to deal with groups of server objects. In fact, a registry running in a certain host can register only objects executing on the same host; this precludes the possibility of registering groups of servers concurrently running on several different hosts.

In order to solve this problem, the Jgroup toolkit includes the *dependable registry service* (DRS), a repository facility used by long-lived groups of servers to advertise their

```
package jgroup.registry;

public interface DependableRegistry
  extends jgroup.rmi.GmiRemote {

  public Remote lookup(String name)
    throws RemoteException, NotBoundException, AccessException;

  public BID bind(String name, Remote obj)
    throws RemoteException, AccessException, McastRemoteException;

  public void unbind(BID bid)
    throws RemoteException, NotBoundException, AccessException,
            McastRemoteException;

  public String[] list()
    throws RemoteException, AccessException;

}

public final class LocateDependableRegistry {

  public static JgroupRegistry getRegistry(DistributedSystem ds)
    throws RemoteException;

  public static JgroupRegistry getRegistry(DistributedSystem ds,
                               RMIClientSocketFactory csf)
    throws RemoteException;

  /* ... */

}
```

Figure 3.15: The DependableRegistry interface and the LocateDependableRegistry class included in the jgroup.registry package.

availability to provide certain services, and by clients to retrieve stubs for these groups needed to perform method invocations on them. Being constituted by a group of servers that actively maintain a replicated database of bindings between group names and servers, the DRS does not present any single point of failure. It is interesting to note that the DRS implementation is based on the services provided by Jgroup itself; in other words, the servers composing a dependable registry service are collected in a group using Jgroup, and clients access the registry service through external group method invocations. The implementation of the DRS constitutes a good example of programming based on Jgroup itself; for this reason, it is presented in Chapter 6.

Figure 3.15 contains DependableRegistry and LocateDependableRegistry, respectively the external remote interface of DRS and the bootstrap mechanism to obtain references to the dependable registry service running in the system. Both classes are included in the jgroup.registry package. The DependableRegistry interface contains four methods. The bind(String name, Remote remote) method is used to add server remote to the group identified by name; it returns a BID object, i.e. an identifier of the bind operation needed to subsequentially remove the server object from the group. The unbind(BID bid) method is used to remove a binding identified by bid. Note that bind and unbind includes McastRemoteException in their *throws* clause; this means that their invocation follow the multicast semantics. The lookup(String name) method is used to retrieve a stub for an object group by name. Finally, the list() method is used to obtain a list of the names of the object groups registered in the registry. As a dependable registry is a service based on Jgroup, clients need a stub to access it. Clearly, they cannot use the DRS to obtain it; so, a bootstrap mechanism is provided, implemented by the getRegistry methods of the LocateJgroupRegistry class.

A dependable registry has several advantages over its non-replicated counterpart. First, it offers a high-available registry facility. Second, clients no longer need to be aware of the registry location as in the JDK implementation. Distributed systems can be designed by including a certain number of registry replicas running on different hosts and possibly on distinct portions of the communication network. Clients access these replicas through standard RMI interactions as if they were a single registry, and are guaranteed that their invocations will successfully terminate, provided that at least one operational replica is running in their partitions.

During a partitioning, a dependable registry presents a partitioned behavior reflecting the current failure scenario. A bind operation executed inside a partition will not affect the replicas not contained in that partition, while a lookup operation will not be able to retrieve bindings registered outside the current partition after the beginning of the partitioning. Nevertheless, replicas contained in a partition consistently maintain the same set of bindings and act as a single entity; moreover, the disappearance of the partitioning causes the execution of a state merging protocol in order to re-establish a consistent set of bindings among the replicas that belonged to different partitions. It is important to note that this behavior is perfectly suitable for a partitionable distributed system, since clients asking for remote services are interested only in servers running in their partitions.

Chapter 4

Formal Specification

In this chapter, the specification of the main components of Jgroup is presented. The provision of a formal specification for the Jgroup system is important, as developers may reason about the correctness of their applications using only the specification, without knowing anything about the implementation. Furthermore, this chapter formalizes and clarifies important details of the Jgroup system that have been omitted in the informal presentation of Jgroup provided in the previous chapter. Our specification is based on previous work on partitionable group communication [76, 61, 23], and in particular extends the specification of Relacs [9]. Nevertheless, it presents also important differences with respect to classical group communication, since interactions are based on group method invocations rather than on multicast messages.

The chapter begins with a formal model for the underlying system in which Jgroup will be deployed (Section 4.1). Then, we discuss the specification of the main components of Jgroup, i.e. the partitionable group membership service, the group method invocation service and the state merging service.

4.1 System Model

The system is composed of a finite set of *objects* $\mathcal{O}$, which communicate via a datagram service. Objects are identified through unique names that they keep throughout their life. Every pair of objects is connected by an *unreliable* communication channel, i.e. a channel may drop messages due to failures. Objects may crash. Crashed objects do not recover.

The system is asynchronous, in the sense that it is impossible to bound both computation and communication delays. Practical distributed systems often have to be considered as being asynchronous due to transient failures, unknown scheduling strategies and variable loads on the computing and communication resources. To simplify the presentation, however, we assume the existence of a discrete global clock. This simplification is not in conflict with the asynchrony assumption since objects are never allowed to access this clock. We denote the set of all clock values by $\mathcal{T}$; for simplicity, we assume that $\mathcal{T}$ corresponds to the set of natural numbers.

4.1.1 Computation Model

The execution of a distributed algorithm results in each object performing a sequence of *events*, possibly null, chosen from a set $\mathcal{E}$. In addition to internal events not relevant for this work, $\mathcal{E}$ includes method invocation and group communication events. The *global history* of an execution is a function σ from $\mathcal{O} \times \mathcal{T}$ to $\mathcal{E} \cup \{\epsilon\}$ where ϵ denotes the null event. If object p executes an event $e \in \mathcal{E}$ at time t, then $\sigma(p,t) = e$. Otherwise, $\sigma(p,t) = \epsilon$ indicating that p performs no event at time t. Given some interval $\mathcal{I}$ of $\mathcal{T}$, we write $e \in \sigma(p,\mathcal{I})$ if p executes event e sometime during interval $\mathcal{I}$ of global history σ (i.e, $\exists t \in \mathcal{I} : \sigma(p,t) = e$). Objects may fail by *crashing* whereby they stop executing events. For simplicity, we do not consider object recovery after a crash. The evolution of object failures during an execution is captured through the *crash pattern* function C from $\mathcal{T}$ to $2^{\mathcal{O}}$ where $C(t)$ denotes the set of objects that have crashed by time t. Since crashed objects do not recover, we have $C(t) \subseteq C(t+1)$. With $\mathit{correct}(C) = \{p \mid \forall t : p \notin C(t)\}$ we denote those objects that never crash, and thus, are correct in C.

We distinguish among two types of events: those corresponding to primitives requested by applications to join a group or to start a new application, and those associated with upcalls invoked by Jgroup on applications to notify a view change or a communication event. Upcalls correspond to executions of methods included in the member interface. These methods may perform complex sequences of actions. In the following, we need to identify the beginning and the end of an upcall execution; for this reason, we associate a pair of events with each upcall: $\mathit{event}(\mathit{args})$ is the event executed at the beginning of the upcall, while $\overline{\mathit{event}}(\mathit{args}, r)$ is the event associated with its end. The value r is the result returned by the upcall (if any).

Clearly, there are few properties that must be satisfied by the pair of events associated with an upcall. First of all, we assume that before executing an event $\overline{\mathit{event}}(\mathit{args}, r)$, a server must have executed the corresponding event $event(args)$, and each server executes events $event(args)$ and $\overline{event}(args, r)$ for a given pair $(event, args)$ at most once. Formally,

$$\sigma(p,t) = \overline{event}(args, r) \;\Rightarrow\; event(args) \in \sigma(p, [0, t[)$$
$$\sigma(p,t) = event(args) \;\Rightarrow\; event(args) \notin \sigma(p, \mathcal{T} - \{t\})$$
$$\sigma(p,t) = \overline{event}(args, r) \;\Rightarrow\; \overline{event}(args, r') \notin \sigma(p, \mathcal{T} - \{t\}).$$

Another important condition is that upcalls eventually terminate, i.e. that $event(args)$ events executed by correct servers are always followed by corresponding $\overline{event}(args, r)$ events. Formally,

$$p \in correct(C) \wedge \sigma(p,t) = event(args) \;\Rightarrow\; \exists r : \overline{event}(args, r) \in \sigma(p,]t, \infty[)$$

It is important to note that this condition cannot be enforced by Jgroup, as it depends on the code associated with the upcalls (written by Jgroup users). In other words, application programmers are required to write upcall methods that eventually terminate. In the absence of this condition, the liveness properties of the Jgroup specification cannot be guaranteed.

4.1.2 Communication Model

The datagram service provides primitives for transmitting unicast messages. If object p sends message m to object q at time t, we say that p executes the event $send(m, q)$ at time t (denoted $\sigma(p,t) = send(m, q)$). If object q receives messages m sent by object p at time t, we say that q executes the event $recv(m, p)$ at time t (denoted $\sigma(q,t) = recv(m, p)$). To simplify the specification of the datagram service, we assume that each message is uniquely identified. In other words, two messages are different even when they are sent by the same object and have the same "contents".

Communication is unreliable (as described below) and sequencing among multiple messages sent to the same destination need not be preserved (i.e., channels are not FIFO). Without loss of generality, we define two requirements for the datagram service:

- *Validity*: a message is received only if it has been previously sent. Formally,

$$\sigma(q,t) = recv(m, p) \;\Rightarrow\; send(m, q) \in \sigma(p, [0, t[).$$

- *No-duplication*: each object can receive a message at most once. Formally,

$$\sigma(p, t_1) = recv(m_1, p) \wedge \sigma(p, t_2) = recv(m_2, q) \wedge t_1 \neq t_j \;\Rightarrow\; m_1 \neq m_2.$$

Note that this communication model is extremely faithful to practical distributed systems built on top of typical unreliable datagram transport services such as IP and UDP.

A variety of events, including link crashes, buffer overflows, incorrect or inconsistent routing tables, may disable communication between objects. We refer to them generically as *communication failures*. Unlike object crashes, which are permanent, communication failures may be temporary due to subsequent repairs. The evolution of communication failures and repairs during an execution is captured through the *unreachability pattern* function U from $\mathcal{O} \times \mathcal{T}$ to $2^{\mathcal{O}}$ where $U(p, t)$ denotes the set of objects with which p cannot communicate at time t. If $q \in U(p, t)$, we say that object q is *unreachable* from p at time t, and write $p \not\rightsquigarrow_t q$ as a shorthand; otherwise we say that object q is *reachable* from p at time t, and write $p \rightsquigarrow_t q$. As noted above, communication failures are not necessarily permanent but may appear and disappear dynamically. This is reflected by the fact that the sets $U(p, t)$ and $U(q, t+1)$ may differ arbitrarily.

Note that the unreachability pattern is an abstract characterization of the communication state of a system, just as the crash pattern is an abstract characterization of its computational state. Only an omnipotent external observer can construct the unreachability and crash patterns that occur during an execution and neither can be inferred from within an asynchronous system. Nevertheless, they are useful in stating desired properties for a group communication service. Any implementation of the specified service in an asynchronous system will have to be based on approximations of unreachability and crashes provided by *failure detectors* [19] as we discuss in the next section.

Reachable/unreachable are attributes of individual communication channels (identified as ordered object pairs), just as correct/crashed are attributes of individual objects. In the rest of the thesis, we also refer to communication failure scenarios called *partitionings* that involve multiple sets of objects. A partitioning disables communication among different *partitions*, each containing a set of objects. Objects within a given partition can communicate among themselves, but cannot communicate with objects outside the partition. When communication between several partitions is re-established, we say that they *merge*.

Object and communication failures that occur during an execution are not totally independent, but must satisfy certain constraints that are captured through the notion of a *failure history*:

Definition 4.1 (Failure History) *A failure history F is a pair (C, U), where C is a crash pattern and U is an unreachability pattern, such that (i) an object that has crashed by time t is unreachable from every other object at time t, and (ii) an object that has not crashed by time t is reachable from itself at time t. Formally,*[1]

$$\begin{array}{lrcl} (i) & p \in C(t) & \Rightarrow & q \not\rightsquigarrow_t p; \\ (ii) & p \notin C(t) & \Rightarrow & p \rightsquigarrow_t p. \end{array}$$

By definition, the unreachability pattern subsumes the crash pattern in every failure history. We nevertheless choose to model crash and unreachability patterns separately so that specifications can be made in terms of properties that need to hold for *correct* objects only.

Finally, we need to relate crash and unreachability patterns to the events of the execution itself. In other words, we need to formalize notions such as "crashed objects halt prematurely" and "unreachable objects cannot communicate directly". We do this by requiring that the global and failure histories of the same execution conform to constraints defining a *run*.

Definition 4.2 (Run) *A run R is a pair (σ, F), where σ is a global history and $F = (C, U)$ is the corresponding failure history, such that (i) a crashed object stops executing events, and (ii) a message that is sent will be received if and only if its destination is reachable from the sender at the time of sending. Formally,*

$$\begin{array}{lrcl} (i) & p \in C(t) & \Rightarrow & \forall t' \geq t : \sigma(p, t') = \epsilon; \\ (ii) & \sigma(p, t) = send(m, q) & \Rightarrow & (\ recv(m, p) \in \sigma(q, \mathcal{T}) \Leftrightarrow p \rightsquigarrow_t q\ ). \end{array}$$

Note that by Definition 4.1(ii), the reachable relation for correct objects is *perpetually* reflexive — a correct object is always reachable from itself. Transitivity of reachability, on the other hand, need not hold in general. We make this choice so as to render our

[1] In these formulas and all others that follow, free variables are assumed to be universally quantified over their respective domains (events, time, messages, views, etc.), which can be inferred from context.

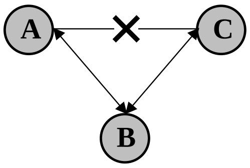

Figure 4.1: An example of scenario where reachability is not transitive.

model realistic by admitting scenarios that are common in wide-area networks, including the Internet, where a site B may be reachable from site A, and site C reachable from B, at a time when C is unreachable from A directly (Figure 4.1). Yet the three sites A, B and C should be considered as belonging to the same partition since they *can* communicate with each other (perhaps indirectly) using communication services more sophisticated than the send/receive primitives offered by the network. As we shall see in Section 5.3, such services can indeed be built in our system model so that two objects will be able to communicate with each other whenever it is possible. And our notion of a partition as the set of objects that can mutually communicate will be based on these services.

We do not assume perpetual symmetry for the reachable relation. In other words, at a given time, it is possible that some object p be reachable from object q but not vice versa. This is again motivated by observed behavior in real wide-area networks. Yet, to make the model tractable, we require a form of *eventual* symmetry as stated below:

Property 4.1 (Eventual Symmetry) *If, after some initial period, object q becomes and remains reachable (unreachable) from p, then eventually p will become and remain reachable (unreachable) from q as well. Formally,*

$$(i) \quad \exists t_0, \forall t \geq t_0 : p \leadsto_t q \ \Rightarrow\ \exists t_1, \forall t \geq t_1 : q \leadsto_t p;$$

$$(ii) \quad \exists t_0, \forall t \geq t_0 : p \not\leadsto_t q \ \Rightarrow\ \exists t_1, \forall t \geq t_1 : q \not\leadsto_t p.$$

This is a reasonable behavior to expect of practical asynchronous distributed systems. Typically, communication channels are bidirectional and rely on the same physical and logical resources in both directions. As a result, the ability or inability to communicate in one direction usually implies that a similar property will eventually be observed also in the other direction.

To conclude the system model, we impose a fairness condition on the communication network so as to exclude degenerate scenarios where two objects are unable to communicate despite the fact that they become reachable infinitely often. In other words, the communication system cannot behave maliciously such that two objects that are normally reachable become unreachable precisely at those times when they attempt to communicate.

Property 4.2 (Fair Channels) *Let p and q be two objects that are not permanently unreachable from each other. If p sends an unbounded number of messages to q, then q will receive an unbounded number of these messages. Formally,*

$$(\forall t, \exists t_1 \geq t : p \rightsquigarrow_{t_1} q) \wedge (\forall t, \exists t_2 \geq t : \sigma(p, t_2) = send(m, p)) \;\Rightarrow$$
$$(\forall t, \exists t_3 \geq t : \sigma(q, t_3) = recv(m', q) \wedge send(m', q) \in \sigma(p, \mathcal{T})).$$

4.1.3 Failure Detector Model

In this Section, we formalize the stability conditions that are necessary for implementing the Jgroup specification. We do so indirectly by stating a set of abstract properties that need to hold for failure detectors that have been suitably extended to partitionable systems. Similar failure detector definitions extended for partitionable systems have appeared in other contexts [50, 22]. The failure detector abstraction originally proposed by Chandra and Toueg [19] is for systems with perfectly-reliable communication. In partitionable systems, specification of failure detector properties has to be based on reachability between pairs of objects rather than individual objects being correct or crashed. For example, it will be acceptable (and desirable) for the failure detector of p to suspect q that happens to be correct but is unreachable from p.

Informally, a failure detector is a distributed oracle that tries to estimate the unreachability pattern U that occurs in an execution. Each object has access to a local module of the failure detector that monitors a subset of the objects and outputs those that it currently suspects as being unreachable from itself. A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π} that describes the outputs of the local modules at each object. If $q \in H(p,t)$, we say that *p suspects q at time t in H*. Formally, a *failure detector* $\mathcal{D}$ is a function that associates with each failure history $F = (C, U)$ a set $\mathcal{D}(F)$ denoting failure detector histories that could occur in executions with failure history F.

In asynchronous systems, failure detectors are inherently unreliable in that the information they provide may be incorrect. Despite this limitation, failure detectors satisfying certain *completeness* and *accuracy* properties have proven to be useful abstractions for solving practical problems in such systems [19]. Informally, completeness and accuracy state, respectively, the conditions under which a object should and should not be suspected for $H(p,t)$ to be a meaningful estimate of $U(p,t)$. We consider the following adaptations of completeness and accuracy to partitionable systems, maintaining the same names used by Chandra and Toueg for compatibility reasons [19]:

FD1 (Strong Completeness) *If some object q remains unreachable from correct object p, then eventually p will always suspect q. Formally, given a failure history $F = (C,U)$, a failure detector $\mathcal{D}$ satisfies Strong Completeness if all failure detector histories $H \in \mathcal{D}(F)$ are such that:*

$$\exists t_0, \forall t \geq t_0 : p \in \mathit{correct}(C) \wedge p \not\rightsquigarrow_t q \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : q \in H(p,t).$$

FD2 (Eventual Strong Accuracy) *If some object q remains reachable from correct object p, then eventually p will no longer suspect q. Formally, given a failure history $F = (C,U)$, a failure detector $\mathcal{D}$ satisfies Eventual Strong Accuracy if all failure detector histories $H \in \mathcal{D}(F)$ are such that:*

$$\exists t_0, \forall t \geq t_0 : p \in \mathit{correct}(C) \wedge p \rightsquigarrow_t q \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : q \notin H(p,t).$$

Borrowing from Chandra and Toueg [19], failure detectors satisfying Strong Completeness and Eventual Strong Accuracy are called *eventually perfect*, and their class denoted $\diamond\tilde{\mathcal{P}}$. In addition to the properties stated above, we can also formulate their weak and perpetual counterparts, thus generating a hierarchy of failure detector classes similar to those of Chandra and Toueg [19]. Informally, *weak* completeness and accuracy require the corresponding property to hold only for *some* pair of objects (rather than all pairs), while their *perpetual* versions require the corresponding property to hold from the very beginning (rather than eventually).

While a detailed discussion of failure detector hierarchy for partitionable systems and reductions between them is beyond the scope of this thesis, we make a few brief observations. In absence of partitionings, failure detector classes with the weak version of Completeness happen to be equivalent to those with the strong version. [2] In such systems, it suffices for *one* correct object to suspect a crashed object since it can (reliably) communicate this information to *all* other correct objects. In partitionable systems, this is

[2]These are the $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$ and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$ results of Chandra and Toueg [19].

not possible and failure detector classes with weak completeness are strictly weaker than those with strong completeness.

In principle, it is impossible to implement a failure detector $\mathcal{D} \in \diamond\tilde{\mathcal{P}}$ in partitionable asynchronous systems, just as it is impossible to implement a failure detector belonging to any of the classes $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ in asynchronous systems with perfectly-reliable communication [19]. In practice, however, asynchronous systems are expected to exhibit reasonable behavior and failure detectors for $\diamond\tilde{\mathcal{P}}$ can indeed be implemented. For example, consider the following algorithm, which is similar to that of Chandra and Toueg [19], but is based on *round-trip* rather than *one-way* message time-outs. Each object p periodically sends a p-ping message to every other object in Π. When a object q receives a p-ping, it sends back to p a q-ack message. If object p does not receive a q-ack within $\Delta_p(q)$ local time units, p adds q to its list of suspects. If p receives a q-ack message from some object q that it already suspects, p removes q from the suspect list and increments its time-out period $\Delta_p(q)$ for the channel (p, q).

Note that since objects send *ack* messages only in response to *ping* messages, a object p will continually time-out on every other object q that is unreachable from it. Thus, the above algorithm trivially satisfies the Strong Completeness property of $\diamond\tilde{\mathcal{P}}$ in partitionable asynchronous systems. On the other hand, in an asynchronous system, it is possible for some object p to observe an unbounded number of premature time-outs for some other object q even though q remains reachable from p. In this case, p would repeatedly add and remove q from its list of suspects, thus violating the Eventual Strong Accuracy property of $\diamond\tilde{\mathcal{P}}$. In many practical systems, increasing the time-out period for each communication channel after each mistake will ensure that eventually there are no premature time-outs on any of the communication channels, thus ensuring Eventual Strong Accuracy.

The only other scenario in which the algorithm could fail to achieve Eventual Strong Accuracy occurs when object q is reachable from object p and continues to receive p-ping messages but its q-ack messages sent to p are systematically lost. In a system satisfying Eventual Symmetry, this scenario cannot last forever and eventually p will start receiving q-ack messages, causing it to permanently remove q from its suspect list and thus satisfy Eventual Strong Accuracy.

Given that perfectly reliable failure detectors are impossible to implement in asynchronous systems, it is reasonable to ask: what are the consequences of mistakenly suspecting a object that is actually reachable? As described in Chapter 5, our use of failure detectors in solving PGMS is such that incorrect suspicions may cause installation of views smaller than what are actually feasible. In other words, they may compromise View Accuracy but cannot invalidate any of the other properties. As a consequence, objects that are either very slow or have very slow communication links may be temporarily excluded from the current view of other objects to be merged back in when their delays become smaller. This type of "view splitting" is reasonable since including such objects in views would only force the entire computation to slow down to their pace. Obviously, the notion of "slow" is completely application dependent and can only be established on a per-group basis.

4.2 The Partitionable Group Membership Service

A *group* is a collection of server objects that cooperate towards some common goal. Every group is uniquely identified by a *group name* g. The group composition may vary dynamically as new servers are added and existing ones removed. Servers willing to participate have to become *member* of a group by *joining* it through primitive join. Later on, a member may decide to terminate its collaboration by *leaving* the group through primitive leave. A server p joining or leaving a group g results in events $join(g)$ and $leave(g)$, respectively. The membership of g at time t is denoted by the set $memb(g,t)$ and comprises those servers that have joined the group but that have not crashed or left the group by time t. Formally,

$$memb(g,t) = \{p \mid \exists t' < t : \sigma(p,t') = join(g) \wedge leave(g) \notin \sigma(p,]t',t]) \wedge p \notin C(t)\}.$$

For sake of brevity, in this work we assume that there could be at most one group in the system; this simplifies the specification, as interactions among views installed for different groups need not to be considered; furthermore, it permits to omit to indicate the group name on each event in the global history. In the following, with the term member we mean a member of the only group of the system.

Given the possibility of failures, each member may have a different perception of the current membership of a group. The task of a partitionable group membership service is to track voluntary variations in the membership, as well as involuntary events corresponding to failures or repairs occurring at servers or communication links. All these variations are reported to members through the *installation* of *views*. The PGMS installs a view v at a server by invoking upcall $\mathsf{vchg}(v)$ on it. In the global history, the execution of this upcall corresponds to the pair of events $vchg(v)$ and $\overline{vchg}(v,r)$. Due to asynchrony and the possibility of partitions, a view for a group g can only be an approximation of the current membership. Views are labeled in order to be globally unique. Given a view v, we write $v.comp$ to denote its composition as a set of object names. The *current view* of object p at time t is v, denoted $view(p,t) = v$, if v is the last view to have been installed at p before time t. Events are said to occur *in the view* that is current. View w is called *immediate successor of* v *at* p, denoted $v \prec_p w$, if p installs w in view v. View w is called *immediate successor* of v, denoted $v \prec w$, if there exists some object p such that $v \prec_p w$. The *successor* relation $\prec^*$ denotes the transitive closure of $\prec$. Two views that are not related through $\prec^*$ are called *concurrent*. Given two immediate successor views $v \prec w$, we say that a object *survives* the view change if it belongs to both v and w.

4.2.1 The PGMS Properties

Without lack of generality, the specification of our PGMS is stated for a single group. This means that the properties contained in this section do not explicitly refer to a particular group. The only exception is given by the following property, which requires that PGMS tracks changes in the membership of the group due to *join* and *leave* operations performed by servers:

GM1 (Group Consistency)

(i) If a server permanently leaves the group, then eventually it will be excluded from all views for the group installed by other servers. (ii) A server that has never joined the group cannot be included in any view installed for the group. Formally,

$$\begin{aligned} (i) \quad & \exists t_0, \forall t \geq t_0 : p \notin memb(g,t) \;\Rightarrow \\ & \exists t_1, \forall t \geq t_1 : q \notin memb(g,t) \vee p \notin view(q,t).comp \\ (ii) \quad & \sigma(p,t_0) = vchg(v) \;\Rightarrow\; \exists t < t_0 : p \in memb(g,t). \end{aligned}$$

The composition of installed views cannot be arbitrary but should reflect reality through the unreachability pattern that occurs during an execution. In other words, members should be aware of other members with which they can and cannot communicate directly in order to adapt their behaviors consistently. Informally, each member should install views that include all members reachable from it and exclude those that are unreachable from it. Requiring that the current view of a object perpetually reflect the actual unreachability pattern would be impossible to achieve in an asynchronous system. Thus, we state the requirement as two eventual properties that must hold in stable conditions where reachability and unreachability relations are persistent.

GM2 (View Accuracy)
If there is a time after which two correct members are permanently and mutually reachable, then eventually all views installed by one will contain the other. Formally,

$$\exists t_0,\ \forall t \geq t_0 : p \in \mathit{correct}(C) \wedge p \rightsquigarrow_t q \ \Rightarrow\ \exists t_1, \forall t \geq t_1 : q \in \mathit{view}(p,t).\mathit{comp}.$$

GM3 (View Completeness)
If there is a time after which two correct members are permanently and mutually unreachable (directly or indirectly), then eventually all views installed by one will exclude the other. Formally,

$$\exists t_0, \forall t \geq t_0, \forall q \in \Theta, \forall p \notin \Theta : p \not\rightsquigarrow_t q \ \Rightarrow$$
$$\exists t_1, \forall t \geq t_1, \forall r \in \mathit{correct}(C) - \Theta : \mathit{view}(r,t).\mathit{comp} \cap \Theta = \emptyset.$$

View Accuracy and View Completeness are of fundamental importance for every PGMS. They state that the composition of installed views cannot be arbitrary but must be a function of the actual unreachability pattern occurring during a run. Any specification that lacked a property similar to View Accuracy could be trivially satisfied by installing at every member either an empty view or a singleton view consisting of the member itself. The resulting service would exhibit what has been called *capricious view splitting* [4] and would not be very useful. View Accuracy prevents capricious view splitting by requiring that eventually, all views installed by two permanently-reachable members contain each other. On the other hand, the absence of View Completeness would admit implementations in which members always install views containing the entire group, again rendering the service not very useful.

Note that View Accuracy and View Completeness are stated slightly differently. This is because the reachable relation between objects is not transitive. While q being reachable

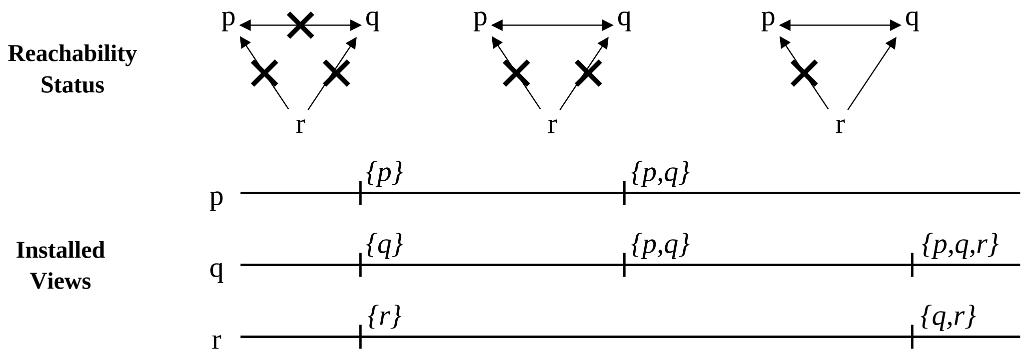

Figure 4.2: A scenario in which View Coherency is not satisfied.

directly from p is justification for requiring p to include q in its view, the converse is not necessarily true. The fact that a object p cannot communicate directly with another object q does not imply that p cannot communicate indirectly with q through a sequence of pairwise-reachable intermediate objects. For this reason, View Completeness has to be stated in terms of complementary sets of members rather than member pairs. Doing so assures that a member is excluded from a view only if communication is impossible because there exists no path, directly or indirectly, for reaching it.

View Accuracy and View Completeness state requirements for views installed by individual members. A group membership service that is to be useful must also place constraints on views installed by different members. Without such coherency guarantees for views, two members could behave differently even though they belong to the same partition but have different perceptions of its composition. For example, consider a system with two members p and q that are permanently reachable from each other (Figure 4.2). By View Accuracy, after some time t, both p and q will install the same view v containing themselves. Now suppose that at some time after t, a third member r becomes and remains reachable from q alone. Again by View Accuracy, q will eventually install a new view w that includes r in addition to itself and p. Presence of member r is unknown to p since they are not directly reachable. Thus, p continues believing that it shares the same view with q since its current view v continues to include q, when in fact member q has gone on to install view w different from v. The resulting differences in perception of the environment could lead members p and q to behave differently even though they belong to the same partition. The following property has been formulated to avoid such undesirable scenarios.

GM4 (View Coherency)

(i) If a correct member p installs view v, then either all members in $v.comp$ also install v, or p eventually installs an immediate successor to v. Formally,

$$p \in Correct(C) \wedge vchg(v) \in \sigma(p, \mathcal{T}) \wedge q \in v.comp \;\; \Rightarrow \;\; (vchg(v) \in \sigma(q, \mathcal{T})) \vee (\exists w : v \prec_p w).$$

(ii) If two members 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,

$$vchg(v) \in \sigma(p, \mathcal{T}) \wedge vchg(v) \in \sigma(q, \mathcal{T}) \wedge v \prec_p w_1 \wedge q \in correct(C) \;\; \Rightarrow \;\; \exists w_2 : v \prec_q w_2.$$

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

$$\sigma(p, t_0) = vchg(w) \wedge v \prec_p w \wedge q \in v.comp \cap w.comp \wedge q \neq p \;\; \Rightarrow \;\; vchg(v) \in \sigma(q, [0, t_0[).$$

Returning to the above example, the current view of member p cannot remain v indefinitely as Property GM4(ii) requires p to eventually install a new view. By assumption, q never installs another view after w. Thus, by Property GM4(i), the new installed by p must be w as well and include r. As a result, members p and q that belong to the same partition return to sharing the same view (Figure 4.3). In fact, we can generalize the above example to argue that View Coherency together with View Accuracy guarantee that every view installed by a correct member is also installed by all other members that are permanently reachable from it. Note that the composition of the final view installed by p and q includes member r as belonging to their partition. This is reasonable since p and r can communicate (using q as a relay) even though they are not reachable directly.

View Coherency is important even when reachability and unreachability relations are not persistent. In these situations where View Accuracy and View Completeness are not applicable, View Coherency serves to inform a member that it no longer shares the same view with another member. Consider two members p and q that are initially mutually reachable. Suppose that p has installed a view v containing the two of them by some time t. The current view of member q could be different from v at time t either because it never installs v (e.g., it crashes) or because it installs another view after having installed v (e.g., there is a network partitioning or merge). In both cases, Property GM4(i) and

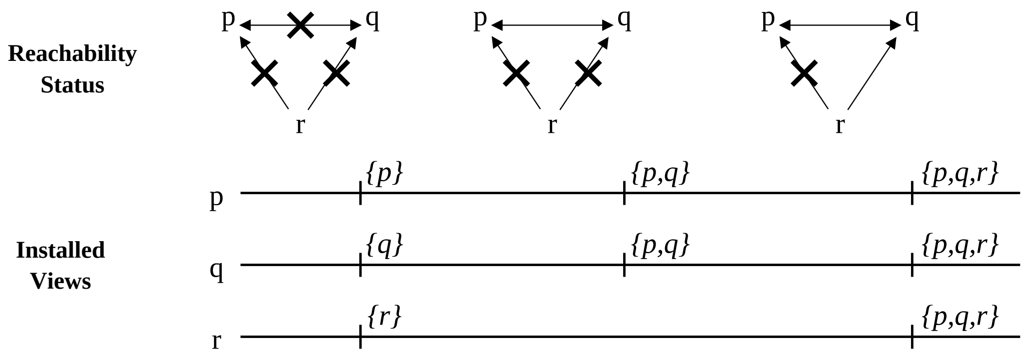

Figure 4.3: A scenario in which View Coherency is satisfied.

Property GM4(ii), respectively, ensure that object p will eventually become aware of this fact because it will install a new view after v.

When a object installs a new view, it cannot be sure which other members have also installed the same view. This is an inherent limitation due to asynchrony and possibility of failures. Property GM4(iii) allows a member to reason *a posteriori* about other members: At the time when object p installs view w as the immediate successor of view v, it can deduced which other members have also installed view v. And if some member q belonging to view v never installs it, we can be sure that q cannot belong to view w. Note that these conclusions are based entirely on *local* information (successive pairs of installed views) yet they allow a member to reason globally about the actions of other members.

The next property for group membership places restrictions on the order in which views are installed. In systems where partitionings are impossible, it is reasonable to require that all correct members install views according to some total order. In a partitionable system, this is not feasible due to the possibility of concurrent partitions. Yet, for a partitionable group membership service to be useful, the set of views must be consistently ordered by those members that do install them. In other words, if two views are installed by a member in a given order, the same two views cannot be installed in the opposite order by some other member.

GM5 (View Order)

The order in which objects install views is such that the successor relation is a partial order. Formally, $v \prec^* w \;\Rightarrow\; w \not\prec^* v$.

When combined with View Accuracy and View Coherency, View Order allows us to conclude that there is a time after which permanently reachable members not only install the same set of views, they install them in the same order.

The final property of our specification places a simple integrity restriction on the composition of the views installed by a member. By Definition 4.1(ii), every correct member is always reachable from itself. Thus, Property GM2 ensures that eventually, all views installed by a object will include itself. However, it is desirable that self-inclusion be a perpetual, and not only eventual, property of installed views.

GM6 (View Integrity)

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

$$vchg(v) \in \sigma(p, \mathcal{T}) \;\Rightarrow\; p \in v.comp.$$

Properties GM2–GM6 taken together define a *partitionable group membership service.*

4.2.2 Discussion

Note that Property GM2, Property GM3 and Property GM4 place constraints on the composition of the current view of a member in relation to characteristics of the operating environment or actions of other members. As such, they can be satisfied only by installing a new view whenever the current view of a member does not conform to the specification. In an asynchronous system, it is impossible to put a time bound on this action. Thus, the properties are stated so as to hold eventually. This is sufficient to guarantee that our specification of PGMS is *live* since it excludes implementations where installation of justified views are arbitrarily delayed.

Recall that Property GM2 and Property GM3 are stated in terms of runs where reachability and unreachability relations are persistent. They are, however, sufficient to exclude trivial solutions to PGMS also in runs where reachability and unreachability among members are continually changing due to transient failures. As an example, consider a system composed of two members p and q and a run R_0 where they are permanently mutually reachable. By View Accuracy and View Coherency, we know there is a time t_0 by which both p and q will have installed a view composed of themselves alone. Now, consider run R_1 identical to R_0 up to time $t_1 > t_0$ when p and q become unreachable. The behavior of members p and q under runs R_0 and R_1 must be identical up to time t_1 since they cannot distinguish between the two runs. Thus, if they install views composed of p and q by time t_0 under run R_0, they must install the same views also under run R_1 where reachability relations are not persistent but transient. This example can be generalized to conclude that any implementation satisfying our specification cannot delay arbitrarily installation of a new view including members that remain reachable for sufficiently long periods. Nor can it delay arbitrarily installation of a new view excluding members that remain unreachable for sufficiently long periods.

Asynchronous distributed systems present fundamental limitations for the solvability of certain problems in the presence of failures. Consensus [33] and primary-partition group membership [18] are among them. Partitionable group membership service, as we have defined it, happens to be not solvable in an asynchronous system as well. Here we give a proof sketch of this impossibility. Consider a system consisting of two members p and q. Let $\mathcal{R}$ be the set of all runs in which p and q are correct and are permanently reachable from each other. Any implementation of our specification must guarantee that for all runs in $\mathcal{R}$, there exists a time after which the current view of p always contains q.

For contradiction, we will show that there is at least one run in $\mathcal{R}$ for which such a time does not exist. Let R_0 be a run in $\mathcal{R}$, and let t_0 be a time value. There are two possibilities: if the current view of p never contains q after t_0 in R_0, our claim is trivially proven. Thus, suppose the current view of p contains q at time t_0 in run R_0. Now consider another run R'_0, identical to R_0 up to t_0 at which time member q crashes. By Property GM3, there must be a time $t'_0 > t_0$ by which p installs a view v in R'_0 that excludes q. Now, consider run $R_1 \in \mathcal{R}$ such that (i) the global and failure histories of R_1 are identical to those of R'_0 up to time t_0; (ii) the global and failure histories of R_1 restricted to p are identical to those of R'_0 up to time t'_0, and (iii) all messages sent by q after t_0 are received by p after t'_0. At time t'_0, object p cannot distinguish run R_1 from R'_0. Thus, p installs view v at time t'_0 in run R_1 just as it does in run R'_0. Again, there are two possibilities. If the current view of p never contains q after t'_0 in R_1, our claim is proven. Otherwise, let $t_1 > t'_0$ be a time such that the current view of p contains q in R_1 at time t_1. We repeat this construction now starting from run R_1 and time t_1, obtaining a run $R_2 \in \mathcal{R}$ in which p installs a view excluding q at time $t'_1 > t_1$. By iterating the construction, we obtain a run $R \in \mathcal{R}$ in which the current view of p never contains q after a certain time, or p

installs an unbounded number of views that exclude q.

This impossibility result for PGMS can be circumvented by requiring certain stability conditions to hold in an asynchronous system. This stability conditions are formalized by the failure detector properties described in Section 4.1.3. In the next chapter we provide an algorithm implementing our specification using this failure detector, and we prove its correctness in Appendix A.

4.3 The Group Method Invocation Service

The aim of this section is to discuss the specification of GMIS. We start out by defining some terms and introducing notation. We also specify some integrity requirements that apply to both internal and external invocations. Then, we illustrate how the *view synchrony* paradigm has been adapted in order to deal with group method invocations. Finally, we specify the behavior of GMIS for internal and external method invocations.

Each server p is characterized by an *internal remote interface* $int_I(p)$ and an *external remote interface* $int_E(p)$, where the former consists of the collection of methods that may be invoked on p by clients, while the latter consists of the methods that may be invoked on p by servers. Every method is uniquely identified through a name m. For sake of simplicity, in this chapter we do not consider the problems related to formal parameters and actual arguments for method invocations, and focus our attention on the issues related to fault-tolerance and distribution. For the same reason, the specification does not consider issues related to inheritance, implementation of multiple interfaces, overriding, etc. We assume that members of a group implement the same external and internal interfaces; in this way, it is possible to associate unique external and internal interfaces with groups, respectively denoted by $int_E(g)$ and $int_I(g)$.

A server p *performs an internal invocation* of method m on the group to which p belongs if p executes event $\mathit{invoke}(m, k)$. The *invocation identifier* k serves to distinguish multiple invocations of the same method. A client *performs an external invocation* of method m on object group g if it executes event $\mathit{invoke}(g, m, k)$. Note that external invocations need to mention the object group explicitly while it is implicit for internal invocations.

Invocation k of method m *returns* value r through the execution of event $\mathit{return}(m, k, r)$ by the invoker. How the return value is related to the values returned by servers executing the methods is specified in Sections 4.3.3 and 4.3.4. Here, we simply assume that before executing event $return(m, k, r)$, an object must have performed the corresponding invocation, and that each object executes event $return$ for a given invocation at most once. Formally,

$$\text{(I1)} \quad \sigma(p,t) = return(m,k,r) \;\Rightarrow\; invoke(m,k) \in \sigma(p,[0,t[) \vee invoke(g,m,k) \in \sigma(p,[0,t[)$$

$$\text{(I2)} \quad \sigma(p,t) = return(m,k,r) \;\Rightarrow\; return(m',k,r') \notin \sigma(p,\mathcal{T} - \{t\}.$$

Invocations cause the execution of the invoked method at certain number of servers, depending on the failure scenario. Execution of method m at server p corresponding to invocation k results in a sequence of events determined by the internal state of p and by the algorithm associated with m. This sequence starts with $method(m, k)$ and, in the absence of failures, terminates with $\overline{method}(m, k, r)$, where r is the return value. We say that object p *completes* invocation k of method m at time t if p executes event $\overline{method}(m, k, r)$ at time t.

We must guarantee that before a server executes event $method(m, k)$, some other object (client or server) must have performed method invocation k. Formally,

$$\text{(I3)} \quad \sigma(p,t) = method(m,k) \;\Rightarrow\; \exists q : invoke(m,k) \in \sigma(q,[0,t[) \vee invoke(g,m,k) \in \sigma(q,[0,t[)$$

We assume that view installations cannot interrupt the execution of a method: given an event $method(m, k)$ executed by p at time t_1 and an event $\overline{method}(m, k, r)$ executed by p at time t_2, no vchg upcall may be executed by p in the period occurring between t_1 and t_2. Formally,

$$\text{(I4)} \quad \sigma(p,t_1) = method(m,k) \wedge \sigma(p,t_2) = \overline{method}(m,k,r) \;\Rightarrow\; vchg(v) \notin \sigma(p,[t_1,t_2]) \wedge \overline{vchg}(v,r) \notin \sigma(p,[t_1,t_2]).$$

Without this assumption, it would be difficult to program methods whose actions were dependent on the composition of the current view.

Properties I1–I4 taken together define the integrity requirements of our group method invocation service:

GMI1 (Invocation Integrity)
GMIS must satisfy properties I1–I4 defined above.

In the following, we use $methods(p, v)$ to denote the set of method invocations that have been completed by object p in view v. Formally,

$$methods(p, v) = \{(m, k) \mid \sigma(p, t) = \overline{\mathrm{method}}(m, k, r) \wedge view(p, t) = v\}.$$

4.3.1 View Synchrony

Ideally, all correct servers belonging to a given view should complete the same set of method invocation in that view. In partitionable systems, this requirement could result in an invocation to block if at least one server belonging to the view becomes unreachable before having had a chance to complete the invocation. In that case, the invocation would terminate only upon the repair of the communication failures. Thus, we relax this condition on the completion of method invocations as follows. First of all, a correct server completing an invocation is guaranteed that each server in the current view will complete the same invocation, unless it crashes or becomes partitioned. Servers completing an invocation are eventually notified of servers not completing it through the installation of a new view. Furthermore, two servers surviving from a view to its immediate successor complete the same set of invocations in the first view. These requirements are summarized by the following properties:

GMI2 (Invocation Liveness)
If a correct server p completes an invocation in a view v, then each server included in v will eventually complete the same invocation, or p will eventually install a new view after v. Formally,

$$\sigma(p, t) = \overline{\mathrm{method}}(m, k, r) \wedge p \in Correct(C) \wedge view(p, t) = v \wedge q \in v.comp \;\Rightarrow$$
$$(\exists r' : \overline{\mathrm{method}}(m, k, r') \in \sigma(q, \mathcal{T})) \vee (\exists w : v \prec_p w)$$

GMI3 (Invocation Agreement)
Given two views v, w such that w is an immediate successor of v, all servers belonging to both views complete the same set of invocations in view v. Formally,

$$v \prec_p w \wedge q \in v.comp \cap w.comp \;\Rightarrow\; methods(p, v) = methods(q, w).$$

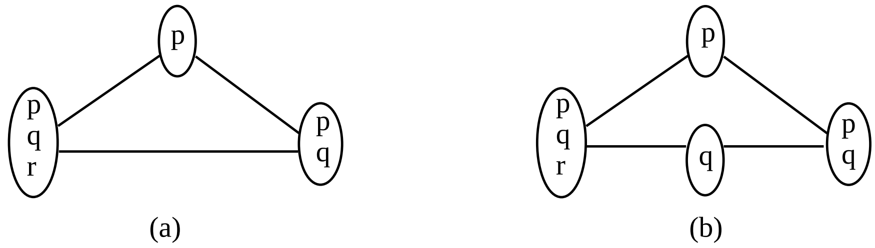

Figure 4.4: Merging scenarios. Ovals depict view compositions as sets of server names. Directed edges depict *immediate successor* relations between views.

Note that these properties, corresponding to *liveness* and *safety* of view synchrony, are stated in terms of "invocations", as they hold for both internal and external invocations.

The view synchrony paradigm may be considered as the partitionable counterpart of *virtual synchrony* [15]. Virtual synchrony has proven to be an extremely useful paradigm for developing fault-tolerant applications [14], and view synchrony is an attempt to provide developers of partition-aware applications with a tool of the same power. Unfortunately, the two properties introduced in this section do not satisfy this promise. Consider the scenario depicted in Figure 4.4(a) where three servers p, q, r have all installed view v_1. At some point server r crashes and p becomes temporarily unreachable from q. Server p reacts to both events by installing view v_2 containing only itself before merging back with q and installing view v_3. Server q, on other hand, reacts only to the crash of r and installs view v_3 excluding r. Suppose that p and q share the same state in view v_1 and that p modifies its state during v_2. When p and q install v_3, p knows immediately that their states may have diverged, while q cannot infer this fact based on local information alone. Therefore, q could behave inconsistently with respect to p. In an effort to avoid this situation, p could collaborate by invoking a special "warning" method on q as soon as it installs view v_3, but q could perform inconsistent operations before receiving such an invocation. The problem stems from the fact that views v_1 and v_2, that merge to form view v_3, have at least one common member (p). The scenario of the above example can be easily generalized to any run where two overlapping views merge to form a common view. We rule out these runs with the following property.

GMI4 (Merging Rule)

Two views merging into a common view must have disjoint compositions. Formally,

$$(v \prec u) \land (w \prec u) \land (v \neq w) \ \Rightarrow \ v.comp \cap w.comp = \emptyset.$$

The sequence of view installations in a run respecting this property is shown in Figure 4.4(b): Before installing v_3, server q has to first install view v_4. Thus the two views that merge to form v_3 have empty intersection. As a result, when p and q install view v_3, they both knows immediately that their states could have diverged during the partitioning.

Property GMI4, when integrated with Property GMI2 and Property GMI3, enables servers to reason about the state of other servers using their local knowledge about installed views and completed invocations: Server p knows that all other servers surviving a view change along with it have completed the same set of internal invocations. And if two servers share some global state in a view and this state depends only on the set of delivered messages regardless of their order, then they will continue to share the same state in the next view if they both survive the view change[3]. Note that Property GMI4 may appear to be part of the group membership specification since it is concerned with view installations alone. Nevertheless, we choose to include it as part of the group method invocation service specification since Property GM4 becomes relevant only in the context of invocation deliveries.

4.3.2 Postponed Invocations

In an invocation-based, view-synchronous group communication service, the invocation and the execution of methods occur within the context of views. So far, we have not expressed which is the relationship between the view in which a method invocation is performed and the view in which the message method is executed. Ideally, these view should always coincide. In the literature, a similar requirement for message-based group communication services is called *sending view delivery* [77]. This property is exploited by applications to minimize the amount of context information that needs to be sent with each message; moreover, there are cases in which applications are only interested in processing messages (in the Jgroup case, invocation) that arrive in the view in which they

[3] For applications where the shared state is sensitive to the order in which messages are delivered, specific order properties can be enforced by additional system layers.

are sent [77]. This is usually the case with state transfer messages sent when new views are installed [2, 72, 40, 35, 1, 42, 45, 44]

Unfortunately, Friedman and van Renesse [34] have proven that in order to satisfy sending view delivery, servers must block sending of messages for a certain period of time before a new view is installed. This *blocking period* is needed to reach an agreement on a new view and on the set of messages delivered in the current one. Without this blocking, other useful properties would be violated, such as view view synchrony described in the previous section. Jgroup postpones invocation of methods while a view change is taking place. In order to notify the application that its invocations will be postponed until a new view is installed, Jgroup invokes the upcall method prepare on application servers. In the global history, the execution of this upcall in view v is associated with events $prepare(v)$ and $\overline{prepare}(v)$; they corresponds to the beginning and the end of the upcall execution, respectively. As described in the next section, invocations performed before the conclusion of the prepare upcall will be completed in the same view in which they have been performed; invocations performed after the conclusion of the prepare upcall will be completed in the successor view.

As integrity requirement, we simply assume that events associated with prepare upcall for a given view must be executed in that view, before installing the following one. This condition is formalized in the following property:

GMI5 (Integrity of Blocking Period)

When a server p installs a view w as the immediate successor to view v, then p executed event $\overline{\text{prepare}}(v)$ in v. Furthermore, events $\text{prepare}(v)$ and $\overline{\text{prepare}}(v)$ may be executed only in view v. Formally,

$$(i) \quad \sigma(p,t) = vchg(w) \wedge v \prec_p w \;\Rightarrow\; \exists t' < t : \sigma(p,t') = \overline{prepare}(v) \wedge \mathsf{View}(p,t') = v$$

$$(ii) \quad \sigma(p,t) = prepare(v) \;\Rightarrow\; \mathsf{View}(p,t) = v$$

$$(iii) \quad \sigma(p,t) = \overline{prepare}(v) \;\Rightarrow\; \mathsf{View}(p,t) = v.$$

As an additional liveness property, we require that prepare upcalls executed by correct servers are always followed by a view installation; in other words, we do not admit scenarios in which performed invocations are postponed to the next view, but no next view is installed.

GMI6 (Termination of Blocking Period)

If a correct server p executes an event $\overline{\text{prepare}}(v)$, then eventually the server will install the successor view of v. Formally,

$$\sigma(p,t) = \overline{\text{prepare}}(v) \;\Rightarrow\; \exists w : v \prec_p w.$$

4.3.3 The Internal Group Method Invocation Service

After having joined group g, server p may perform an internal invocation of method m on group g, provided that m belongs to $int_I(g)$. The task of GMIS is to force the execution of method m at each of the members included in the current view, including the invoker itself; the invocation terminates when a *reply set* is returned to the invoker, containing the return values obtained by servers executing the method.

The first property of our specification regards termination of internal invocations. If the invoker is correct, internal invocation k of method m will eventually terminate with event $return(m,k,R)$. Invocations performed in a view before the beginning of its blocking period terminate in the same view, while invocations performed in the blocking period of a view are postponed until the next view is installed. The reply set R contains a collection of pairs (p,r), where p is a server identifier and r is a return value. Note that synchronous and asynchronous internal invocations do not require to be specified differently; a *return* event for an internal invocation may correspond to either an actual return statement, or to the invocation of a notification method on a callback object (see Section 3.5 for details).

IGMI1 (Internal Invocation Termination)

A correct server performing an internal invocation will eventually obtain a reply set for that invocation. If the invocation has been performed in view v before event $\overline{\text{prepare}}(v)$, it will terminate in v; otherwise, if it has been performed in view v after event $\overline{\text{prepare}}(v)$, it will terminate in the successor view. Formally,

$$(i) \quad \sigma(p,t) = invoke(m,k) \wedge p \in correct(C) \wedge \mathsf{View}(p,t) = v \wedge \overline{\text{prepare}}(v) \in \sigma(p,[0,t[)$$
$$\Rightarrow \exists R, \exists t' > t : \sigma(p,t') = return(m,k,R) \wedge \mathsf{View}(p,t') = v$$

$$(ii) \quad \sigma(p,t) = invoke(m,k) \wedge p \in correct(C) \wedge \mathsf{View}(p,t) = v \wedge \overline{\text{prepare}}(v) \notin \sigma(p,[0,t[)$$
$$\Rightarrow \exists R, \exists t' > t : \sigma(p,t') = return(m,k,R) \wedge \mathsf{View}(p,t') = w \wedge v \prec_p w.$$

Clearly, the set of values contained in a reply set cannot be arbitrary, but must be related with the values actually returned by servers completing the method. The following property rules out undesirable scenarios in which reply sets contained values generated "out of thin air", i.e. never returned by any server.

IGMI2 (Internal Invocation Validity)

If the pair (p, r) is included in the reply set R obtained by a server q, then server p must have previously returned r through an event $\overline{\mathrm{method}}(m, k, r)$. Formally,

$$\sigma(q,t) = \mathbf{return}(m,k,R) \wedge (p,r) \in R \;\Rightarrow\; \overline{\mathit{method}}(m,k,r) \in \sigma(p,[0,t[).$$

The properties given so far may be trivially satisfied by returning empty reply sets containing no values. On the contrary, reply sets should contain a value for each of the servers perceived as reachable by the group membership service. In other words, in the absence of new view installations, we require that a reply set obtained in a view v contains a return value for each of the servers composing v. Otherwise, if a failure occurs and a new view is installed, we require that the reply set contains a value for each of the servers survived from the previous view (i.e., that are in the intersection between the new and the previous views). These requirements are summarized in the following property:

IGMI3 (Internal Invocation Non-Triviality)

(i) If a correct server p obtains a reply set R for an internal invocation performed in view v, then R contains a return value for each of the servers included in v, or p will eventually install a new view after v. Formally,

$$\sigma(p,t) = \mathbf{return}(m,k,R) \wedge p \in \mathit{correct}(C) \wedge \mathit{view}(p,t) = v \wedge q \in v.\mathit{comp} \;\Rightarrow$$
$$(\exists r : (q,r) \in R) \vee (\exists w : v \prec_p w).$$

(ii) Given two views v, w such that w is an immediate successor of v, all reply sets R obtained through event $\mathit{return}(m,k,R)$ executed in view v are such that there is a pair (q,r) for each server q that survives from v to w. Formally,

$$v \prec_p w \wedge q \in v.\mathit{comp} \cap w.\mathit{comp} \wedge \sigma(p,t) = \mathbf{return}(m,k,R) \wedge \mathit{view}(p,t) = v \;\Rightarrow$$
$$\exists r : (q,r) \in R.$$

Note that this requirement, when combined with property View Integrity defined in the previous section, guarantees that every reply set contains at least the value returned by the invoker itself.

Each return value contained in the reply set obtained from an internal invocations may be associated with the view in which the corresponding server has actually completed the invocation itself. If the same method invocation has been completed by two servers in different views, their return values may be inconsistent, as they are related to views that may have different composition. The next property avoids this unfavorable scenario by requiring that completions of the same method invocation occur in at most one view.

IGMI4 (Internal Invocation Uniqueness)
An internal invocation is completed in at most one view. Formally,

$$\sigma(p,t) = invoke(m,k) \wedge k \in methods(q,v) \wedge k \in methods(r,w) \;\;\Rightarrow\;\; v = w.$$

Properties IGMI1–IGMI4, together with Properties GMI1–GMI6, specify the behavior of GMIS with respect to internal method invocations.

4.3.4 The External Group Method Invocation Service

Clients access services offered by an object group through external invocations of group methods. As we have seen, external invocations may follow two different semantics. The anycast semantics guarantees that an external method invocation on a group of servers will force the execution of the method on at least one of the servers, unless the client is completely partitioned from the object group. The multicast semantics guarantees that an external method invocation on a group of servers will force the execution of the method on every server contained in the client's partition, following the view synchrony semantics described in the previous section.

A client p may perform an external invocation of a method m on group g provided that $m \in int_E(g)$. External invocations differ from internal invocations in two aspects. In order to preserve transparency of clients, external invocations return a single value to the invoker (rather than a set), exactly as in standard remote method invocations. In other words, clients need not be aware of the fact that the service they are accessing is replicated. The second difference is that external invocations may fail, since there may be no functional servers that are reachable from the client. In these cases, an exceptional value $\perp$ is returned.

The GMI specification for external invocations is very simple. First of all, external invocations performed by correct clients always terminate, either by returning a normal value or by returning $\perp$. This requirement gives clients the possibility to react to crashes or partitionings without remaining permanently blocked during an invocation. Obviously, a trivial implementation that always returned $\perp$ would satisfy this requirement. To avoid this, we require that external invocations on a group eventually return a value different from $\perp$ if at least one server in the group is permanently reachable by the client. Finally, we add a validity property requiring that return values different from $\perp$ must correspond to a return value generated by some server. How to select the single value to return to the client among several possibilities is left to the implementation. One possibility is to pick the first value to be produced by a server. These observations are summarized through the following properties:

EGMI1 (External Invocation Termination)

External invocations performed by correct clients always terminate. Formally,

$$\sigma(p,t) = \mathrm{invoke}(g,m,k) \wedge p \in \mathit{correct}(C) \;\Rightarrow$$
$$\exists r \in \mathcal{R} \cup \{\perp\} : \mathrm{return}(m,k,r) \in \sigma(p,]t, \infty[).$$

EGMI2 (External Invocation Non-Triviality)

If there is a time after which server q remains permanently reachable from a correct client p and q is permanently a member of group g, then eventually return values obtained by p for external invocations performed on g are different from $\perp$. Formally,

$$\exists t_0, \forall t \geq t_0 : p \leadsto_t q \wedge q \in \mathit{memb}(g,t) \;\Rightarrow$$
$$\exists t_1, \forall t > t_1 : (\sigma(p,t) = \mathrm{return}(m,k,r) \;\Rightarrow\; r \neq \perp).$$

EGMI3 (External Invocation Validity)

A return value r different from $\perp$ obtained through an event $\mathrm{return}(m,k,r)$ *for an external invocation on group g must have been previously returned by a server in g through an event* $\overline{\mathrm{method}}(m,k,r)$*. Formally,*

$$\sigma(p,t_0) = \mathrm{return}(m,k,r) \wedge \mathrm{invoke}(g,m,k) \in \sigma(p,[0,t_0[) \;\Rightarrow$$
$$\exists q, \exists t_1 < t_0 : \overline{\mathrm{method}}(m,k,r) \in \sigma(q,t_1) \wedge q \in \mathit{memb}(g,t_1).$$

These properties applies to external invocations with both anycast and multicast semantics; in particular, Properties EGMI1–EGMI3 specify completely the behavior of external invocations with anycast semantics. Properties EGMI1–EGMI3, together with Properties GMI1–GMI4, specify the behavior of external invocations with multicast semantics.

4.4 The State Merging Service

In this section, we present the specification of the state merging service included in the Jgroup toolkit. As we have seen in Section 3.6, the functioning of the state merging service is based on upcall methods for "getting" state information from servers and subsequentially "merging" information coming from other servers to the local state. In the global history, the execution of these methods is represented by events $getState(m, D)$ and $mergeState(m, S)$, respectively, where m represents the information about the state, D is the destination set of m (i.e. a collection of members that are supposed to require information contained in m), and S is the source set of m (i.e. a collection of members that already know the information contained in m).

Given a server p, its update set at time t is denoted with $\textit{up-to-date}(p, t)$. Initially, the update set of a server contains only the server itself. Then, the composition may change because the server may become up-to-date with respect to other servers; in particular, a server p executing an event $mergeState(m, S)$ results in the update set of p augmented with the servers included in S. The composition may change also due to view changes generated by the group membership service. The installation of a new view at member p causes the removal from the update set of p of all members that have been excluded from the new view. These members are removed because are partitioned, and thus p may be not up-to-date with respect to them any more. In the absence of the events listed here, the composition of an update set remains unaltered. These requirements are summarized by the following property:

SMS1 (Subview Composition)

(i) Initially, the update set of a server p contains only p itself. Formally,

$$\textit{up-to-date}(p, 0) = \{p\}.$$

(ii) The execution of an event vchg(v) *by member* p *causes the removal of all members not included in* p *from the update set of* p. *Formally,*

$$\sigma(p,t) = \text{vchg}(v) \Rightarrow \textit{up-to-date}(p,t) - v.comp = \emptyset.$$

(iii) The execution of an event mergeState(m,S) *by* p *forces the insertion of members contained in* S *in the update set of* p. *Formally,*

$$\sigma(p,t) = \text{mergeState}(m,S) \Rightarrow S \subseteq \textit{up-to-date}(p,t).$$

(iv) The update set of p *may vary only due to* vchg *and* mergeState *events executed by* p *itself. Formally,*

$$\sigma(p,t+1) \neq \text{vchg}(v) \wedge \sigma(p,t+1) \neq \text{mergeState}(m,S) \Rightarrow$$
$$\textit{up-to-date}(p,t+1) = \textit{up-to-date}(p,t).$$

Upcalls *getState* and *mergeState* are not executed by SMS arbitrarily, but only when the failure scenario actually requires the execution of a state merging protocol. In particular, an upcall *getState* may be generated only if its execution is needed to make a server up-to-date with respect to another one; furthermore, an upcall *mergeState* carrying some state information m may be executed only if a corresponding *getState* upcall returning m has been executed previously. These considerations are formalized by the following property:

SMS2 (State Merging Integrity)
(i) A member p *executes an event* getState(m,D) *during a view* v *only if there is a server* q *in* $v.comp$ *not up-to-date with respect to* p *when* p *installs* v. *Formally,*

$$\sigma(p,t) = \text{vchg}(v) \wedge \sigma(p,t') = \text{getState}(m,D) \wedge \textit{view}(p,t') = v \Rightarrow$$
$$\exists q \in v.comp : p \notin \textit{up-to-date}(q,t).$$

(ii) A member p *executes an event* mergeState(m,S) *only if some other member included in* S *has previously performed an event* getState(m,D). *Formally,*

$$\sigma(p,t) = \text{mergeState}(m,S) \Rightarrow \exists t' < t : \textit{getState}(m,D) \in \sigma(q,t') \wedge q \in S.$$

When executing a $mergeState(m, S)$ event, it could be useful for a server to know which other servers in its current view have also executed $mergeState(m, S)$. Having this knowledge, a server may easily maintain information about the set of updates appliedy by other servers, and thus reduce the amount of update to transmit when elected as a coordinator. For this reason, the SMS specification includes a property similar to view synchrony, that requires that servers surviving from a view to another execute the set of $mergeState(m, S)$ events.

SMS3 (State Merging Agreement)

Given two views v, w such that w is an immediate successor of v, all servers belonging to both views complete the same set of mergeState *events in view v. Formally,*

$$v \prec_p w \wedge q \in v.comp \cap w.comp \Rightarrow P_p^v = P_q^v.$$

where $P_p^v = \{\ (m, S) | \sigma(p, t) = mergeState(m, S) \wedge view(p, t) = v\ \}$ corresponds to the set of mergeState *events executed by p in v, assuming that each update information m is unique.*

The properties presented so far specify the *safety* requirements of our state merging service. Clearly, the specification of the SMS would not be complete without a *liveness* requirement. As introduced in Section 3.6, a state merging protocol has to be executed whenever a member included in a view is not up-to-date with respect to another member included in the same view. As a liveness requirement, we require that if two members are permanently reachable from each other after a certain time (i.e., they stop installing views excluding the other), the state merging protocol among them will eventually terminate; i.e., each of them will be eventually up-to-date with respect to the other. This requirement is specified by the following property:

SMS4 (State Merging Termination)

Let t_0 be a time at which a server p installs a view containing a server q. Suppose that after t_0, all views installed by p contains q; moreover, that q is not included in the update set of p at time t_0. Then, p will eventually execute an event $mergeState(m, S)$ *such that $q \in S$. Formally,*

$$\sigma(p, t_0) = vchg(v) \wedge \forall t > t_0 : q \in view(p, t) \wedge q \notin up\text{-}to\text{-}date(p, t_0) \Rightarrow$$
$$\exists t_1 > t_0 : \sigma(p, t_1) = mergeState(m, S) \wedge q \in S.$$

Properties SMS1–SMS4 define the specification of the state merging service included in Jgroup.

Chapter 5

The Jgroup Prototype

A prototype for Jgroup has been developed in the context of this thesis. The aim of this chapter is to describe its implementation. We start out by presenting an overview of the architecture of the Jgroup toolkit (Section 5.1) and by introducing notation (Section 5.2). Jgroup is composed of a collection of layers that cooperate in order to implement the specification presented in Chapter 4. Sections 5.3–5.8 describe the algorithms implementing each of these layers. Finally, Section 5.9 illustrates some of the details of the Java implementation.

5.1 Overview of the Jgroup Architecture

The system model underlying Jgroup, introduced in Section 3.1, is illustrated in Figure 5.1. The system is composed of a collection of Java virtual machines connected through a communication network. Each Java virtual machine hosts a collection of objects. Server objects coordinate their actions in order to provide some service to client objects. Clients access the services provided by object groups through group method invocations.

As introduced in Chapter 3, group communication facilities are provided to server objects by *group managers*, which act as proxies for performing internal group method invocations and provide primitives for group management. Each server is associated with one group manager for each of the group it joins, while each group manager is associated with exactly one server. Group managers are responsible for notifying their servers of group communication events such as view changes; furthermore, group managers take also care of dispatching internal and external invocations.

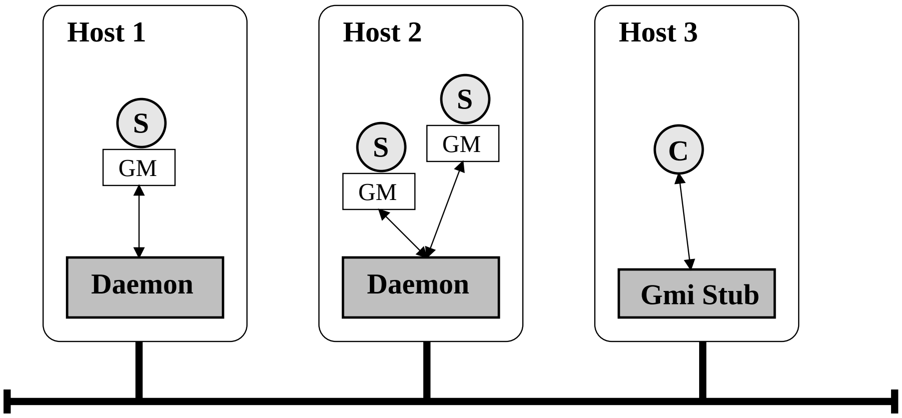

Figure 5.1: The Jgroup architecture

Group managers do not implement the entire group communication service described and specified in the previous chapters. Instead, basic group membership and multicast communication facilities are implemented in a separate module called a *Jgroup daemon*. In a JVM hosting Jgroup servers, a single instance of Jgroup daemon is executed. Jgroup daemons communicate among themselves to exchange messages and reachability information.

There are several motivations behind the choice of using this architectural model, as opposed to having the group communication service provided by every group manager:

- The number of message exchanged to establish group communication is reduced; for example, low-level services as failure detection are implemented by Jgroup daemons on a per-JVM basis, and not replicated by every group manager.

- With this model, it is possible to distinguish between server objects local to a given JVM and those that are remote. Servers local to a given JVM always share the same destiny with respect to failures: the crash of a JVM causes the crash of all servers hosted in it, and if a JVM becomes partitioned from the rest of the system, all its local servers stop communicating with remote servers. Thus, two distinct membership are maintained, one regarding servers, and the other regarding daemons. Voluntary variations in the membership due to join and leave operations are translated in single messages requiring to modify the list of servers hosted in a daemon, without requiring complex agreement protocols.

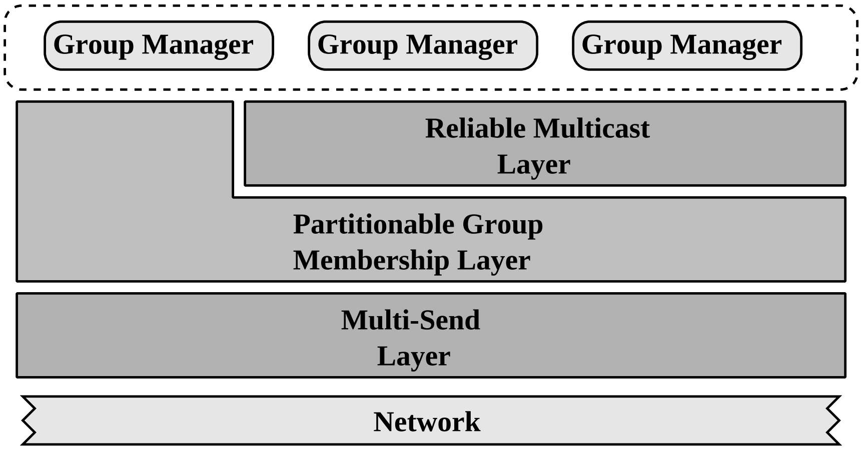

Figure 5.2: The architecture of a Jgroup Daemon

- This daemon-based architecture may be further extended and improved by enabling server objects on a JVM to use the daemon running on another JVM. This may be interesting for scalability, as it would be possible to structure the system in order to have a single instance of Jgroup daemon in each LAN containing server object. In this way, it would be possible to share the costs of maintenance of the membership protocol among servers composing a LAN, thus reducing the number of messages exchanged. This extension, however, is not already included in the Jgroup prototype.

Clients do not require Jgroup daemons and group managers running in their virtual machines; interactions between clients and servers are handled by a component called a *GMI stub*. Each stub enables clients to perform external method invocations on a single group; thus, several stubs may co-exist in the same JVM.

Each Jgroup daemon has a layered structure, illustrated in Figure 5.2, composed of the following logical components:

- *Multi-Send Layer* (MSL): Implementing a group membership service directly on top of a point-to-point unreliable, unsequenced datagram transport service provided by the network would be difficult. The difficulty is aggravated by the lack of transitivity of the reachability relation as discussed in Section 3.1. The task of MSL is to hide this complexity by transforming the unreliable, point-to-point network communication primitives to their best-effort, one-to-many counterparts. Informally, MSL tries to deliver messages sent through it to all daemons in some destination list. More-

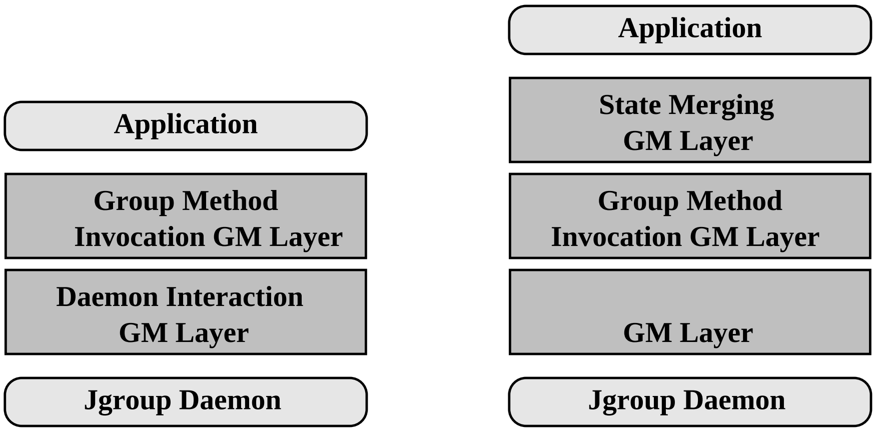

Figure 5.3: Two possible stack of group manager layers.

over, its keeps daemons informed about which servers are reachable and which are suspected to be unreachable. For this purpose, it integrates an eventually perfect failure detector (see Section 4.1.3 for details).

- *Partitionable Group Membership Layer* (PGML): The services provided by MSL are used by PGML in order to construct and install views as defined by the PGMS specification. The task of PGML is to manage the membership of multiple groups by serving *join* and *leave* requests coming from servers, and to translate the possibly inconsistent suspicion sets generated by MSL in agreed views.

- *Reliable Multicast Layer* (RML): The task of RML is to provide a message-based reliable multicast service. RML is integrated with the partitionable group membership layer and satisfy the message-based version of the view synchrony properties introduced in Section 4.3.1.

Jgroup daemons are the basic building block of the Jgroup architecture; they provide the fundamental components of a group communication service, such as failure detection, group membership and reliable multicasts. Additional facilities, such as group method invocations and state merging are provided by group managers. The internal architecture of group managers is highly modular and has been designed in order to make possible to easily insert new layers implementing additional facilities not described in this dissertation. For example, it may be easily possible to insert ordering layers for performing ordered group method invocations with FIFO, causal and atomic semantics.

Figure 5.3 illustrates the configurations that group managers may assume in the current version of Jgroup. Three different layers may compose a group manager:

- *Daemon Interaction Layer* (DIL): the task of this layer is to handle the interactions between group managers and daemons. This layer transmits basic group communication primitives (such as join, leave and mcast) to the Jgroup daemon running in the system; moreover, it enqueues the upcall requests coming from the Jgroup daemon and delivers them to the upper layers.
- *Group Method Invocation Layer* (GMIL): GMIL uses the services provided by RML through DIL to provide servers with the communication paradigm specified in Section 4.3. Since issues related to view synchrony are handled by RML, the only task of this layer is to manage the problems related to method dispatching, parameters marshaling and unmarshaling, and collection of return values.
- *State Merging Layer* (SML): Finally, this layer provides the state merging facilities described in Section 4.4.

In order for a group manager layer to be inserted in the group manager stack, it must implements two interfaces: Member to intercept and forward group communication notifications called up by lower layers, and GroupManager to accept group management primitives invoked by higher layers and forward them to lower layers. The only exception is the daemon interaction layer, which implements only interface GroupManager and interact directly with the Jgroup daemon. Definitions of Member and GroupManager may be found in Chapter 3.

In the rest of this chapter, we describe and analyze the main algorithms used to implement each of these layers. Since MSL and RML are internal layer not described in Chapter 4, we provide their specification as well.

5.2 Algorithm Structure and Notation

The Jgroup prototype is a complex toolkit. Apart from those described in this dissertation, other layers have been or are going to be implemented. Complex interactions exist among different layers composing the daemon, between daemons and group managers, and between group managers and stubs. Instead of illustrating the entire implementation with all its details, we have chosen to present selected sections of it, which highlight the basic programming issues related to Jgroup. The reader interested in all of the details of the implementation may analyze the code, that is available at the Jgroup home page[1]. In order to simplify the presentation, the algorithms presented here are limited to a single server per machine, and a single group per server. It is easy to extend the algorithms by allowing each machine to host more than one server using the same Jgroup daemon, and by enabling servers to join more than one group. It would be sufficient to add lists of local servers to the information exchanged among daemons, and to parameterize each message and each data structure with respect to group identifiers.

To further simplify the presentation, we also simplify the signatures of some of the methods defined in the classes and interfaces described in Chapter 3. For example, method join takes only one integer as argument, and upcall vchg takes only a view. Group-related data structures created by users have been removed from the argument list of both these methods, as only one group per server may be joined at a time. Another example of simplification is method getGroupManager, which does not require to specify a DistributedSystem object describing the set of host addresses in which servers are expected to be executed. As in the specification, we assume the existence of a static set of hosts represented by Π.

All the algorithms described here are expressed using pseudo-code based on Java, and mask most of their details by using complex data structures such as sets and associative arrays (dynamic arrays that can be indexed not only by primitive types, but also by complex objects). We use *italic* font for variable and method names, and sans-serif for class names and Jgroup methods. **Boldface** is used for primitive data types and keywords. Tags denoting message types are written in SMALLCAPS. Beside the usual ones, two new statements are added: **wait – for** is used to suspend the execution of a thread until a low-level event occurs (such as timeout expirations and message receipts); **foreach** is used to apply a block of code to each element that satisfies a condition.

5.3 The Multi-Send Layer

The task of the *multi-send layer* (MSL) is to provide daemons with best-effort, one-to-many communication primitives needed by PGML to implement the partitionable group

[1] `http://www.cs.unibo.it/projects/jgroup`

membership service specified in Section 4.2. All interactions with the communication network and the failure detector are limited to MSL which uses the unreliable, unsequenced datagram transport service of the network through the primitives $send()$ and $recv()$. Each MSL can also read the suspect list of the corresponding failure detector module FD.

Informally, MSL tries to deliver messages to all servers in some destination set. MSL also "filters" the raw failure detector suspect list by eliminating from it those servers that can be reached indirectly. In other words, the notion of reachability above MSL corresponds to the transitive closure of reachability at the failure detector layer. What distinguishes MSL from a typical network routing protocol is the integration of message delivery semantics with the reachability information. In that sense, MSL is much closer to the *dynamic routing layer* of Phoenix [49] or the MUTS layer of Horus [75].

5.3.1 Specification

In order to distinguish between the various layers in our discussion, we say that a server *m-sends* and *m-receives* messages when it communicates through the MSL primitive $msend()$ and the MSL upcall $mrecv()$, respectively. We reserve *send* and *receive* to denote communication directly through the network services without going through MSL. Similarly, we say that a server *m-suspects* those servers that are notified through a $msuspect()$ event while *suspect* is reserved for describing the failure detector itself.

Let $\mathcal{R}(p,t)$ denote the *reachable set function* defined as those servers that are not m-suspected by p at t. In other words, $q \in \mathcal{R}(p,t)$ if and only if the last $msuspect(P)$ event generated at server p by time t is such that $q \notin P$. Properties of MSL that are needed by the group membership algorithm are as follows.

MS1 (MSL Specification) *The Multi-Send Layer (MSL) satisfies the following properties: (a) if a server q is continuously unreachable from p, then eventually p will continuously m-suspect q; (b) if server q is continuously reachable from p, then eventually each server will permanently m-suspect both or none of p and q; (c) each server m-receives a message at most once and only if some server actually m-sent it earlier; (d) messages from the same sender are m-received in FIFO order; (e) a message that is m-sent by a correct server p is eventually m-received by all servers in the destination set that are not m-suspected by p; (f) function $\mathcal{R}$ is perpetually reflexive; (g) function $\mathcal{R}$ is eventually symmetric. Formally,*

$$
\begin{array}{ll}
(a) & p \in \Theta \wedge q \notin \Theta \wedge (\exists t_0, \forall t \geq t_0 : p \not\rightsquigarrow_t q) \;\Rightarrow \\
 & \exists t_1, \forall r \in \mathit{correct}(C) \cap \Theta, \forall t \geq t_1 : \mathcal{R}(r,t) - \Theta = \emptyset \\
(b) & \exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \;\Rightarrow\; \exists t_1, \forall t \geq t_1, \forall r : p \in \mathcal{R}(r,t) \;\Leftrightarrow\; q \in \mathcal{R}(r,t) \\
(c) & \sigma(p,t) = \mathrm{mrecv}(m,q) \;\Rightarrow\; \mathrm{mrecv}(m,q) \notin \sigma(p, \mathcal{T} - \{t\}) \wedge \\
 & \mathrm{msend}(m, \mathit{dests} \cup \{p\}) \in \sigma(q, [0,t[) \\
(d) & \sigma(p,t_1) = \mathrm{mrecv}(m_1,q) \wedge \sigma(p,t_2) = \mathrm{mrecv}(m_2,q) \wedge t_1 < t_2 \;\Rightarrow \\
 & \exists t'_1, t'_2 : t'_1 < t'_2 \wedge \sigma(q,t'_1) = \mathrm{msend}(m, \mathit{dests}_1) \wedge \sigma(q,t'_2) = \mathrm{msend}(m, \mathit{dests}_2) \\
(e) & \sigma(p,t) = \mathrm{msend}(m, \mathit{dests} \cup \{q\}) \;\Rightarrow \\
 & \exists t' > t : \sigma(q,t') = \mathrm{mrecv}(m,p) \vee q \notin \mathcal{R}(p,t') \vee p \in C(t') \\
(f) & p \in \mathcal{R}(p,t) \\
(g) & p, q \in \mathit{correct}(C) \wedge (\exists t_0, \forall t \geq t_0 : q \in \mathcal{R}(p,t)) \;\Rightarrow\; (\exists t_1, \forall t \geq t_1 : p \in \mathcal{R}(q,t))
\end{array}
$$

Properties (a) and (b) are the non-triviality conditions of our communication service. Properties (c) and (d) place simple integrity and order requirements on m-receiving messages. Property (e) defines a liveness condition on the m-sending of messages. Finally, property (f) prevents servers from m-suspecting themselves, while property (g) requires that if a correct server p stops m-suspecting another correct server q, then eventually q will stop m-suspecting p. It is important to note that from the combination of properties (b) and (f) we conclude that if q is continuously reachable from p, then p eventually stops m-suspecting q. Moreover, from properties (b) and (e) we conclude that if q is continuously reachable from p, then every message m-sent by q to p is eventually m-received.

5.3.2 Algorithm

In Figures 5.4 and 5.5, we illustrate an algorithm for implementing MSL. The aim of the MSL algorithm presented here is to illustrate that the properties defining the MSL specification may be satisfied in an asynchronous distributed system augmented with a failure detector of class $\diamond\tilde{\mathcal{P}}$. For this reason, the algorithm is designed for simplicity rather than efficiency. The algorithm uses a simple flooding strategy based on the forwarding of every received message on each output channel. Note that the actual implementation of the algorithm is more efficient, as it is based on distance vector routing.

```
class MultiSendLayer extends Thread {

   MultiSendLayer(JgroupDaemon daemon) {                              /* Constructor */
      seq = (0, ..., 0);                                              /* Sequence number array */
      ack = ( (0, ..., 0), ... , (0, ..., 0) );                       /* Acknowledged messages */
      msg = ( (∅, ..., ∅), ... , (∅, ..., ∅) );                       /* Messages to deliver */
      fd = (∅, ..., ∅);                                               /* Failure Detector readings */
      reachable = {p};                      /* Reachable objects (directly or indirectly) */
      this.daemon = daemon;                 /* Daemon reference used to notify upcalls) */
      this.run();                                                     /* Start the thread */
   }

   void run() {
      while true {
         wait − for event;                  /* Remain idle until some event occurs */
         case event of {
            tick:
               fd[p] = D_p;                 /* Read the local failure detector output */
               foreach (q ∈ Π) do send(⟨msg, ack, fd⟩, q);
               break;
            recv(⟨M, A, F⟩):
               foreach (q ∈ Π − {p} and upToDate(M[q], A[q], F[q])) do {
                  msg[q] = M[q]; ack[q] = A[q]; fd[q] = F[q];
                  if ((SUSPECT, n, G) ∈ M[q][p] and q ∈ reachable) {
                     ack[p][q] = n;
                     daemon.msuspect(Π − (reachable − G));
                     daemon.msuspect(Π − reachable);
                  }
                  foreach ((MSEND, m, n) ∈ M[q][p] and n == ack[p][q] + 1 ) do {
                     ack[p][q] = n;
                     daemon.mrecv(m, q);
                  }
                  msg[p][q] = msg[p][q] − { (MSEND, −, n) | n < ack[p][q] };
               }
               P = {p}; P′ = ∅;        /* Compute the transitive closure of reachability */
               while (P ≠ P′)
                  { P′ = P; P = ⋃_{q∈P′} (Π − fd[q]); }
               foreach (q ∉ P and (SUSPECT, −, −) ∉ msg[p][q]) do {
                  seq[q] = seq[q] + 1;
                  msg[p][q] = { (SUSPECT, seq[q], P) };
               }
               if (P ≠ reachable)
                  daemon.msuspect(Π − P);
               reachable = P;
               break;
         }
      }
   }
```

Figure 5.4: Algorithm for MSL using failure detector $\mathcal{D} \in \diamond\tilde{\mathcal{P}}$ for server p: Part (a).

At each server p, MSL maintains a local state defined by the variables *reachable*, *seq*, *fd*, *ack* and *msg*. *reachable* is the set of servers that are believed to be reachable through direct or indirect paths. This set is constructed from the outputs of the failure detector modules, including remote ones, as they are learned through incoming messages. *seq* is a vector indexed by Π, where $seq[q]$ represents the number of messages m-sent by p to server q; it is used to generate the sequence numbers associated with messages. *fd*, *ack* and *msg* are three vectors indexed by Π; for every server $q \in \Pi$, $fd[q]$ is a set of servers, $ack[q]$ is a vector (indexed by Π) of sequence numbers, while $msg[q]$ is a vector (indexed by Π) of sets of messages. The variables *fd*, *ack* and *msg* may be seen as partitioned in two sections: the local section refers to server p ($fd[p]$, $ack[p]$ and $msg[p]$); the remote section refers to all other servers ($fd[q]$, $ack[q]$ and $msg[q]$, for each $q \neq p$). Process p modifies its local section in response to local events (for example, a *msend* request from PGMS, the receipt of a message from the network, or a change in the output of the local failure detector module). In particular, $fd[p]$ records p's last reading of the failure detector; $ack[p][q]$ is the sequence number of the last message from q m-received by p; $msg[p][q]$ is the set of messages m-sent by p to q for which p has not yet m-received an acknowledgment. For each server $q \neq p$, $fd[q]$, $ack[q]$ and $msg[q]$ contain p's local perception of the corresponding variables of q.

The algorithm is driven by *recv*() events from below (network), *msend* primitives invoked from above (PGML) and local *tick* events that are produced periodically. It performs *mrecv* and *msuspect* upcalls on the above layer (PGML). At each *tick* event, server p reads the output of the local failure detector module as $\mathcal{D}_p$ and sends a message containing the value of *msg*, *ack* and *fd* to all servers, including itself. As explained in the next paragraph, the local copy of the message will cause the re-computation of the transitive closure of the individual reachability sets.

When p receives a message with contents $\langle M, A, F \rangle$, it verifies whether, for each server q, the data regarding q contained in the message are more recent than the ones saved in the local variables $msg[q]$, $ack[q]$ and $fd[q]$. The check is done through the predefined function *upToDate*(). If so, p updates its local variables. If the set of messages sent by q to p as contained in M includes tuples of the form $(\textsc{suspect}, n, G)$ that have not yet been handled, then p temporary m-suspects q and the servers in G by generating two *msuspect* events, the first one m-suspecting the servers in G and the second one immediately removing the servers in G from the list of m-suspected servers. Then, p copies the value n in $ack[p][q]$,

```
    void msend(m, dests) {
        foreach (q ∈ (dests ∩ reachable) − {p}) do {
            seq[q] = seq[q] + 1;
            msg[p][q] = msg[p][q] ∪ {(MSEND, m, seq[q]) };
        }
        if (p ∈ dests)
            daemon.mrecv(m, p);
    }

}                                                   /* End MultiSendLayer */
```

Figure 5.5: Algorithm for implementing MSL using failure detector $\mathcal{D} \in \diamond\tilde{\mathcal{P}}$ for server p: Part (b).

to indicate that the messages m-sent by q before m-suspecting p can be discarded. The SUSPECT tuples addressed to p by q are created to avoid persistent asymmetric scenarios. The set G is chosen so that it guarantees Property MS1(a). Then, p m-receives (in FIFO order) those messages that it has not yet done so. Finally, p modifies $msg[p]$ by removing all messages that have been acknowledged by q. At this point, p computes the transitive closure of the individual reachability sets. The resulting set P contains all the servers reachable from p through a direct or an indirect path. Finally, p creates the SUSPECT tuples needed to guarantee the eventual symmetry of function $\mathcal{R}$. If P differs from *reachable*, p generates the corresponding *msuspect* event and updates the set *reachable*.

When the above layer requests to m-send a message m to destination set G through method *msend*, the tuple (MSEND, m, n) is inserted into $msg[p][q]$ for each q in the destination set G, so that it will be sent to all servers at the next *tick* event. The value n represents the current sequence number. If p belongs to the destination set, then m is locally m-received. The MSEND tag is used to distinguish between messages sent on behalf of the upper layer and those sent internally which have SUSPECT tags.

5.4 The Group Membership Layer (GML)

In this section we present an algorithm that implements the PGMS specification described in Section 4.2 using the services provided by MSL in order to construct and install views. At each server, let the *reachable set* correspond to those servers that are not currently m-suspected. These reachable sets form a good basis for constructing views since part of the PGMS specification follows immediately from the properties of MSL that produce them. In particular, Property GM3 is satisfied by Property MS1(a) requiring that if a server q is continuously unreachable from p, then eventually p will continuously m-suspect q. Property GM2 is satisfied by Property MS1(b) and Property MS1(f), as discussed above. Finally, Property GM6 is satisfied by Property MS1(f).

The main difference between reachable sets as constructed by MSL and views as defined by PGML is with respect to coherency. While reachable sets are completely individualistic and lack any coordination, views of different servers need to be coherent among themselves as defined by Property GM4. PGML achieves this property by using reachable sets as initial estimates for new views but installs them only after having reached agreement on their composition among mutually-reachable servers. To guarantee liveness of our solution, each execution of the agreement algorithm must terminate by actually installing a new view. Yet the composition of installed views cannot invalidate any of the properties that are inherited from MSL as described above.

The algorithm for PGML, illustrated in Figures 5.6–5.9, alternates between an *idle phase*, a *synchronization phase* and an *estimate exchange phase*. Synchronization and estimate exchange phases are abbreviated in *s-phase* and *ee-phase*, respectively. S-phase and ee-phase are collectively indicate as *agreement phase*. In each of these phases, servers react to the same events in different manners. For example, a *msuspect* upcall received in s-phase causes a server to m-send a synchronization message and to move to s-phase, while a *msuspect* upcall received in ee-phase causes a server to m-send an estimate message and move to ee-phase. Instead of having complex methods whose behavior depends on the current phase, we have separated the code related to each of the phases. Figure 5.6 illustrates the idle phase; Figure 5.7 describes the s-phase; the ee-phase is contained in Figures 5.8 and 5.9.

A server remains idle until either it is informed by MSL that there is a change in its perception of the reachable set (through a *msuspect*() event), or it m-receives a message from another server that has observed such a change. Both of these events cause the server to enter s-phase.

At the beginning of s-phase, each server m-sends a synchronization message containing a version number to those servers it perceives as being reachable, and then waits for responses. This message acts to “wake-up” servers that have not yet entered s-phase. Furthermore, version numbers exchanged in the s-phase are used in the subsequent ee-phase to distinguish between messages of different agreement protocol invocations. A server leaves s-phase to enter ee-phase either when it m-receives a response to its synchronization message from every server that has not been m-suspected during the s-phase, or when it m-receives a message from a server that has already entered ee-phase.

Each server enters ee-phase (Figures 5.8 and 5.9) with its own estimate for the composition of the next view. During this phase, a server can modify its estimate to reflect changes in the approximation for reachability that is being reported to it by MSL. In order to guarantee liveness, the algorithm constructs estimate sets that are always monotone decreasing so that the agreement condition is certain to hold eventually. Whenever the estimate changes, the server m-sends a message containing the new estimate to every server belonging to the estimate itself. When a server m-receives estimate messages, it removes from its own estimate those servers that are excluded from the estimate of the sender. At the same time, each change in the estimate causes a server to m-send an agreement proposal to a server selected among the current estimate to act as a *coordinator*. Note that while estimates are evolving, different servers may select different coordinators. Or, the coordinator may crash or become unreachable before the agreement condition has been verified. In all these situations, the current agreement attempt will fail and new estimates will evolve causing a new coordinator to be selected.

When the coordinator eventually observes that proposals m-received from some set S of servers are all equal to S, agreement is achieved and the coordinator m-sends to the members of S a message containing a new view identifier and composition equal to S. When a server m-receives such a message, there are two possibilities: it can either install a *complete* view, containing all servers indicated in the message, or it can install a *partial* view, containing a subset of the servers indicated in the message. Partial views are necessary whenever the installation of a complete view would violate Property GM4(iii). This condition is checked by verifying whether the current views of servers composing the new view intersect[2]. If they do, this could mean that a server in the intersection has never installed one of the intersecting views, thus violating Property GM4(iii). For this reason, the m-received view is broken in to a set of non-intersecting partial views,

[2]This condition can be checked locally since current views of servers composing the new view are included in the message from the coordinator.

```
class JgroupDaemon {

    JgroupDaemon(Member member) {                               /* Constructor */
        reachable = {p};                              /* Set of unsuspected objects */
        version = (0, . . . , 0);                                   /* Vector clock */
        symset = ({p}, . . . , {p});                                /* Symmetry set */
        view = ( UniqueID(), {p} );               /* Current view id and composition */
        cview = view;                               /* Corresponding complete view */
        this.member = member;               /* Member reference used to notify upcalls */
        member.vchg(view);
    }

    void msuspect(MemberId[] P) {
        foreach (r ∈ (Π − P) − reachable) do {
            symset[r] = reachable;
        msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable);
        reachable = Π − P;
        synchronizationPhaseStart();
    }

    void mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, MemberId q) {
        if (version[q] < V[q]) {
            version[q] = V[q];
            if (q ∈ reachable)
                synchronizationPhaseStart();
        }
    }
```

Figure 5.6: Idle phase for server p.

each of them satisfying Property GM4(iii). If, on the other hand, current views do not intersect, each server can install the new complete view as m-received from the coordinator. Note that classification of views as being complete or partial is completely internal to the implementation. An application programmer using the provided service is unaware of the distinction and deals with a single notion of view. Although each invocation of the agreement protocol terminates with the installation of a new view, it is possible that the new view does not correspond to the current set of servers perceived as being reachable, or that a new synchronization message has been m-received during the previous ee-phase. In both cases a new agreement phase is started.

In Appendix A, we give a proof that our algorithm satisfies the properties of PGMS. Here, we discuss in high-level terms the techniques used by the algorithm in order to satisfy the specification. Leaving out the more trivial properties such as View Completeness,

```
void synchronizationPhaseStart() {
    estimate = reachable;                              /* Next view estimation */
    synchronized = {p};                                /* Synchronized servers */
    version[p] = version[p] + 1;                       /* Generate new version number */
    foreach (r ∈ estimate − {p}) do
        msend(⟨SYNCHRONIZE, version[r], version[p], symset[r]⟩, {r});
}

void msuspect(MemberId[] P) {
    foreach (r ∈ (Π − P) − reachable) do symset[r] = reachable;
    msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable);
    reachable = Π − P;
    estimate = estimate ∩ reachable;
    checkSynchronization();
}

void mrecv(⟨SYMMETRY, V, P⟩, MemberId q) {
    if (version[p] == V[p] and q ∈ estimate)
        estimate = estimate − P;
    checkSynchronization();
}

void mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, MemberId q) {
    if (version[p] == V_p)
        synchronized = synchronized ∪ {q};
    if (version[q] < V_q) {
        version[q] = V_q;
        agreed[q] = V_q;
        msend(⟨SYNCHRONIZE, version[q], version[p], symset[q]⟩, {q});
    }
    checkSynchronization();
}

void mrecv(⟨ESTIMATE, V, P⟩, MemberId q) {
    version[q] = V[q]
    if (q ∉ estimate)
        msend(⟨SYMMETRY, version, estimate⟩, {q});
    elseif (version[p] == V[p] and p ∈ P)
        estimate = estimate ∩ P;
        synchronized = P;
        agreed = V;
    }
    checkSynchronization();
}

void checkSynchronization() {
    if (estimate ⊆ synchronized) estimateExchangePhaseStart();
}
```

Figure 5.7: Synchronization phase for server p.

View Integrity and View Order, we focus our attention on View Coherency, View Accuracy and liveness of the solution. Property GM4 consists of three parts. Property GM4(iii) is satisfied through the installation of partial views, as explained above. As for Property GM4(i) and Property GM4(ii), each server is eventually informed if another server belonging to its current view has not installed it, or if it has changed views after having installed it, respectively. The server is kept informed either through a message, or through an m-suspect event. In both cases, it starts a new agreement phase. As for liveness, each invocation of the agreement protocol terminates with the installation of a new view, even in situations where the reachability relation is highly unstable. This is guaranteed by the fact that successive estimates of each server for the composition of the next view are monotone decreasing sets. This is achieved through two actions. First, new m-suspect lists reported by MSL never result in a server being added to the initial estimate. Second, servers exchange their estimates with each other and remove those servers that have been removed by others. In this manner, each server continues to reduce its estimate until it coincides exactly with those servers that agree on the composition of the next view. In the limit, the estimate set will eventually reduce to the server itself and a singleton view will be installed. This approach may seem in conflict with View Accuracy: if server p m-receives from server r a message inviting it to remove a server q, it cannot refuse it. But if p and q are permanently reachable, non-triviality properties of MSL guarantee that after some time, r cannot remove q from its view estimate without removing p as well. So, after some time, r cannot m-send a message to p inviting it to exclude q, because p cannot belong to the current estimate of r. Furthermore, s-phase of view agreement constitutes a "barrier" against the propagation of old "remove q" messages. In this way, it is possible to show that there is a time after which all views installed by p contain q.

Variables and Messages

The local state of each server p is defined by the global variables *view*, *cview*, *reachable*, *version*, *agreed*, *estimate*, *symset*, *stable* and *ctbl*.

- Variable *view* is composed of the fields *view.id* and *view.comp* that represent, respectively, the identifier and the composition of the last view passed up to the application.

- Variable *cview* has the same fields as *view*, but contains the complete view that has been m-received with the last VIEW message. When the current view of p is a complete view, *view* and *cview* are equal.

- Variable *reachable* corresponds to the set of servers perceived to be reachable (i.e., not m-suspected by MSL). During the idle phase, *reachable* and *view.comp* coincide. A new agreement phase is started whenever *reachable* changes due to a *msuspect* event, or when *reachable* is different from the composition of the last installed view.

- Variable *version* is an array indexed by Π; for each $q \in \Pi$, $version[q]$ contains the last version number of q known by p. $version[p]$ is the current version number of p. Version numbers are generated by servers whenever they enter a new agreement phase.

- When a server enters ee-phase, it creates a new array of version numbers, called *agreed*. While *version* can continually change during an ee-phase, *agreed* is fixed and is used to identify the messages related to this particular ee-phase.

- Variable *estimate* is a set of servers and represents the proposal for the composition of the next view.

- Variable *symset* is an array indexed by Π; for each $q \in \Pi$, $symset[q]$ contains the last value stored in variable *reachable* such that $q \notin reachable$. This information, communicated by p to q through SYNCHRONIZE messages, is used by the algorithm to satisfy Property GM2.

- Variable *stable* is a boolean variable indicating whether the last installed view corresponds to the approximation of reachability supplied by the MSL, or the server has to enter agreement phase again.

- *ctbl* is the *coordinator table* and contains all information needed by a server when it assumes the role of coordinator. In particular, *ctbl* is an array of records indexed by Π, where each record contains the entries *cview*, *agreed* and *estimate*. For each q, $ctbl[q]$ represent p's perception of the value of those variables at server q.

- Variable *synchronized* is used in the s-phase and contains the set of servers from which p has m-received an answer to its synchronize request.

```
void msuspect(MemberId[] P) {
    foreach (r ∈ (Π − P) − reachable) do
        symset[r] = reachable;
    msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable);
    msend(⟨ESTIMATE, agreed, estimate⟩, (Π − P) − reachable);
    reachable = Π − P;
    if (estimate ∩ P ≠ ∅)
        sendEstimate(estimate ∩ P);
}

void mrecv(⟨SYMMETRY, V, P⟩, MemberId q) {
    if ((agreed[p] == V[p] or agreed[q] ≤ V[q]) and (q ∈ estimate))
        sendEstimate(estimate ∩ P);
}

void mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, MemberId q) {
    version[q] = V_q;
    if (agreed[q] < V_q and q ∈ estimate) {
        sendEstimate(estimate ∩ P);
}

void mrecv(⟨ESTIMATE, V, P⟩, MemberId q) {
    if (q ∈ estimate);
        if (p ∉ P and (agreed[p] == V[p] or agreed[q] ≤ V[q]))
            sendEstimate(estimate ∩ P);
        elseif (p ∈ P and (∀r ∈ estimate ∩ P : agreed[r] == V[r]))
            sendEstimate(estimate − P);
}

void mrecv(⟨PROPOSE, S⟩, MemberId q) {
    ctbl[q] = S;
    if (q ∈ estimate and checkAgreement(ctbl))
        installView(UniqueID(), ctbl);
}

void mrecv(⟨VIEW, w, C⟩, q) {
    if (C[p].cview.id == cview.id and q ∈ estimate)
        installView(w, C);
}
```

Figure 5.8: Estimate exchange phase for server p: Part (a).

All m-sent messages contain a tag indicating the message type plus other fields relevant for that type. Five message types are used by the algorithm. $\langle \text{SYNCHRONIZE}, V_p, V_q, P\rangle$ messages are used during the s-phase. Fields V_p and V_q represent the version numbers of the sender and the destination as known by the sender at the time of m-sending, and P represents the last *symset* associated with the destination server. When a server enters s-phase, it m-sends a SYNCHRONIZE message to each reachable server, that respond with another SYNCHRONIZE message. $\langle \text{SYMMETRY}, V, P\rangle$ messages are used in both s- and ee-phases to handle situations where approximations of reachability obtained by the MSL are temporarily asymmetric. Here, V is a version number array, while P is the approximation of the set of reachable servers known by p at the time of m-sending. $\langle \text{ESTIMATE}, V, P\rangle$, $\langle \text{PROPOSE}, S\rangle$ and $\langle \text{VIEW}, w, C\rangle$ messages are used during the ee-phase. A ESTIMATE message is m-sent to servers belonging to the current estimate whenever this estimate changes. Once again, V is a version number array, while P is the set of servers in the current estimate of the sender. PROPOSE messages are m-sent by the servers to the coordinator each time the next view estimate changes; the field S contains the status of the sender, that corresponds to a *ctbl* entry. Finally, VIEW messages are m-sent by the coordinator to servers when agreement is reached; field w is the the new complete view identifier, while C is the value of the coordinator table when agreement has been reached and contains the composition of the view and other information used to construct partial views (if necessary). We say that a server *accepts* an m-received message if it modifies its status according to the contents of the message. Otherwise, we say that the server *discards* the message, for example because the message is obsolete.

5.4.1 Algorithm Description

Instead of giving an exhaustive description of the entire algorithm, we illustrate some of its peculiarities in order to simplify the understanding of the overall structure. Figure 5.6 contains the idle phase of the algorithm. In the Jgroup daemon constructor, each server initializes some variables and installs a view containing only itself. In this way, every server can independently create its initial view without having to reach agreement with any other server. This view will last until the first *msuspect* upcall that is invoked. Variables *view* and *cview* are updated in order to simulate the m-receipt of a VIEW message generated by p itself. *UniqueID*() is a predefined method used to create new identifiers. After

```
    void estimateExchangePhaseStart() {
        sendEstimate(∅);
    }

    void sendEstimate(MemberId[] P) {
        estimate = estimate − P;
        msend(⟨ESTIMATE, agreed, estimate⟩, reachable − {p});
        msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate));
    }

    boolean checkAgreement(CoordinatorTable C) {
        return   (∀q ∈ C[p].estimate : C[p].estimate == C[q].estimate)
                 and (∀q, r ∈ C[p].estimate : C[p].agreed[r] == C[q].agreed[r]);
    }

    void installView(View w, CoordinatorTable C) {
        msend(⟨VIEW, w, C⟩, C[p].estimate − {p});
        if (∃q, r ∈ C[p].estimate : q ∈ C[r].cview.comp and C[q].cview.id ≠ C[r].cview.id)
            view = ( (w, view.id), { r | r ∈ C[p].estimate and C[r].cview.id == cview.id } );
        else
            view = ((w, ⊥), C[p].estimate);
        member.vchg(view);
        cview = (w, C[p].estimate)
        stable =  (view.comp == reachable) and
                  (∀q, r ∈ C[p].estimate : C[p].agreed[r] == agreed[r]);
        if (stable)
            idlePhaseStart();
        else
            synchronizationPhaseStart();
    }

}                                                        /* End JgroupDaemon */
```

Figure 5.9: Estimate exchange phase for server p: Part (b).

initialization, the server remains idle until an event occurs causing it to enter s-phase.

There are two conditions under which a server enters agreement phase: the execution of a *msuspect*(P) upcall, or the m-receipt of a SYNCHRONIZE message. The second case is straightforward: p enters agreement phase if the message is not obsolete and the sender is believed reachable (remember that a server may m-receive messages from unreachable servers due to fact that MSL guarantees only eventual, and not perpetual, symmetry between servers). As for the first case, the code associated with this upcall must be described more carefully, since it is related both to the agreement protocol termination and to View Accuracy. Furthermore, similar code is repeated in other parts of the algorithm.

First of all, p updates the *symset* array entries of all servers that have become reachable since the previous *msuspect* event and m-sends a SYMMETRY message to them. The aim of these actions is to re-establish symmetry on view estimates whenever the approximation obtained by MSL is temporarily asymmetric. To see this point, consider the following scenario. Let q be a server engaged in agreement phase, waiting for agreement to be reached with p. Suppose p temporarily m-suspects q before the agreement can be reached. As we noted in the introduction, p removes q from its view estimate and the exclusion is permanent during this agreement phase. Note that p may reinsert q into the view estimate in the next view agreement. Process q, however, cannot participate in the new agreement phase of p until it has terminated its previous agreement phase in order to prevent propagation of obsolete exclusions. Thus, q cannot install a new view, since it waits forever for the participation of p in its current agreement phase. The SYMMETRY message forces q to remove p from its view estimate and allows q to reach an agreement on a new view. Furthermore, note that each SYMMETRY message carries also a set of servers corresponding to the value of variable *reachable* of the sender. This set is necessary to satisfy Property GM2: when a server m-receives a SYMMETRY message, it excludes from its view estimate all servers believed reachable by the sender. In this manner, if p m-receives a SYMMETRY message from q, and q and r are permanently reachable, p removes r as well and it will not m-send a ESTIMATE message to r inviting it to remove q. The value *reachable* is stored in array *symset* to be used in the same way with SYNCHRONIZE messages. After these steps, p updates variable *reachable* and invokes method *synchronizationPhaseStart*().

Method *synchronizationPhaseStart*() initializes s-phase by computing the next view estimate, by generating a new version number and by initializing variable *synchronized* (which contains servers that know p's new version number) to contain only p. Finally, SYNCHRONIZE messages are m-sent announcing its new version number.

Note that if p has entered s-phase due to a SYNCHRONIZE message m from a server q, the message m-sent by p acts also as a reply to m. S-phase lasts until all servers in the view estimate have replied to the SYNCHRONIZE message of p, or when p m-receives a message from a server in its view estimate that has already entered ee-phase. This condition is checked by method *checkSynchronization*, which enters e-phase of the algorithm in the case variable *synchronized* is contained in variable *estimate*. To guarantee the termination of this phase, every time p m-suspects a server q, or m-receives a SYMMETRY message from q,

p removes q from its view estimate. This exclusion cannot be revoked during the current view. On the contrary, when p m-receives a SYNCHRONIZE message containing its last version number from a server q, it adds q to *synchronized*. In this manner, *synchronized* will eventually contain *estimate*. Another method for p to enter ee-phase is to m-receive a message $\langle \text{ESTIMATE}, V, P \rangle$ from a server q that knows p's current version number and has not be removed from p's view estimate. If so, p modifies *synchronized* and *estimate* in order to guarantee the exit from the s-phase.

Figures 5.8 and 5.9 illustrate the ee-phase algorithm. Method *sendEstimate*(P) is used to modify the view estimate and to inform the other servers of the change. At the same time, during this method p m-sends a PROPOSE message carrying its current status to a coordinator selected from its view estimate through function *Min*(). Method *estimateExchangePhaseStart* invokes method *sendEstimate*() in order to guarantee the m-sending of at least one ESTIMATE and one PROPOSE message. Function *checkAgreement*(C) is used when a server m-receives PROPOSE messages. It verifies whether the proposals stored in the coordinator table are in agreement, by checking if all of the view estimates are equal and if all servers know the same version number arrays (restricted to servers in the view estimate itself). Finally, method *installView* is used to install a new view. First of all, p forwards a VIEW message containing the new complete view identifier w and the coordinator table C to all servers with which have reached the agreement. This operation is necessary, since the coordinator may crash before m-sending the VIEW message to all its recipients. Then, p checks whether a partial view is needed or not. Suppose there exists a server q whose last view v contains a server r, but the last view installed by r is different from v. If p installed the complete view, this would violate Property GM4. For this reason, the complete view is split into a set of partial views. Each partial view is composed of the set of servers that have m-received the same previous complete view, and is identified through a pair (w, v) composed of the identifiers of the new complete view and the previous view. Note that all servers in each partial view compute, starting from C, the same identifier and composition. Otherwise, if there is no need of partial views, p will install the complete view, identified by the pair $(w, \perp)$, where $\perp$ means that w is a complete view.

During the ee-phase, servers exchanges ESTIMATE messages to promote the reaching of an agreement. A message $\langle \text{ESTIMATE}, V, P \rangle$ is accepted if, for each r in the intersection between the estimate of p and the estimate of q, the version number *agreed*$[r]$ known by p is equal to $V[r]$ known by q. This check is needed to guarantee that p never accepts a ESTIMATE message containing an estimate generated before the start of the previous s-phase that could cause exclusion of reachable servers. If the message is accepted, p invokes method *sendEstimate*(), removing servers not in P. Note that in case $p \notin P$, the ESTIMATE message is interpreted as a SYMMETRY message. Apart from ESTIMATE and SYMMETRY messages and direct *msuspect*() events, during ee-phase a server p may exclude a server from its view estimate also through SYNCHRONIZE messages. When a server m-receives a message $\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle$ from a server q such that V_q is greater than *agreed*$[q]$, p is informed that q has completed the agreement phase identified by *agreed*$[q]$ and has installed a new view. There are two possibilities: p does not belong to the view installed by q, and thus p can exclude q from its view estimate; or p belongs to the view installed by q, but the VIEW message m-sent by q to p has been lost (by the FIFO order condition of MSL). This is another good reason for p to exclude q from its view estimate. In order to not violate View Accuracy, p removes all servers contained in P.

5.4.2 Discussion

The main property of our algorithm is that the agreement protocol is guaranteed to terminate at correct servers, independent of the failure scenario. In particular, the algorithm has been designed to tolerate any number of failures and repairs that occur before and *during* the execution of the agreement protocol. This capability, however, implies degraded performance in case of complex failure scenarios. To illustrate the point, recall that servers can be added to a view estimate only at the beginning of an agreement phase. This implies that when a set of servers initiate agreement towards a new view, this view may be already obsolete since some servers may cease to be m-suspected in the meantime. This in turn implies that a new agreement phase has to be started immediately after the termination of the current agreement one. Furthermore, the designated coordinator may change several times during an agreement due to some server m-suspecting the coordinator. Every time the coordinator changes, each server has to m-send its proposal to the new coordinator and wait for a reply. Finally, the installation of partial views, needed to satisfy Property GM4, may cause further invocations of the agreement protocol.

Although many optimizations are possible (for example, we could add a sub-protocol to handle the installation of a complete view after the installations of partial views without exiting ee-phase and thus saving the costs of repeated executions of the agreement protocol), we argue that in common failure scenarios the performance of our algorithm is reasonable. In particular, consider the scenario in which a single server q crashes. Eventually, every server will m-suspect q and enter s-phase. At the beginning of s-phase, each server m-sends a SYNCHRONIZE message and waits for a reply. Then, each server m-sends a proposal to the designated coordinator, and a ESTIMATE messages to every other server. In this scenario, ESTIMATE messages m-received from others will be ignored since all servers have the same view estimate (current view minus q, the crashed server). When the coordinator observes agreement on the m-received proposals, it m-sends a VIEW message to all. Hence, four end-to-end message delays are in general sufficient to exclude a crashed server from a view. The same delay analysis also characterizes partition merges or partitionings that result in the formation of disjoint partitions.

5.5 The Reliable Multicast Layer

The task of the *reliable multicast layer (RML)* is to provide daemons with primitives for multicasting messages following the *view synchrony* semantics. These primitives are used by the group method invocation layer and the state merging layer.

5.5.1 Specification

Group members communicate through reliable multicasts by invoking primitive $\mathsf{mcast}(m)$ that attempts to deliver message m to each of the servers in the current view. In the global history, the invocation of primitive $\mathsf{mcast}(m)$ corresponds to event $mcast(m)$. Multicast messages are labeled in order to be globally unique. We denote the identifier of message m with $m.id$. RML delivers messages by invoking the upcall $\mathsf{dlvr}(q, m)$, where q is the sender of multicast message m. In the global history, the execution of upcall $\mathsf{dlvr}(q, m)$ corresponds to events $dlvr(q, m)$ and $\overline{dlvr}(q, m, r)$. Servers delivering a message have the possibility to respond with a special response message addressed uniquely to the sender. Response messages are sent by returning them at the end of the dlvr upcall. In the global history, returning a response r at the end of the dlvr upcall for a message m multicast by server q corresponds to event $\overline{dlvr}(q, m, r)$. Responses are delivered to servers through the

$\mathsf{response}(q, m, r)$ upcall, where q is the sender of the response, m is the identifier of the original message and r is the response. In the global history, such upcall corresponds to events $\mathit{response}(q, m, r)$ (and $\overline{\mathit{response}}(q, m, r, -)$, which is not used in the specification).

To simplify the presentation, we use $msgs(p, v)$ to denote the set of messages that have been delivered by server p in view v; formally,

$$msgs(p,v) = \{\ m \mid \exists q, \exists t : \sigma(p,t) = dlvr(q,m) \wedge \mathsf{View}(p,t) = v\ \}.$$

The RML specification is the message-based counterpart of the GMIS specification for internal method invocations. For this reason, many of the properties describing RML have a correspondent property in the GMIS specification. For example, the following properties corresponds to Properties GMI2 and GMI3 specified in Section 4.3.1.

RM1 (Liveness)
Let p be a correct server that delivers message m in view v that includes some other server q. If q never delivers m, then p will eventually install a new view w as the immediate successor to v. Formally,

$$p \in correct(C) \wedge \sigma(p,t) = dlvr(m) \wedge q \in v.comp \wedge view(p,t) = v \wedge dlvr(m) \notin \sigma(q, \mathcal{T})$$
$$\Rightarrow\ \exists t' > t : \sigma(p,t') = vchg(w)$$

RM2 (Message Agreement) *Given two views v and w such that w is an immediate successor of v, all servers belonging to both views deliver the same set of multicast messages in view v. Formally,*

$$v \prec_p w \wedge q \in v.comp \cap w.comp\ \Rightarrow\ msgs(p,v) = msgs(q,w).$$

The following property is the message-based counterpart of Property IGMI4:

RM3 (Uniqueness) *Each multicast message, if delivered at all, is delivered in exactly one view. Formally,*

$$m \in msgs(p,v) \wedge m \in msgs(q,w)\ \Rightarrow\ v = w.$$

The next property places simple integrity requirements on delivery of messages to prevent a message from being delivered “out of thin air” without first being multicast. It partially corresponds to Property GMI1.

RM4 (Message Integrity) *Each server delivers a message only if some server actually multicast it earlier. Formally,*

$$\sigma(p,t) = dlvr(m) \;\Rightarrow\; \exists q, \exists t' < t : \sigma(q,t') = mcast(m).$$

RML satisfies also Property GMI4, requiring that two views merging into a common view must have disjoint compositions, and Properties GMI5 and GMI6, which specify the behavior of prepare upcall. Note that all these properties do not involve method invocations in their definition, and so remain valid without modification also for message multicasts.

Note that a specification consisting of these properties alone can be trivially satisfied by not delivering any messages at all. We exclude such useless solutions by including the following property.

RM5 (Multicast Termination)

A correct server always delivers its own multicast messages. If a message is multicast in a view v before the $\overline{\mathsf{prepare}}(v)$ event, the message will be delivered in v; otherwise, if it is multicast after the $\overline{\mathsf{prepare}}(v)$ event, it will terminate in the successor view. Formally,

(i) $\sigma(p,t) = mcast(m) \wedge p \in correct(C) \wedge \mathsf{View}(p,t) = v \wedge \overline{\mathsf{prepare}}(v) \notin \sigma(p,[0,t[) \;\Rightarrow$
$\exists R, \exists t' > t : \sigma(p,t') = dlvr(m) \wedge \mathsf{View}(p,t') = v$

(ii) $\sigma(p,t) = mcast(m) \wedge p \in correct(C) \wedge \mathsf{View}(p,t) = v \wedge \overline{\mathsf{prepare}}(v) \in \sigma(p,[0,t[) \;\Rightarrow$
$\exists R, \exists t' > t : \sigma(p,t') = dlvr(m) \wedge \mathsf{View}(p,t') = w \wedge v \prec_p w.$

Response messages must satisfy the following integrity requirements: they must be sent only in response to regular messages, and they can be delivered only if some server has sent it earlier. Together, these properties implies that responses are delivered only by the server that has multicast the original message. Furthermore, we require that responses for a regular message m are sent and delivered in the same view in which m has been delivered. This property will be used when implementing GMIS for guaranteeing that all return values are generated and delivered by the invoker in the same view. These conditions are formalized in the following property:

RM6 (Response Integrity)

Each server p delivers a response in view v only if some server actually returned it earlier *at the end of a* dlvr *upcall performed in v. Each server returns a response for a message m only if it actually delivered m earlier in the same view. Formally,*

$$\begin{array}{ll}(i) & \sigma(p,t) = \mathit{response}(q, m.id, r) \wedge \mathsf{View}(p,t) = v \;\Rightarrow \\ & \exists t' < t : \sigma(q,t') = \overline{\mathit{dlvr}}(p,m,r) \wedge \mathsf{View}(q,t') = v \\ (ii) & \sigma(p,t) = \overline{\mathit{dlvr}}(q,m,r) \wedge \mathsf{View}(p,t) = v \;\Rightarrow \\ & \exists t' < t : \sigma(p,t') = \mathit{dlvr}(q,m) \wedge \mathsf{View}(p,t') = v.\end{array}$$

Clearly, Response Integrity could be satisfied by an implementation that do not deliver any response message. Since we cannot require that a response is eventually delivered in any case due to the possibility of failures, we simply require that a new view excluding a server is installed if the sender does not receive the response returned by that server:

RM7 (Response Liveness)
Let p be a correct server that returns a response r in view v to correct server q. If q never delivers r, then q will eventually install a new view w excluding p as the immediate successor to v. Formally,

$$p \in \mathit{correct}(C) \wedge \sigma(p,t) = \overline{\mathit{dlvr}}(q,m,r) \wedge \mathsf{View}(p,t) = v \wedge q \in v.\mathit{comp} \wedge$$
$$\mathit{response}(p, m.id, r) \notin \sigma(q, \mathcal{T}) \;\Rightarrow\; \exists t' > t, \exists w : \sigma(p,t') = \mathit{vchg}(w) \wedge q \notin w.\mathit{comp}.$$

Properties RM1–RM7, together with Properties GMI4–GMI6 define the specification of the reliable multicast service contained in Jgroup.

5.5.2 Algorithm

In this section we show how to extend our solution for PGMS to provide the reliable multicast service specified in the previous section. Both the idle and the agreement phases of the algorithm have to be modified: the idle phase must handle *mcast*() primitives, while the agreement function must verify that Property RM2 is satisfied before declaring agreement has been reached. Figures 5.10, 5.11 and 5.12 contain the changes to the original algorithm of Section 5.4. The code of Figure 5.10 contains three new methods that have to be inserted in both the idle phase algorithm (Figure 5.6) and the s-phase algorithm

```
void mcast(m) {
    r = member.dlvr(m);
    responses[m.id] = {false, ..., false};
    responses[m.id][p] = true;
    buffer = buffer ∪ {(m, p)};
    msend(⟨MULTICAST, view.id, m⟩, view.comp − {p});
}

void mrecv(⟨MULTICAST, w, m⟩, MemberId q) {
    if (view.id == w) {
        r = member.dlvr(m);
        buffer = buffer ∪ {(m, q)};
        msend(⟨RESPONSE, m.id, r⟩, q);
    };
}

void mrecv(⟨RESPONSE, m, r⟩, MemberId q) {
    if ((m, p) ∈ buffer)) {
        member.response(q, m, r);
        responses[m][q] = true;
    };
}
```

Figure 5.10: Reliable multicast service extension to idle- and s-phase for server p.

(Figure 5.7). The code of Figure 5.11 contains four new methods that have to be added to the ee-phase algorithm (Figure 5.8). Finally, the code in Figure 5.12 substitutes the corresponding methods of Figure 5.9 used during ee-phase.

We introduce three new global variables that are used to implement view synchrony.

- Variable *buffer* contains a set of pairs (m, q), where m is a message multicast by q and delivered by p during the current view.

- Variable *responses* is a bidimensional array of booleans index by message identifiers and by server identifiers, where $responses[m][q]$ is true if and only if a response from server q for message m has been obtained.

- Variable *suspended* contains a set of triple (v, m, q), where m is a message multicast by q whose delivery cannot be performed in the current view, but has to be postponed to view v in order to satisfy Property RM2. If v is equal to the special value $\perp$, then m has to be postponed to the next view.

Furthermore, three new message types are needed:

```
void mcast(m) {
    suspended = suspended ∪ {(⊥, m, p)};
}

void mrecv(⟨MULTICAST, w, m⟩, q) {
    suspended = suspended ∪ {(w, m, q)};
}

void mrecv(⟨DELIVERED, w, B⟩, q) {
    if (view.id = w and q ∈ estimate) {
        foreach((m, q') ∈ B − buffer) do {
            r = dlvr(m);
            msend(⟨RESPONSE, m.id, r⟩, q');
        }
        msend(⟨DELIVERED, view.id, B − buffer⟩, estimate);
        buffer = buffer ∪ B;
        if (∀m, ∀q : (m, p) ∈ buffer and q ∈ estimate and responses[m][q])
            msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate));
    }
}

void mrecv(⟨RESPONSE, m, r⟩, MemberId q) {
    if ((m, p) ∈ buffer)) {
        member.response(q, m, r);
        responses[m][q] = true;
        if (∀m, ∀q : (m, p) ∈ buffer and q ∈ estimate and responses[m][q])
            msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate));
    }
}
```

Figure 5.11: Reliable multicast service extension to ee-phase for server p: Part (a).

- $\langle$MULTICAST, $v, m\rangle$ messages are used to inform the other servers in the current view that m has to be delivered during view v;
- $\langle$RESPONSE, $m, r\rangle$ messages are used to transmit a response r to the sender of a multicast message identified by m;
- $\langle$DELIVERED, $B, v\rangle$ messages are used to inform that during v, the sender has delivered the set of messages contained in B.

MULTICAST messages are accepted only during idle- or s-phase, while DELIVERED messages are accepted only during an ee-phase. DELIVERED messages are needed to allow servers to reach agreement on the set of messages to be delivered during the last installed view.

Some changes are needed also for existing variables and messages. Variable *ctbl*, the coordinator table describing the current status of each server participating in an agreement is extended to contain a field called *buffer* to be used for agreement on the set of delivered messages. For the same reason, PROPOSE messages must contain the *buffer* field as well.

Whenever a server p wants to multicast a message m, it invokes the primitive $mcast(m)$. The actions of the algorithm are different depending to the current phase. Let v be the current view of server p. During idle phase and s-phase (Figure 5.10), p m-sends a $\langle \text{MULTICAST}, v, m \rangle$ message to all servers in $v.comp$ (excluding p itself); then, p locally delivers the message and adds it to *buffer*. The local response obtained when delivering the message is notified to the server by invoking upcall response. The fact that p has obtained a response for r is registered in variable *responses*. If the multicast request occurs in the ee-phase (Figure 5.11), the pair $(\perp, m, p)$ is added to *suspended*, where the place holder $\perp$ means that m has to be multicast at the beginning of the next view. We use variable *suspended* to avoid that a server delivers new messages during ee-phase after having performed the prepare upcall on servers.

The behavior of the algorithm depends on the current phase even when a server m-receives a $\langle \text{MULTICAST}, w, m \rangle$ message from a server q. Let v be the current view of p. If the message is m-received during the idle- or the s-phase (Figure 5.10), p delivers the message and adds it to *buffer* (unless w is different from the current view, in which case the message is discarded as obsolete). Furthermore, p m-sends to the invoker a RESPONSE message with the value obtained by upcall dlvr. If the message is m-received during ee-phase (Figure 5.11), the pair (w, m, q) is inserted in *suspended*, because the MULTICAST message may have been generated by q after having installed the next view.

Few changes are also necessary in other sections of the ee-phase algorithm. During the ee-phase initialization, the prepare upcall is invoked on the server; moreover, a $\langle \text{DELIVERED}, B, v \rangle$ message is m-sent to servers in the current view estimate, where B is equal to buffer. Method *sendEstimate* is modified in order to m-send a PROPOSE message only in the case that responses have been obtained for all messages multicast by p during the current view, from all servers in the current estimate. In this way, we postpone the reach of an agreement on a new view until all responses to multicast messages delivered in the current view are arrived. Method *checkAgreement*() (Figure 5.12) will return true only if all servers that will survive from a view in the next one (partial or complete) have delivered also the same set of messages. Finally, after having installed the new view w,

```
void estimateExchangePhaseStart() {
    member.prepare(view);
    sendEstimate(∅);
    msend(⟨DELIVERED, view.id, buffer⟩, estimate);
}

void sendEstimate(MemberId[] P) {
    estimate = estimate − P;
    msend(⟨ESTIMATE, agreed, estimate⟩, reachable − {p});
    if (∀m, ∀q : (m, p) ∈ buffer and q ∈ estimate and responses[m][q])
        msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate));
}

boolean checkAgreement(CoordinatorTable C) {
    return (∀q ∈ C[p].estimate :
                    C[p].estimate == C[q].estimate and
                    C[p].buffer == C[q].buffer) and
                    (∀q, r ∈ C[p].estimate : C[p].agreed[r] == C[q].agreed[r]);
}

void installView(View w, CoordinatorTable C) {
    msend(⟨VIEW, w, C⟩, C[p].estimate − {p});
    if (∃q, r ∈ C[p].estimate : q ∈ C[r].cview.comp and C[q].cview.id ≠ C[r].cview.id)
        view = ( (w, view.id), { r | r ∈ C[p].estimate and C[r].cview.id == cview.id } );
    else
        view = ( (w, ⊥), C[p].estimate );
    member.vchg(view);
    cview = (w, C[p].estimate)
    stable = (view.comp == reachable) and
            (∀q, r ∈ C[p].estimate : C[p].agreed[r] == agreed[r]);
    buffer = {m|(⊥, m) ∈ suspended ∨ (view.id, m) ∈ suspended}
    foreach (w, m, q′) ∈ suspended do {
        r = member.dlvr(q′, m);
        msend(⟨RESPONSE, m.id, r⟩, q′);
    }
    foreach (⊥, m, p) ∈ suspended do {
        r = member.dlvr(m);
        responses[m.id] = {false, . . . , false};
        responses[m.id][p] = true;
        member.response(p, m, r);
        msend(⟨MULTICAST, view.id, m⟩, view.comp − {p})
    }
    suspended = ∅
    if (stable)
        idlePhaseStart();
    else
        synchronizationPhaseStart();
}
```

Figure 5.12: Reliable multicast service extension to ee-phase for server p: Part (b).

method *installView*() (Figure 5.12) delivers all messages whose delivery in the previous view had to be postponed.

When a RESPONSE message is m-received, we first check whether the response is associated with one of the messages delivered in the current view. If so, the response is delivered to the server through a response upcall; the corresponding boolean value in *responses* is set to true. Provided that the algorithm is in ee-phase and all responses have been delivered for each message, a PROPOSE message is m-sent to the current coordinator. In this way, we guarantee that when all responses arrive, at least one PROPOSE message is m-sent.

When a server m-receives a DELIVERED message from a server not excluded from its view estimate (Figure 5.11), it delivers all undelivered messages contained in B, adds them to *buffer* and sends the opportune responses. Furthermore, it m-sends a new $\langle \text{DELIVERED}, B', v\rangle$ message, where B' contains the subset of messages not yet delivered before the receiving of B. This additional message is necessary since p could accept DELIVERED messages from servers included in its current view estimate, but excluded from the estimate of other servers participating in the agreement. Finally, server p m-sends a PROPOSE message to the current coordinator.

Property RM2 is guaranteed by the fact that during ee-phase, each server can deliver only messages previously delivered by servers in its estimate. Moreover, agreement on a new view may be observed only when the proposals m-sent by servers surviving from a view to the same successor view contain the same messages. Property RM3 is guaranteed by the fact that each message is associated with the view in which it must be delivered. Property GMI4 is guaranteed by the creation of partial views. Property RM4 is straightforward, while Property RM1 follows from the liveness condition of MSL and from Property GM4. Once again, the aim of this algorithm is to show the implementability of our specifications. Many possible optimizations have been neglected for sake of simplicity. For example, acknowledgements can be piggybacked in *mcast* messages to decrease the size of *buffer*.

5.6 The Daemon Interaction Layer

The daemon interaction layer is the basic building block of every group manager. Its task is to manage the interactions between the set of layers composing a group manager and the instance of Jgroup daemon running in a certain JVM. The daemon interaction layer

```
class DILGroupManager extends Thread implements GroupManager {

    DILGroupManager(Member member) {                          /* Constructor */
        queue = ∅;                                            /* Upcall queue */
        this.member = member;                                 /* Member reference */
        daemon = getDaemon(this);                             /* Obtain a group manager */
        this.run();
    }

    on invocation of primitive m(Object[] args) by application server {
        daemon.invoke(m, args);
    }

    Object upcallNotify(String name, Object[] args) {
        queue = queue ·(name, args);
        notify();
    }

    void run() {
        while (true) {
            while (queue . isEmpty())
                wait();
            (name, args) = queue . getFirst();
            r = member.invoke(name, args);
            daemon.returnValue(name, args, r);
        }
    }
}
```

Figure 5.13: The daemon interaction layer for server p.

accepts join, leave and mcast requests coming from the upper layer and forwards them to the Jgroup daemon, and takes care of notifying the upper layer of basic group communication upcalls generated by the Jgroup daemon, such as view changes and message deliveries.

A sketch of the algorithm implementing the daemon interaction layer for server p is shown in Figure 5.13. Class DILGroupManager extends Thread, and so it runs its own thread of control. When the layer is initialized, an empty queue of upcall events is created; furthermore, p stores a reference for the Jgroup daemon in variable *daemon* and a reference for an upcall event listener (which must implement interface Member) in variable *member*. The Jgroup daemon reference is obtained through the predefined method *getDaemon*, while the member reference is obtained as an argument of the constructor.

Whenever an application server invokes one of the basic group communication primitives (join, leave and mcast), the daemon interaction layer forwards it immediately to the daemon. On the contrary, upcalls generated by the Jgroup daemon, addressed to the upper layer and notified through *upcallNotify* are inserted in the upcall queue. The `run()` method of the thread consumes upcalls stored in the queue and notifies the upper layer of them. In this way, each server is associated with exactly one thread for each of the group it has joined, which actually executes the upcall generated by the Jgroup daemon. Return values possibly generated by event notification are returned to the Jgroup daemon through method *returnValue*.

5.7 The Group Method Invocation Layer and Stubs

In this section, we show the algorithms of the group method invocation layer and of stubs implementing the GMIS specification described in Section 4.3. The task of these algorithms is to transform the message-based reliable multicast service offered by RML in the corresponding invocation-based service.

As discussed in Chapter 3, clients and servers need stubs and group managers to perform external and internal group method invocations. Stubs implement the external remote interfaces for the object groups they represent, while group managers implement the multi-response interfaces derived from internal remote interfaces. Multi-response interfaces, stubs and group managers are generated by the `gmic` compiler.

Stubs and group managers implement the group method invocation service. Clients interact only with stubs (as in standard Java RMI), while servers interact only with group managers. External group method invocations are transmitted by stubs to one of the group managers, which takes care of dispatching the invocation (in case of anycast semantics), or multicasting the invocation to group managers associated with the servers included in the current view. Internal group method invocations are directly multicast by the group manager associated with the invoker. When a group manager delivers an invocation, it invokes the corresponding method on its associated server and sends back the return value to the invoker.

General schemas for the algorithms implementing stubs and group managers are contained in Figures 5.14, 5.15 and 5.16. In the next sections, we provide a detailed description of these algorithms.

```
class GroupStub extends RemoteStub {

    GroupStub(GroupRef gref) {                                /* Constructor */
        counter = 0;                                          /* Invocation counter */
        acknum = 0;                                           /* Acknowledgement number */
        ackset = ∅;                                           /* Acknowledgement set */
        incarnation = getIncarnation();                       /* Incarnation number */
        this.gref = gref;                                     /* Group reference */
    }

    on invocation of method m(args) {
        completed = false;
        counter = counter + 1;
        iid = (p, incarnation, counter);
        ref = gref.getFirst();
        while (not completed and ref ≠ null) {
            try {
                ref.extInvoke(m, args, iid, acknum, isMulticast(m));
            } catch (Exception e) {
            }
            ref = gref.getNext();
        }
        ackset = ackset ∪ {counter};
        acknum = Max { v | ∀v' : v' ≤ v  ⇒  v ∈ ackset };
        ackset = ackset − { v | v ≤ acknum };
    }

}                                                             /* End GroupStub */
```

Figure 5.14: The generic stub algorithm for client c.

5.7.1 General Schema for Stub Algorithms

The task of a stub is to transmit method invocations to at least one of the group managers associated with the servers composing the group (provided that at least one correct and reachable group manager exists). Object groups are located by a stub through a *group reference*, which contains a standard Java RMI remote reference for each of the group managers of servers forming a group. When designing group references, two different approaches may be adopted:

- *Static*: the composition of group reference is fixed and does not take into consideration variations in the group composition given by voluntary requests to join or leave a group, or to involuntary events like server crashes or partitionings.

- *Dynamic*: group references are kept continuously informed about variations in the membership of the group through the installation of views. In this way, method invocations issued by clients may be directly multicast to all servers included in the current view.

Unfortunately, dynamic group references are too costly for environments with a potentially large number of stubs that need to be continuously updated with the current view composition. For this reason, the approach adopted by Jgroup is substantially static. Stubs are obtained by invoking method lookup on the dependable registry service. The group reference known to the stub is an approximation of the group's membership as known by the registry at the time of the *lookup* invocation. Method invocations are transmitted by stubs to one of the group managers associated with the group, using the standard Java RMI protocol. The group manager receiving the invocation takes care of multicasting the invocation to the actual group members. Clearly, this choice may cause references to become stale, particularly in the case of highly dynamic, short-lived server groups. For this reason, when an external invocation on a group fails because all servers known to the stub have since left the group, the stubs contact the registry in order to be updated on the membership of the group.

The algorithm implementing a generic stub for an object group is shown in Figure 5.14. When a method is invoked, a stub selects one of the group managers associated with the group and tries to transmit the invocation to it. Remote references for group managers are maintained in variable *gref*. If the invocation succeeds, its return value is sent back to the invoking client. If a remote exception is thrown, the contacted group manager is suspected as unreachable and a new group manager is selected. This process continues until a correct and reachable group manager is found, or all group managers are suspected. In the latter case, a remote exception is thrown, in order to notify that the requested service cannot be accessed.

It is important to note that a single group manager is contacted, even if the invocation is based on the multicast semantics. The group manager receiving the invocation multicasts the invocation to all members of its current view. As Jgroup is based on static group references, the actual composition of a group may be different from those contained in group references maintained by stubs. In other words, a direct multicasting of the invocation to the group managers associated with the group would be useless: in any case, an additional communication step among servers is needed, in order to transmit the invocation to servers not included in the group reference and to guarantee view synchrony. The function *isMulticast*() is used to distinguish between invocations with anycast semantics and those with multicast semantics; such information is sent by stubs to group managers.

Stubs are also responsible for the creation of invocation identifiers (IID). Invocation identifiers are used by group managers to identify and discard duplicated executions of the same method on the same server, and may be used by application servers to identify invocations executed by different servers. Each invocation identifier is composed of the identifier of the client (the IP address of the machine hosting the JVM), an *incarnation number* (used to distinguish different virtual machines residing on the same host) and an invocation counter (incremented at each invocation). The incarnation number is maintained in variable *incarnation*, while the invocation counter is maintained in variable *counter*. Invocation identifiers are transmitted together with the method name and the invocation arguments.

As explained in the next section, each group manager is required to maintain information about the method invocations it has executed and the return values obtained by them. This information has to be maintained until the group manager knows that the stub has completed the invocation and a value has been returned to the client. Stubs inform group managers about which invocation have been completed by adding an acknowledgement counter to each invocation transmitted, maintained in variable *acknum*. The ack counter is the maximum integer k such that all invocations with counter less or equal than k have been completed. The ack counter is associated with an acknowledgement set *ackset*, which contains all counters of invocations that have been completed and are greater than the ack counter. The acknowledgement set is needed because several group method invocations may be issued in parallel, as the client may be multithreaded. Whenever an invocation is completed, its counter is inserted in *ackset*. At this point, the stub checks if the ack counter may be incremented. If so, variable *acknum* is updated and the ack set is pruned by removing all counters less or equal than *acknum*.

5.7.2 General Schema for Group Method Invocation Layers Algorithms

The task of a group method invocation layer of a group manager is to multicast and dispatch internal invocations originating from servers and external invocations issued by clients. GMIL uses the reliable multicast primitives of RML to communicate with other group managers. When a server issues an internal invocation, the group manager multicast a message containing the method signature, the invocation arguments and an invocation identifier to all servers in the current view. When an external invocation performed by a client is received through a standard RMI invocation, the group manager locally executes it (if the invocation is based on the anycast semantics), or multicasts it to all servers in its current view (if the invocation has multicast semantics).

The algorithm for the group method invocation layer is shown in Figures 5.15 and 5.16. As introduced in Section 5.1, class GMILGroupManager implements both GroupManager and Member. When instantiated, the layer stores a reference for the lower layer in variable *group*, and a reference for the upper layer in variable *member*. Variable *member* is instantiated using the parameter passed with the constructor, while *group* is obtained through method getGroupManager, which returns an opportune instance of group manager, depending on the interfaces implemented by *member*.

Apart from this one, the group method invocation maintains also the following variables:

- Variable *vcomp* maintains the composition of the current view and is updated at every view installation.
- Variable *counter* and *incarnation* have the same meaning as in stubs, and are used to generate unique invocation identifiers;
- Variable *completed* is an associative array indexed by invocation identifiers, whose elements are sets of member identifiers. Variable *completed*[*iid*] contains the set of members that have already executed invocation *iid* performed by p and have returned a result.
- Variable *waiting* is an associative array indexed by invocation identifiers, whose elements are sets of member identifiers. Variable *waiting*[*iid*] contains the set of members that are expected to execute invocation *iid* during the current view. If this set is empty, the invocation has not to be completed in this view.
- Variable *callbacks* is an associative array indexed by invocation identifiers, whose elements are callback objects to be notified when the results for an asynchronous

```
class GMILGroupManager implements GroupManager, Member {

    GMILGroupManager(Member member) {                                      /* Constructor */
        vcomp = {p};                                          /* Current view composition */
        counter = 0;                                                /* Invocation counter */
        incarnation = getIncarnation();                             /* Incarnation number */
        completed = {∅, ..., ∅};                        /* Completed sets for invocation ids */
        waiting = {∅, ..., ∅};                            /* Waiting sets for invocation ids */
        callbacks = {null, ..., null};  /* Callbacks object for asyncronous invocations */
        monitor = {null, ..., null};                                   /* Monitor objects */
        intresults = {∅, ..., ∅};                   /* Result sets for internal invocation */
        extresults = {null, ..., null};               /* Results for external invocation */
        this.member = member;                                         /* Member reference */
        group = GmFactory.getGroupManager(this);                /* Obtain a group manager */
    }

    on invocation of primitive name(Object[] args) {        /* Group manager primitives */
        return group.invoke(name, args);
    }

    on invocation of method name(Object[] args) {                        /* Synch. version */
        iid = (p, incarnation, ++counter);
        completed[iid] = ∅;
        intresults[iid] = ∅;
        callbacks[iid] = null;
        group.mcast(⟨INTERNAL, name, args, iid⟩);
        monitor[iid] = getMonitor();
        monitor[iid].wait();
        R = intresults[iid];
        intresults[iid] = null;
        return R;
    }

    on invocation of method name(Object[] args, Callback cb) {          /* Asynch. version */
        iid = (p, incarnation, ++counter);
        completed[iid] = ∅;
        intresults[iid] = ∅;
        callbacks[iid] = cb;
        group.mcast(⟨INTERNAL, name, args, iid⟩);
    }

    Object dlvr(⟨INTERNAL, String name, Object[] args, IID iid⟩, MemberId q) {
        if (q == p)
            waiting[iid] = vcomp;
        result.iid = iid;
        result.value = member.invoke(name, args);
        return result;
    }
```

Figure 5.15: The generic GMI group manager algorithm for server p: Part (a).

invocation identified by *iid* are available.

- Variable *monitor* is an associative array indexed by invocation identifiers, whose elements are monitor objects used to suspend the execution until a certain condition is satisfied. Each synchronous invocation and each external invocation with multicast semantics is associated with one of this objects.

- Variables *intresults* and *extresults* are associative arrays indexed by invocation identifiers, whose elements are set of return values and single return value, respectively. These array are used to store return values until all the results for an internal invocation are available, or until a client acknowledges the receipt of a return value for an external invocation.

When an internal invocation of method m is performed, a new invocation identifier *iid* is generated. The elements of variables *completed* and *intresults* are initialized to empty, meaning that so far, no servers have completed the invocation and no results for it have been obtained. If the invocation is synchronous, variable *callbacks*[*iid*] is set to null; otherwise, the reference of the callback object passed up as an argument is stored in *callbacks*[*iid*]. Then, a message tagged INTERNAL containing the method name, the arguments and the invocation identifier is multicast to all servers in the group.

At this point, the method invocation evolves differently depending on its semantics: synchronous or asynchronous. In the latter case, the method returns immediately, and return values will be delivered to the callback object. In the former case, the method activates a monitor and stores it in *monitor*[*iid*]; then, it waits on the monitor until it is notified that all return values for the invocation are available. When this occurs, p removes the values from variable *intresults*[*iid*] and returns them to the invoker.

When the group method invocation layer at p delivers an INTERNAL message containing an invocation with identifier *iid*, it first checks whether the sender is equal to p or not. If so, it stores in variable *waiting* the set of servers that are expected to execute the invocation. By Properties GMI2 and GMI3, this set corresponds to the current view composition. Finally, the invocation is performed on the associated server, and a result object containing the invocation identifier and the return value is returned.

This result value will be delivered to the invoker through upcall response. When a return value from server q arrives, p memorizes this fact by adding q to variable *completed*[*iid*]

```
    void response(MessageId mid, MemberId q, Object result) {
        completed[result.iid] = completed[result.iid] ∪ {q};
        intresults[result.iid] = intresults[result.iid] ∪ {result.value)};
        if (waiting[result.iid] ≠ ∅ ∧ waiting[result.iid] ⊆ completed[result.iid])
            if (callbacks[result.iid] == null) {              /* Synchronous invocation */
                monitor[result.iid].notify();
            else {                                             /* Asynchronous invocation */
                callbacks[result.iid].deliverResults(intresults[result.iid]);
                intresults[iid] = null;
            }
    }

    void vchg(View v) {                                        /* From Member interface */
        vcomp = v.comp;
        foreach (iid : waiting[iid] ≠ ∅) do {
            if (callbacks[iid] == null) {                      /* Synchronous invocation */
                monitor[iid].notify();
            else                                               /* Asynchronous invocation */
                callbacks[iid].deliverResults(intresults[iid]);
        }
        member.vchg(v);
    }

    void extInvoke(String name, Object[] args, IID iid, int acknum, boolean multicast) {
        removeOldResults(extresults, iid.member, acknum);
        if (extresults[iid] == null) {
            if (multicast) {
                group.mcast(⟨EXTERNAL, name, args, iid⟩);
                monitor[iid] = getMonitor();
                monitor[iid].wait();
            } else
                extresults[iid] = member.invoke(name, args);
        return extresults[iid];
    }

    Object dlvr(⟨EXTERNAL, String name, Object[] args, IID iid⟩, MemberId q) {
        if (extresults[iid] == null) {
            extresults[iid] = member.invoke(name, args);
            if (p == q)
                monitor[iid].notify();
        }
    }

}                                                              /* End GMILGroupManager */
```

Figure 5.16: The group method invocation layer for server p: Part (b).

and by storing the result in *intresults*[*iid*]. If all expected return values are arrived (i.e., if *completed*[*iid*] contains all servers in *waiting*[*iid*]), then the results are delivered to the invoker, either by notifying the opportune monitor object activated during the synchronous invocation, or by delivering the results to the callback as an argument of the asynchronous invocation. Variable *waiting*[*iid*] is set to empty, meaning that the invocation has already been completed.

When a view change occurs, variable *vcomp* is updated to reflect the new view composition. Internal invocations that are expected to complete during the current view (i.e., such that *waiting*[*iid*] is not empty) are considered completed. In fact, Properties RM6 and RM6 guarantee that no return message may arrive after the view installation. Thus, the results contained in *intresults* are delivered to the invoker, either by notifying the opportune monitor object activated during the synchronous invocation, or by delivering the results to the callback as an argument of the asynchronous invocation. Variable *waiting*[*iid*] is set to empty, meaning that the invocation has already been completed.

When an external invocation coming from a remote stub arrives, the first action performed is to remove all obsolete values stored in *extresults*. As introduced in the previous section, return values obtained from the execution of external invocations have to be stored until GMIL knows that the stub has completed the invocation and a value has been returned to the client. Stubs inform group managers about which invocations have been completed by adding the acknowledgement counter *acknum* to each invocation transmitted. This information is used in the predefined method *removeOldResults* to substitute all the results for invocations executed by the same client, whose invocation counter is older than *acknum*, with a constant value different from null to avoid duplicated executions.

If the invocation has not been executed, there are two possibilities. If the invocation has multicast semantics (i.e., if argument *multicast* is true), then a message tagged EXTERNAL containing the method name, the arguments and the invocation identifiers is multicast to all servers in the group. Then, the group manager activates a monitor and waits until the invocation is locally executed and a return value is stored in *extresults*. If, on the contrary, the invocation has anycast semantics, the invocation is executed locally, the return value is registered in *extresults* and is returned to the invoker.

When an EXTERNAL message is delivered, the invocation is performed on the associated server (unless it has been executed earlier). If the sender of the multicast message is p

itself, the monitor associated with the invocation is notified of the completion, in order that the return value may be returned to the invoker client.

5.8 The State Merging Layer

In this section, we show an algorithm that implements the state merging service specified in Section 4.4. The task of the state merging layer is to maintain some information about the up-to-date sets of members, and to start a state merging protocol when needed. When two or more partitions merge in a new one, the state merging layer elects a coordinator among the servers in each of the merging partitions; each coordinator acts on the behalf of its partition and is responsible for selecting the information needed to update servers not included in it. The state merging service takes care of diffusing this information among the servers composing the group.

Request to join or leave a group are forwarded to the group membership layer. A view change event generated by the group membership layer is intercepted by the state merging layer; it is forwarded "as is" to the application if the member has not been selected as coordinator; on the contrary, special view change upcalls are addressed to coordinators, which requires to return an object containing the information needed to update the set of previously partitioned members. This information is notified to these members through the execution of a *mergeState* event. In other words, special view change upcalls combine standard view change upcalls defined in Section 4.2 with *getState* events defined in Section 4.4.

The algorithm is illustrated in Figure 5.18. The SMLGroupManager class implements the multi-response interface IntMergingGM illustrated in Figure 5.17, and the empty external remote interface EmptyExternal (as external group method invocations are not needed). The multi-response interface contains only the synchronous and the asynchronous version of method *update*, used by the state merging layer to diffuse information obtained from coordinators. Finally, SMLGroupManager implements also Member, used by Jgroup to notify the state merging layer of events generated by the group membership layer.

5.8.1 Variables

Every server p maintains a set of variables as follows:

```
package jgroup;

public interface Member {

  public void update(Object status, MemberId[] sources);

  public void update(Object status, MemberId[] sources, Callback cb);

}
```

Figure 5.17: The group manager remote interface generated by `gmic` from the internal interface

- *vcomp* maintains the composition of the current view and is updated at every view installation.
- *up-to-date* represents the update set of the server; i.e., a server q is contained in *up-to-date* of p if and only if p is up-to-date with respect to q.
- *mset* contains the set of servers that are up-to-date with respect to p; in other words, each server in *mset* has applied every update performed by p.
- *member* is a reference to the upper layer, i.e. to the application. It is used to perform upcalls.
- *group* is a reference to the group manager used to join the desired group.

Variables *vcomp*, *mset* and *up-to-date* are used to determine what information needs to be exchanged and who is responsible for the exchange during the reconciliation protocol started after the installation of a new view. Note that servers in $\mathit{mset} \cap \mathit{up\text{-}to\text{-}date}$ know the same set of updates as p. If a server q in the current view is not in *mset*, then q could be lagging behind with respect to servers in *mset* and should be brought up to date.

5.8.2 Algorithm Description

In the constructor, the state merging layer associated with server p initializes the variables listed in the previous section. Variables *vcomp*, *mset* and *up-to-date* are set equal to the singleton $\{p\}$, meaning that p is initially considered partitioned from every other member and is up-to-date only with respect to itself. The argument passed to the constructor is used to initialize variable *member*.

```
class SMLGroupManager implements IntMergingGM, EmptyExternal, Member {

    SMLGroupManager(MergingMember m) {                          /* Constructor */
        up-to-date = {p};                                       /* Up-to-date set of p */
        mset = {p};                           /* Servers up-ot-date with respect to p */
        vcomp = {p};                                            /* Current view composition */
        member = m;                           /* Member reference for performing upcalls */
        group = GmFactory . getGroupManager(this);              /* Obtain a group manager */
    }

    on invocation of primitive name(Object[] args) {            /* Group manager primitives */
        return group. invoke(name, args);
    }

    void vchg(View v) {
        vcomp = v.comp;
        up-to-date = up-to-date ∩ v.comp;
        mset = mset ∩ v.comp;
        if (mset ≠ vcompand Min(mset) == p) {
            status = member. vchg(v, vcomp − mset);
            group. update(status, up-to-date, null);
        } else {
            member. vchg(v);
        }
    }

    void update(Object status, MemberId[] sources, Callback cb) {
        up-to-date = up-to-date ∪ sources;
        if (p ∈ sources)
            mset = vcomp;
        member. mergeState(status, sources, mset ∩ up-to-date)
    }

    void update(Object status, MemberId[] sources) {
        /* Do nothing, as the method may be invoked only asynchronously */
    }

}
```

Figure 5.18: The SML algorithm for server p

The core of the algorithm is the code associated with the installation of new views reflecting changes in the failure scenario. When installing a new view, p verifies whether a state merging protocol is needed or not. In order to do this, variables *vcomp*, *up-to-date* and *mset* are updated. The members not included in the new view are removed from *up-to-date* and *mset*, to indicate that p cannot be up-to-date with respect to them and they cannot be up-to-date with respect to p until a state merging protocol is completed. If a member q in the current view does not belong to *mset*, q may be lagging behind and a state merging protocol should be started. Each member in a partition checks whether it has been selected as coordinator or not; this check is performed by verifying if p is the member with the minimum identifier among those contained in *mset*. If so, the server associated with p is required to provide the information needed to update members not included in *mset* (i.e., those that are lagging behind with respect to p). The information obtained from the application is communicated to the other members in the current view through the invocation of the internal method *update*.

Method *update* takes two parameters: the *status* object contains the updates obtained from the application, while *sources* contains the identifiers of the members that are in the up-to-date set of the invoker q. If a member r is in *sources*, this means that the invoker has applied every update applied by r. After the updates coming from q have been merged to the current state of the application through the invocation of *mergeState*, we can conclude that p has applied every update applied by q; for transitivity, p has applied every update applied by r and thus is up-to-date with respect to r. For this reason, the identifiers contained in *sources* are inserted in *up-to-date*. If p is contained in *sources*, this means that q has applied every update applied by p. If the *update* invocation is executed by every member in the current view, this means that every update applied by p are applied by all members of the current view. For this reason, variable *mset* is set equal to *vcomp*. Note that if some member q of the current view does not execute the *update* invocation, a new view excluding q will eventually be installed. This event causes the removal of q from *mset*.

To conclude, we give an informal proof that this algorithm satisfies the SMS specification described in Section 4.4. Property Property SMS3 is satisfied by the fact that a *mergeState* event is executed in correspondence of each execution of the *update* method; *update* is an internal method that follows the view synchrony semantics. Termination of

```
package java.rmi.server;

public interface RemoteRef
  extends java.io.Externalizable {

  Object invoke(Remote obj, java.lang.reflect.Method method,
      Object[] params, long opnum)
    throws Exception;

  RemoteCall newCall(RemoteObject obj, Operation[] op, int opnum, long hash)
    throws RemoteException;

  void invoke(RemoteCall call)
    throws Exception;

  void done(RemoteCall call)
    throws RemoteException;

/* ... */

}
```

Figure 5.19: The `java.rmi.server.RemoteRef`.

the algorithm (Property Property SMS4) is guaranteed by the monotonicity of variable *mset* maintained at each member, from which coordinators is selected: eventually, a coordinator will diffuse updates on the behalf of p, or p will be selected as coordinator and will diffuse its updates by itself. Finally, the algorithm guarantees that a state merging protocol is started only if there is a member not up-to-date with respect to some other.

5.9 Java RMI and Group Method Invocations: Implementation Issues

The current Java RMI version includes some mechanisms to enable users to extend Java RMI with new semantics for method invocations. Unfortunately, these mechanisms have proven to be not sufficient for integrating group communication with Java RMI; for these reasons, in the Jgroup implementation some workarounds were used.

As introduced in Chapter 2, interactions among clients and remote servers are based on a standard mechanism composed of stubs and skeletons. Stubs accept invocation from clients, transmit the invocation data to server-side skeletons, which execute the invocations and send back the ruturn values to stubs, which return them to clients. Each stub contains a *remote reference*, that can be seen as an handle for the remote object. This remote reference is used to carry out remote invocations on the remote object for which it is a reference. Remote references must implement interface `java.rmi.server.RemoteRef`, that is shown in Figure 5.19.

There are fundamental differences between the protocol used in JDK 1.1 and JDK 1.2 to implement remote method invocations:

- **Jdk 1.1**: When a client invokes a method on a stub, invocation data are sent to the skeleton through the support of a `RemoteCall` object generated by the remote reference. Each remote object class is associated with its own skeleton class. Objects of this class receive the invocation data, and depending on them the appropriate method is invoked.

- **Jdk 1.2**: When a client invokes a method on a stub, invocation data are sent through the remote reference directly to the RMI system on the server side. Here, using reflection, generic code takes care of dispatching the method invocation to the remote object. In other words, skeletons are not required in Jdk 1.2.

The current version of Java RMI includes only two types of remote references. Both of them provide unicast (point-to-point) invocation mechanisms, one relative to servers always running on some machine, and one relative to servers that are activated only when one of their methods is invoked. No multicast invocation mechanism is provided. In order to extend Java RMI with anycast and multicast semantics for group method invocations on object groups, new types of remote reference need to be created. These remote references are called *group references* and contain an handle for each of the remote servers composing the group. Group references are created by the dependable registry service included in Jgroup.

Although remote references was already present in Jdk 1.1, and have been designed for enabling users to extend Java RMI, a flaw in their specification actually prevented them to be extended in whatever way. The problem resides in the serialization protocol of stub objects: when a remote object deserializes its remote reference, it assumes that it belongs to the `sun.rmi.server` package, which cannot be modified by users (for example, by inserting new remote reference classes), as `sun.*` classes are internal to Jdk implementation and may be subject to changes without notice [51].

Jdk 1.2 corrected this flaw by enabling remote references to belong to any package. Unfortunately, the new version of the Jdk introduced new problems for implementing group method invocations. As we have seen in this chapter, the handles maintained by a remote reference are static, i.e. are not dynamically updated at every group composition change. This means that the information about the group composition maintained by a remote reference may be stale, as new servers could have joined the group, and old servers may have leaved. For this reason, when performing an external invocation with multicast semantics, a remote reference tries to contact at least one of the server composing the group. The invocation is then multicast to all server in the group. Clearly, the entity that multicast the invocation cannot be the server itself, as it is designed and implemented by Jgroup users. The right component for performing this operation would be the group skeleton, but skeletons have been removed from Jdk 1.2. The current version of Java RMI does not enable users to extend the generic code that accepts invocation requests from clients and dispatch them to servers, so an alternative entity for multicasting invocations should be selected.

As we have seen above, the task of multicasting invocations has been assigned to group managers. Invocation data are transmitted by group references to the group manager of one of the servers composing the group. Depending on the invocation semantics, the group manager either dispatches the method locally, or multicasts it to all servers in the group. Currently, both stubs and group managers are generated by the group method invocation compiler `gmic`. `gmic` has been obtained by opportunely modifying the source code of the `rmic` compiler included in the Java RMI distribution.

Group managers have taken the role of skeletons of Jdk 1.1: each remote server has a corresponding group manager (or more than one, if it has joined more than one group). The group manager contains code to analyze invocation data received from stubs and to invoke the opportune method. Future version will adopt the same reflection techniques used in the Jdk 1.2 Java RMI implementation to avoid generated code. Clearly, it will fundamental to avoid the error made by Java RMI designers, which have deprived users of an hook for extending Java RMI on the server side.

Chapter 6

Programming with Jgroup

The presentation of Jgroup would not be complete without a description of how it can be used to develop realistic distributed applications. In this chapter, we describe some example applications based on Jgroup. Their aim is to introduce the main characteristics of the programming style suggested by the use of Jgroup. The applications contained in this chapter include the *Jgroup registry* described in Chapter 3, which proves how to implement a replicated database with weak-consistency requirements in a partitionable system; the *partitionable execution service*, which shows how Jgroup can be used to compute time-consuming requests in parallel; and a request allocator service, which shows how an application with strong consistency requirements may be designed in order to be partition-aware.

6.1 General Techniques for Programming Partition-Aware Applications

As noted in the previous chapters, the main problem with which programmers have to confront when developing applications in partitionable systems is the possibility that the states maintained by servers belonging to different partitions evolve in an inconsistent way. When partitionings disappears, a state merging protocol is needed, in order to reconcile, whenever possible, the divergence that may have taken place. The state merging service included in Jgroup is aimed at supporting developers in designing state merging protocols by taking care of the problems related to the diffusion of state information among different partitions.

6.1.1 Monotonicity of Data Structures

Clearly, having a support for information diffusion is useful, but does not completely solve the problem. Differently from state transfer services provided with primary-partition object group systems, in which the state marshaled by servers belonging to the primary partition substitutes the state maintained by servers joining the primary partition, merging information coming from different partitions to the local state is a complex, application-dependent problem and cannot be automated.

To understand how an application may be rendered partition-aware, we need to analyze the data structures needed by it and the operations defined on them. For example, consider an high-consistency database and suppose that two independent updates of the same record are applied in different partitions. In this case, it very difficult to re-establish the consistency among the partitions, as there is no way to decide which should be the current value of the record. On the contrary, let us consider two increment operation on a distributed counter. In this case, it is possible to re-establish the consistency by applying both the operations.

As a general rule, an application may be rendered partition-aware whenever there is the possibility to design data structures characterized by *commutative* operations. In the counter example presented above, operations on the counter are commutative. As a further example, consider the first application presented in this chapter, the dependable registry service. Among the others, two operations are defined: a method *bind* to notify that a server has joined a group, and a method *remove* to notify that a server has left the group. Obviously, the join operations of two servers to the same group may performed in whatever order, as the final effect is that both servers are inserted; the same is true for two leave operations. Furthermore, note that we can consider commutative also a join and a leave operation executed by the same server on the same group. If the join operation is executed in one of the partitions, and the leave operation in the other, when partitions merge and the state merging protocol is completed, it is possible to agree again on a consistent state by removing the server from the group in former partition, and by inserting the server in the group in the latter partition and immediately removing it. The particular application semantics gives us the possibility of reordering operations, as the final effect is that the server is removed from the group in every registry replica.

6.1.2 Identifying Method Invocations

As we have seen in Section 3.5.2, external invocations performed by clients are not guaranteed to be executed in at most one view. The problem lies on the fact that, whenever a client's stub does not obtain an answer from a server that has been contacted in order to have a method invocation executed, there is the possibility that the server has actually executed the method, but a partitioning has occurred before having the possibility to deliver the reply to the client. Thus, in order to eventually obtain a reply, the stub has to contact another server, and this may cause the method invocation to be executed in concurrent views.

In the absence of this property, application developers have to consider the problems that may be caused by the execution of the same method by members included in different partitions. When two views merge in a new one, servers contained in them must exchange information about updates derived from external invocations. If the same update has been applied in two different views, the state merging protocol may cause the duplicated application of the update. In Jgroup, the problem has been avoided by enabling servers to obtain an invocation identifier for each of the external method invocations they execute. Each identifier is unique; when exchanging information during merging protocols, servers may associate an identifier with each of the updates, and thus discard duplicated updates.

6.1.3 Maintaining a Monotone State

In order to be capable to provide the opportune information in a state merging protocol, servers in an object group may have to maintain a large amount of information about their current state and the state of other members. This information, as said above, is monotone: in other words, tends to grow without limits. In the registry example, a server should maintain information about *remove* operations indefinitely, in order to avoid to execute a duplicated *bind* operation transmitted from a different partition during a state merging protocol. In order to avoid to have to maintain an unbounded log of the leave operations, the well-known *lease* technique may be used. In the registry example, each *bind* operation has a finite duration, after which it must be renewed. Note that in this way, *remove* operations older than this duration may be deleted from the *remove* log.

6.2 The Dependable Registry Service

As introduced in Section 3.7, the registry implementation included in Java RMI is not suitable for object groups. First of all, a non-replicated registry introduces a single point of failure; if a registry fails, the objects registered in it may not be accessed any more. Furthermore, the Java RMI registry is not capable to handle groups of server objects registered under the same name. For this reasons, the Jgroup implementation contains a dependable version of the registry service.

6.2.1 Design

The dependable registry service is composed of a collection of *registry servers* that actively maintain a database of bindings between *application servers* and group names. The dependable registry service is used by application groups to advertise their availability to provide a certain service, and by clients to retrieve stubs for these groups. Figure 6.1 illustrates the interactions among registry servers, application servers and clients. First, application servers forming a group register themselves on the dependable registry under a certain group name, using the *bind* method. Afterwards, a client that needs to interact with the application group invokes the *lookup* method on the dependable registry, in order to obtain a stub for it. Finally, the client uses this stub to perform method invocations on the application group. Note that in spite of their name, application servers act as clients with respect to the dependable registry service. In other words, both *bind* and *lookup* are included in the external remote interface of the dependable registry service.

6.2.2 Implementation

.

The algorithm used in the DRS implementation is illustrated in Figure 6.2. The DrsServer class implements the external remote interface DependableRegistry illustrated in Figure 3.15, and the empty internal remote interface InternalDrs (as internal group method invocations are not needed). Moreover, DrsServer implements also MergingMember, used by Jgroup to notify the registry server of events generated by the group membership service and the state merging service.

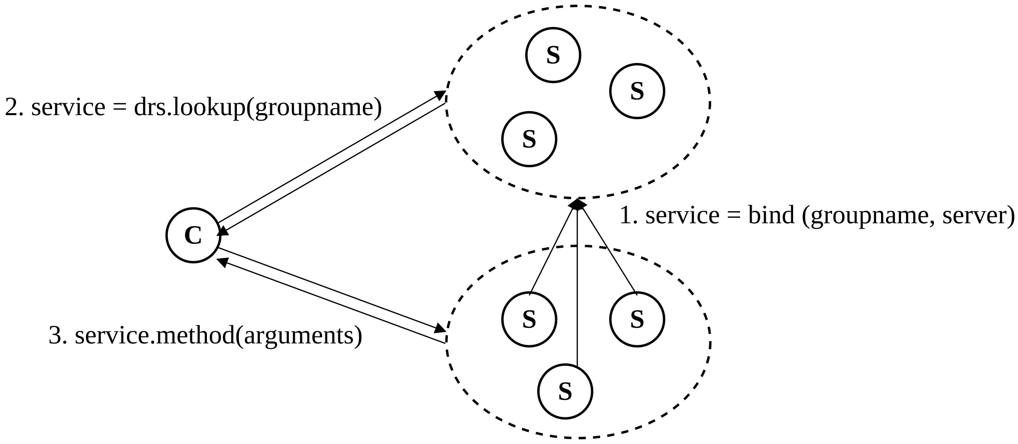

Figure 6.1: Interactions among registry servers, application servers and clients

The variables maintained by each registry server p are initialized in the constructor. Variables *bound* and *removed* are dynamic arrays indexed by registry server identifiers. Given a registry server q, variable $bound[q]$ contains the set of bindings that p knows that q has added to its database, while variable $removed[q]$ contains the set of bind identifiers Bid that p knows that q has removed. Clearly, partitionings may cause these sets to differ from the actual sets of bindings and removals applied by q. They are used to reduce the amount of information exchanged during a state merging protocol. The sets $bound[p]$ and $removed[p]$ maintained by registry server p (in the algorithm indicated as p) have a different meaning. They represent the sets of bindings and removals actually applied by p itself, with the exclusion of those applied during the current view, which are recorded in variables *newb* and *newr*. When a new view is installed, view synchrony guarantees that the operation included in *newb* and *newr* have been applied by all registry servers survived from the previous view. This information is used to update variables *bound* and *removed* at each view installation. Each item in *bound* and *newb* is a triple formed by the name of the service, the remote reference of the service and a Bid identifier. We recall that Bid objects are binding identifiers, returned by the bind method and used in the remove method to explicitly remove the particular binding. Variable *vcomp* is used to record the composition of the previous view and is updated at every view installation. Finally, variable *group* is a reference to the group manager used to join the dependable registry service group.

In the initialization, variables *bound*, *removed*, *newb* and *newr* are set to empty, while

```
class DrsServer implements DependableRegistry, InternalDrs, MergingMember {

    DrsServer() {                                                              /* Constructor */
        bound = {∅, . . . , ∅};                                          /* Array of binding sets */
        removed = {∅, . . . , ∅};                                        /* Array of removal sets */
        newb = ∅;                                       /* Bindings applied during current view */
        newr = ∅;                                       /* Removals applied during current view */
        vcomp = {p};                                              /* Current view composition */
        group = GmFactory . getGroupManager(this);                  /* Obtain a group manager */
        group. join(DRS_ID);                               /* Join the group identified by DRS_ID */
    }

    Remote lookup(String name)
        { return ({r|bid ∉ removed[p] ∪ newr ∧ (name, r, bid) ∈ bound[p] ∪ newb}); }

    String[ ] list()
        { return {name|bid ∉ removed[p] ∪ newr ∧ (name, r, bid) ∈ bound[p] ∪ newb}; }

    Bid bind(String name, Remote remote) {
        bid = group. getIdentifier();
        newb = newb ∪ {(name, remote, bid)};
        return bid;
    }

    void remove(Bid bid)
        { newr = newr ∪ {bid}; }

    void vchg(Object g, View v) {
        foreach (q ∈ v.comp ∩ vcomp) do {
            bound[q] = bound[q] ∪ newb;
            removed[q] = removed[q] ∪ newr;
        }
        newb = newr = ∅;
        vcomp = v.comp;
    }

    Object vchg(Object g, View v, MemberId[ ] dests) {
        vchg(g, v);
        return ( (⋃_{q∈dests} bound[p] − bound[q]) − {(−, −, bid)|bid ∈ removed[p]},
                 ⋃_{q∈dests} removed[p] − removed[q] );
    }

    void mergeState(Object g, Object status, MemberId[ ] sources, MemberId[ ] reconciled) {
        newb = newb ∪ status.newb;
        newr = newr ∪ status.newr;
    }
}
```

Figure 6.2: The Jgroup registry algorithm

variable *vcomp* contains only p itself. A new group manager is obtained by invoking getGroupManager on the GmFactory class. Finally, the server joins the dependable registry service group identified by DRS_ID.

The implementation of methods included in the external remote method interface is very simple. Methods lookup and list are read-only methods used to inspect the database of bindings maintained by DRS, while bind and remove are write methods used to modify this database. As the database is replicated by every registry server forming the DRS group, read-only methods are executed following the anycast invocation semantics, while write methods are based on the multicast invocation semantics. For this reason, methods bind and remove include the McastRemoteException in their *throws* clause (Figure 3.15). Method lookup inspects the local database and returns a stub containing a reference for each of the application servers that have been registered under a certain name and have not been removed. Method list returns a list of names currently in use in the local database. Methods bind inserts a triple $(name, remote, bid)$ in variable *newb*, where *name* and *remote* are provided as parameters of the method, while *bid* is a binding identifier obtained from the invocation identifier returned by the invocation of method getIdentifier on the group manager. Finally, method remove inserts a binding identifier in *newr*.

When the system is stable (i.e. when no new failures and repairs occur), each network partition contains a set of registry servers that will eventually install the same view and execute the same set of external method invocations; every update to the binding database applied by one of these servers is applied by every other server in the partition. When new failures occur, the group membership service react by installing a new view, and registry servers must opportunely react to the new failure scenario. This reaction is the core of the algorithm. When installing a new view, p adds the updates contained in *newb* and *newr* to $bound[q]$ and $removed[q]$, for each replica q in the intersection between the new view and the previous one. Furthermore, it removes from $bound[q]$ the bindings whose identifier is contained in *newr*, in order to avoid the unbound growth of *bound*. Finally, it updates the variable *vcomp* to reflect the new view composition. If a state merging protocol is needed, Jgroup elects a coordinator for each of the merging partitions; on coordinators, Jgroup invokes a special version of the *vchg* method, requiring them to return the updates that may have not been applied by registry servers contained in *dests*. In order to reduce the amount of information exchanged, the contents of arrays *bound* and *removed* are used. As we have seen in Sections 3.6 and 4.4, the information provided by coordinators is diffused to other registry servers by invoking method *mergeState* on them, in which updates are inserted in *newb* and *newr*.

In order to conclude the description of the DRS algorithm, some details about the actual implementation are needed. As noted in the previous section, the size of *removed* is not upper bounded, as identifiers of removed bindings should be kept forever in *removed*. In order to avoid the unbound growth of this set, bindings in the registry are *leased*: they are eventually removed from the database after a certain period of time, unless their lease are renewed by the application server itself. The content of *bound* is periodically inspected in order to find and remove obsolete bindings, and *removed* contains only the identifiers of the bindings that have been removed before their natural expiration. In order to simplify the implementation of application servers, group managers take care of periodically renewing the bindings for their application server.

6.3 The Partitionable Execution Service

Object groups may be used for purposes different from replication. Consider a time-intensive computation that may be decomposed into a number of independent subcomputations. Example of such computations may be ray-tracing or prime factorization. Being independent, subcomputations may be carried out in parallel by a collection of *workers*. These workers should coordinate their actions in order to guarantee the completion of the entire computation in spite of partial failures in the system. The *partitionable execution service* (PES) presented in this section generalizes this problem, by offering a distributed computing facility that accepts computation requests from clients and compute them by balancing the workload among the available workers.

6.3.1 The Design of PES

Our partitionable execution service is composed of a collection of workers that forms a group and coordinate their actions using Jgroup. The composition of the group is dynamic, i.e. new workers may be added and existing ones removed at will by an administrator. Clients issue computation requests to the group by performing external group method invocations. During normal operation (i.e. in the absence of failures), the total work is

```
public interface InternalPes extends Member {

  public void addResult(Job j, Object result);

}

public interface ExternalPes extends GmiRemote {

  public void addJobs(Job[] jobs)
  throws RemoteException, MulticastRemoteException;

}
```

Figure 6.3: The external and internal remote interfaces of the partitionable execution service included in Jgroup.

subdivided among the workers in the group. Each worker carries out the subcomputations (also called *jobs*) assigned to it and diffuses the results through internal invocations. In case of a view change, there are two possibilities. If the new view represents a contraction of the previous one due to partitions or crashes, the remaining work is redistributed among the surviving workers. Otherwise, if the new view is the result of the merging of two or more partitions, then a state merging protocol is needed so as not to repeat jobs that may have already been completed by other workers. When the merging protocol is concluded, a new redistribution of the uncompleted work is performed among all workers in the new membership.

To conclude, it is interesting to note an useful characteristics of the partitionable execution service. Since PES is written using Java and job objects are transmitted by clients to servers through Java RMI, the code executed by job objects may be transmitted from clients to servers as well. In other words, it is not necessary that servers have knowledge about the jobs they will executed; the only requirement is that each job implements a well-defined interface with a *compute* method returning a result object.

6.3.2 The Algorithm Implementing PES

Our solution to the problem is contained in class Worker illustrated in Figures 6.4 and 6.5. As in the registry example, class Worker implements an external remote interface and an internal remote interface that declare which methods may be remotely invoked by clients and by servers; moreover, Worker implements also interface MergingMember needed to receive notifications about *vchg* and *mergeState* events generated by the group membership service and the state merging service. The external interface ExternalPes, shown in Figure 6.3, contains only method *addJobs*(), used by clients to issues new requests to servers. In the same figure is shown the internal interface InternalPes, that contains method *addResult* used to transmit a computed result to the other workers. Finally, each worker extends class Thread and thus is associated with an execution thread whose behavior is described in method run() [17].

Each worker p maintains the following local variables: *my_jobs* contains the jobs that are currently assigned to p, while *vcomp* contains the composition of the current view installed by p. Initially, each worker assumes to be the only one in the system, and *my_jobs* is set to empty. Variables *jobs* and *results* are dynamic vectors indexed by worker names. At worker p, $jobs[q]$ is the list of jobs that p knows that q knows, while $results[q]$ are the partial results that p knows that q knows. Note that $jobs[p]$ contains the uncompleted jobs known by p, excluding those received in the current view; $results[p]$ are the partial results known to p, with the exception of those received in the current view. The results known by a worker are expressed as a set of pairs of the form (j, r), where w identifies a job and r is its result. As in the previous example, the knowledge about states of other workers maintained in variables *jobs* and *results* is used to reduce the amount of information exchanged during state merging protocols. Variables *new_jobs* and *new_res* contains the jobs and the results obtained in the current view, respectively; variables *dist_jobs* and *dist_res* are the jobs and the results exchanged in the current view during the state merging protocol, respectively. This separation is needed to exploit the view synchrony properties of the group method invocation service and the state merging service that guarantee that two workers surviving from a view to another deliver the same set of new jobs and the same set of new results in the first view. Variable *reconciled* contains the set of workers that have completed the last state merging protocols and thus agree and the state of the computation. Finally, variable *group* is a reference to the group manager used to join the group. In the initialization, a new group manager is obtained by invoking getGroupManager on the GmFactory class. Finally, the server joins the PES group identified by PES_ID.

The core of the algorithm is represented by the process of distribution of the workload among workers composing a view. When a view change occurs, each worker updates its variable *jobs* and *results*, by registering the fact that all workers surviving from the previous view have received the same sets of jobs and results. After this, the remaining work is subdivided among the set of workers that agree on the set of jobs and results. This set is stored in variable *reconciled*; workers are removed from *reconciled* of a worker p when a view change exclude them from the view of p; they are re-inserted in it when a state merging protocol re-establish agreement among them. The remaining work is computed through function *WorkToDo*, which returns the set of jobs for which there is no result value yet. Local work resulting from the redistribution is computed through the function *distribute*, which we do not specify here. For termination, it suffices that *distribute* assign each job to at least one worker in a view, and that at least a job is assigned to each worker. But to maximize speedup, function *distribute* has to be designed carefully.

If the new view derives from the merging of two or more partitions, a state merging protocol is needed to re-establish consistency among workers. Jgroup elects a coordinator for each of the merging partitions; on coordinators, Jgroup invokes a special version of the *vchg* method, requiring them to return information about jobs and results that may be not known to workers belonging to different partitions. In order to reduce the amount of information exchanged, the content of arrays *jobs* and *results* is used. As we have seen in Sections 3.6 and 4.4, the information provided by coordinators is diffused to other workers by invoking method *mergeState* on them. In this method, state information coming from other workers are merged to *dist_jobs* and *dist_res*; variable *reconciled* is opportunely updated with the value specified as an argument. If *reconciled* corresponds to *vcomp* (i.e., the state merging protocol has been completed), the workload is redistributed among all workers composing the current view.

The interface between clients and servers is represented by method *addJobs*(), which enables clients to request PES to execute an array of Job objects. The set of jobs is added to the local list of jobs to be performed; being based on the multicast invocation semantics, *addJobs* is executed following view synchrony.

The run() method is executed by the thread associated with the worker. It is basically constituted by a while loop in which an uncompleted job among those contained in *my_jobs* is select and performed. When a result value is available, an internal invocation on method *addResult* is issued; this method adds the new result to *new_res*. If a worker p is “jobless” (i.e. it has completed all jobs assigned to it), but there are still uncompleted

```
class Worker extends Thread implements ExternalPes, InternalPes, MergingMember {

    Worker() {
        jobs = (∅, . . . , ∅);                              /* jobs[q] are the jobs known by q */
        results = (∅, . . . , ∅);                        /* results[q] are the results known by q */
        my_jobs = ∅;                                         /* Subcomputations assigned to p */
        new_jobs = ∅;                                      /* Jobs added during the current view */
        new_res = ∅;                               /* Results computed during the current view */
        dist_jobs = ∅;                               /* Jobs exchanged in the merging protocol */
        dist_res = ∅;                             /* Results exchanged in the merging protocol */
        vcomp = {p};                                              /* Current view composition */
        reconciled = {p};                           /* Members that share the same state as p */
        group = GmFactory . getGroupManager(this);                  /* Obtain a group manager */
        group. join(PES_ID);                            /* Join the group identified by PES_ID */
        this. run();                                           /* Start the computation thread */
    }

    void vchg(Object g, View v) {
        foreach (q ∈ vcomp ∩ v.comp) do {
            jobs[q] = jobs[q] ∪ new_jobs ∪ dist_jobs;
            results[q] = results[q] ∪ new_res ∪ dist_res;
        }
        vcomp = v.comp;
        reconciled = reconciled ∩ v.comp;
        new_res = new_jobs = ∅;
        dist_res = dist_jobs = ∅;
        my_jobs = distribute( WorkToDo(jobs[p], results[p]), reconciled, p);
    }

    Object vchg(Object g, View v, MemberId[ ] dests){
        vchg(g, v);
        return ( ⋃_{r∈dests}(results[p] − results[r]), ⋃_{r∈dests}(jobs[p] − jobs[r]) );
    }

    void mergeState(Object g, Object status, MemberId[ ] sources, MemberId[ ] reconciled) {
        dist_jobs = dist_jobs ∪ status.dist_jobs
        dist_res = dist_res ∪ status.dist_res
        this.reconciled = reconciled;
        if reconciled == vcomp
            my_jobs = distribute( WorkToDo(jobs[p] ∪ dist_jobs, results[p] ∪ dist_res), vcomp, p)
    }

    void addJob(Job[ ] J) {
        new_jobs = new_jobs ∪ J;
    }
```

Figure 6.4: Partitionable Execution Service: algorithm for worker p (Part a).

```
    void addResult(Job j, Object r) {
        new_res = new_res ∪ {(j,r)};
        /* Return a set of results to a client, if possible; */
    }

    void run() {
        while true {
            if (WorkToDo(my_jobs, results[p] ∪ new_res ∪ dist_res) ≠ ∅)
                j = select(WorkToDo(my_jobs, results[p] ∪ new_res ∪ dist_res));
                group.addResult(j, w.compute());
            } elseif (WorkToDo(jobs[p], results[p] ∪ new_res ∪ dist_res) ≠ ∅) {
                group.leave(); group.join();        /* Force view change and redistribution */
            } else {
                /* Wait for new jobs; */
            }
        }
    }
}
```

Figure 6.5: Partitionable Parallel Computation: algorithm for worker p (Part b).

jobs, the algorithm forces the installation of a new view, and thus the redistribution of the workload as described above. In this way, we guarantee that new job requests are eventually redistributed by PES. When all jobs are completed, workers suspend themselves and wait for new requests.

For sake of brevity, in the algorithm we do not show how results are returned back to clients. As the computation of a set of jobs may take a long time, we assume that clients are capable to receive callbacks from servers when a set of jobs submitted with an invocation of *addJobs* is completed.

The proposed algorithm may be optimized and extended in several ways. An important problem is the fact that the same job may be performed multiple times in concurrent partitions, thus reducing the speedup. Although this problem is not completely avoidable (for example when the duration of a partition is longer than the time needed to complete all jobs), it is possible to maintain the highest speedup during partitions of short duration. For example, at the beginning of a partition, work may be redistributed so that each worker performs first all the jobs assigned to it before being partitioned, and then the jobs previously assigned to workers currently partitioned.

As in the previous section, we need to solve also the problems related to the unbounded growth of *results*. First of all, a result can be deleted from variable *results* maintained by

```
public interface InternalAllocator extends Member {

  public int request(MemberId q, int quantum);

  public void update(MemberId q, int[] values);

}

public interface ExternalAllocator extends GmiRemote {

  public void addResources(int d)
    throws RemoteException;

  public int getResources(int d)
    throws RemoteException;

}
```

Figure 6.6: The external and internal remote interfaces of the partitionable resource allocation service.

a worker only when the worker is sure that the invoking clients knows it. Nevertheless, in this case we cannot use the lease technique, as the time needed to compute one of the jobs cannot be bounded. For this reason, each server must keep track of all completed jobs, in order to avoid to repeat a computation. Clearly, servers do not need to maintain information about results, but only request identifiers. If we assume that request identifiers are few, we may maintain them for extended periods of time. The definitive removal of a request identifier may be performed whenever a view containing all workers is installed. In this case, workers may start an additional protocol to establish which information may be completely removed.

6.4 Partitionable Resource Allocation

Consider a resource allocation problem in which a relatively large number of identical resources may be allocated by clients. Practical examples of such resources could be money in a bank account, tickets for a flight, or licenses to use a software package simultaneously. In order to augment the availability of the service, we assume the existence of a collection of servers called *allocators*. Collectively, allocators manage a pool B of resources. A client can issue two kind of requests: it can ask for the allocation of a certain amount of resources, or it can insert resources in the resource pool (either by creating them or by deallocating

```
class Allocator implements InternalAllocator, ExternalAllocator, Member {

    Allocator() {
        vcomp = {p};                                      /* Current view composition */
        sent = (0, ..., 0);                               /* Number of resources sent */
        last_sent = (0, ..., 0);        /* Value of sent at the beginning of the view */
        received = (0, ..., 0);                       /* Number of resources received */
        group = GmFactory.getGroupManager(this);           /* Obtain a group manager */
        group.join(ALLOCATOR_ID);          /* Join the group identified by ALLOCATOR_ID */
    }

    void addResources(int d)
        { received[p] = received[p] + d; }

    void getResources(int d) {
        while (controlled() < d) {
            values = group.request(QUANTUM);
            foreach (q ∈ Π) do
                received[q] = received[q] + values[q];
            if (∀q ∈ Π : values[q] == 0) {
                sent[p] = sent[p] + controlled();
                return controlled();
            }
        }
        sent[p] = sent[p] + d;
        return d;
    }
```

Figure 6.7: Partitionable Resource Allocation: algorithm for allocator p (Part a)

existing ones). When a client insert d resources, B is augmented by d and the request terminates. When a client requests the allocation of an amount d of resources, the service checks the availability of d resources; if so, they are removed from B. A safety property of the protocol is that B equals to the initial balance minus the amount of resources allocated plus the amount of resources inserted in the pool. The solution must also gurantee that the amount of resources B is nonnegative.

6.4.1 Design

The allocators composing the service form a group using Jgroup. Each allocator controls a portion of the available resources and can satisfy requests up to the amount locally. If an allocator cannot satisfy a request locally, it may *transfer* additional units from some other allocator. Failures may result in temporary resource loss: units of the resource will become unavailable if an allocator controlling those units crashes, or if two allocators are partitioned while transferring units between themselves. The goal of the *resource allocation* problem is to devise an algorithm for controlling resource distribution such that if all failures are eventually repaired, the total amount of the resource that is present in the system is conserved. In addition, a client requesting k units is not blocked if at least k units of the resource remain available in the same partition as the client. In order to bound the amount of resources that can be lost, there must be a limit on both the number of transfer operations that can suffer from this problem and the amount that be lost by an individual transfer operation. The former may be addressed by avoiding the execution of more than one transfer operation at a time, and the latter by specifying a *quantum* value for the transfer. Note that resources may be lost temporarily but they should be reclaimed once failures are repaired.

For example, consider the instantiation of this problem for managing a bank account that needs to be accessed through multiple branches. In order to allow withdrawals even when branches may be partitioned, the account balance is distributed among the branches of the bank according to some predetermined policy. In this manner, any branch can honor withdrawals as long as there is enough money in the account either locally or it can be transferred from other reachable branches.

An important problem to consider is the role played by clients. In a partitionable system, it is possible that a resource is allocated to a client, but the client cannot use it due to a partitioning occurred before being notified of the fact. This scenario would require state merging protocols between clients and server. In the bank example, automatic teller machines maintain a log of their operations that may be used to reconstruct a consistent state after a partitioning. For sake of simplicity, in our allocation service we do not consider these problems, and assume that the JVM in which a client is executed contains also a server. Methods of a server may be directly (and reliably) invoked by the client running in the same JVM. In this way, we may concentrate our attention on the problems related to the transfer of resources among allocators.

6.4.2 Implementation

Our partition-aware solution to this problem is illustrated in Figures 6.7 and 6.8. Differently from the previous examples, class Allocator does not implement MergingMember, but only Member. This means that it makes use of the partitionable group membership service and not of the state merging service. The external and internal remote interfaces implemented by Allocator are named ExternalAllocator and InternalAllocator, and are shown in Figure 6.6. The external interface contains method *getResources*() to obtain the allocation of resources by the service, and method *addResources*() to insert resources in the resource pool. As said above, we assume that clients directly invoke methods on the allocators running in their JVM; thus, both methods are based on the anycast invocation semantics. The internal remote interface contains two methods: *request*() is used to request the transfer of resources from other allocators, while *update*() is invoked whenever a state merging protocol is needed.

Each allocator records the state of its resources through the following local variables, that we assume are maintained in stable storage. *vcomp* contains the composition of the current view. Variables *sent*, *last_sent* and *received* are dynamic arrays of integers indexed by allocator identifiers. For each allocator q, variable $sent[q]$ represents the amount of resources that have been transferred from p to q, while variable $last\_sent[q]$ represents the value of $sent[q]$ at the beginning of the current view. The information in *last_sent* will be used during the merging protocol to compare the amount of resources sent during the previous views with the amount of resources actually received. Finally, for each allocator q, variable $received[q]$ contains the total number of resources transferred from q to p. Different meaning have variables $sent[p]$ and $received[p]$. $sent[p]$ corresponds to amount of resources that have been removed from the resource pool by p (i.e., have been allocated to clients); $received[p]$ contains the amount of resources that have been inserted in the resource pool by p (i.e., have been deallocated by clients). The total number of resources controlled by an allocator at a certain time is given by the sum of $received[q] - sent[q]$, for each q in the set Π of allocators. This value may be obtained by invoking function *controlled*.

In the initialization, variables *sent*, *last_sent* and *received* are set to empty, while variable *view.comp* contains only p itself (in the algorithm indicated as p). A new group manager is obtained by invoking getGroupManager on the GmFactory class. Finally, the server joins the dependable registry service group identified by ALLOCATOR_ID.

The request of insertion of a set of resources in the resource pool, corresponding to the invocation of *addResources*(), is performed by opportunely incrementing the value of

```
    void request(MemberId q, QUANTUM) {
        d = Min(controlled(), QUANTUM);
        sent[q] = sent[q] + d;
        return d;
    }

    void vchg(Object g, View v) {
        if (v.comp ⊈ vcomp) {
            group.update(p, received);
                foreach r ∈ vcomp do
                    last_sent[r] = sent[r];
        }
        vcomp = v.comp;
    }

    void update(MemberId q, int[] R) {
        if (q ≠ p and last_sent[q] > R[p]) {
            sent[q] = sent[q] + (last_sent[q] − R[p]);
            warning("Failed transfer", (sent[q] − R[p]), q);
        }
    }
}
```

Figure 6.8: Partitionable Resource Allocation: algorithm for allocator p (Part b)

synchronized$[p]$. When method *getResources*() is invoked on allocator p, p checks whether there are enough resources under its control. If so, an amount d of resources is allocated by incrementing variable *sent*$[p]$ by d units and by returning d to the invoker. If, on the contrary, the resources controlled by p are not enough to satisfy the request, p invokes internal method *request* on the group, requesting each of the allocators included in its current view to transfer an amount QUANTUM of resources to p. The repeated invocation of *request* terminates either when p controls enough resources to satisfy the client request, or when all resources in the current view have been transferred to p. In the latter case, the amount of resources allocated and returned to the client are less than what requested. As the reader can see, the internal group method invocation of *request* returns an array of integers, one for each of the allocators that have executed the method. Once obtained the array, variable *received* is updated in order to record the receipt of the transferred resources. Method *request*() returns the quantity of requested resources, or less if the controlled resources are not enough. The transfer operation is recorded on variable *sent*.

When a view change occurs, there are two possibilities. If the new view is a contraction of the previous one, no action is needed: view synchrony guarantees that each transfer initiated during the previous view between allocators survived in the new one has been successfully completed. If, on the contrary, the new view contains allocators not belonging to the previous one, an additional merging protocol must be executed, in order to resume control over units of resource possibly lost during previous partitionings. Upon a view change that corresponds to the merge of partitions, allocator p invokes internal method *update* on the group, containing its vector of received resources Such message will be delivered in the current view. Then, p copies its vector *sent* in *last_sent*. Finally, p updates variable *view.comp* with the new view composition. When executing method *update*, p checks whether some resources sent by p to the invoker q were lost during the last partition. Specifically, $last\_sent[q]$ contains the number of resources p has sent to q up to the installation of the current view of p, while $R[p]$ contains the number of resources q has received from p up to the installation of the current view of p. If p has sent more resources to q than what q has received, then p may conclude that $last\_sent[q] - R[p]$ units were lost during the last partition. From the Uniqueness property of GMIS, these messages will never be delivered by q. Thus, the resources that were being transferred through the undelivered messages can be reclaimed by p and the application is notified that the transfer has failed.

Chapter 7

Related Work

In this chapter, we discuss how our work may be compared with other projects based on group communication. In particular, we take into consideration three different aspects: first, we compare our formal specification with those of existing group communication systems. Then, we compare our prototype implementation with other algorithms presented in the literature; and finally, we discuss the relationship between the object group model associated with Jgroup and other object group models.

7.1 Group Communication Specifications

The group communication paradigm [13] has been the focus of much experimental work in the last years. Several academic and commercial group communication toolkits have appeared, including Isis [14], Phoenix [49], Transis [23], Totem [61], Newtop [28], Horus [76], Ensemble [39] and Spread [3]. Beside this experimental effort, much theoretical work has been performed as well. In particular, the problem of obtaining a specification for the group communication problem has been discussed in several scientific papers, some of them associated with one of the prototype listed above [70, 50, 49, 5, 62, 28, 34], others discussing the group communication problem and its relation with existing paradigms for developing dependable applications in distributed systems [4, 67, 18, 29, 69, 45, 77]. In this section, we resume some of the main contributions to the theoretical discussion related to group communication and we compare the specification effort performed in the context of this thesis with other specifications in the literature.

In spite of the efforts spent in the attempt of obtaining a specification for group communication, the distributed systems community has not yet reached an agreed definition of the group membership problem, especially for partitionable systems. There are good reasons why this agreement has proven elusive. The specification of any fault-tolerant distributed service requires a delicate balance between two conflicting goals. The specification must be weak enough to be implementable, yet it must be strong enough to be "useful". In Chapter 5 and Appendix A we prove that our specification is implementable in an asynchronous distributed system augmented with an unreliable failure detector of class $\diamond\tilde{\mathcal{P}}$. Unfortunately, obtaining a formal proof of the usefulness for our specification is impossible; the final word about the usefulness of a service should be left to application developers using it. For this reason, in Chapter 6 we discuss how Jgroup can be used to implement realistic application in partitionable systems. Here, we show that the Jgroup specification is strong enough to prevent well-known trivial solutions, as studied in the literature [4].

In their comprehensive study [4], Anceaume et al. analyze some of the existing specifications and point out three different trivial solutions for a partitionable group membership service: (1) At each member p, install only views composed of p alone, (2) Install views with some fixed, *a priori* agreed upon composition, independent of the actual execution, (3) After a certain time, do nothing (install no views). The first trivial solution exhibits what has been called *capricious view splitting*. The second trivial solution exhibits *capricious view installation* since members are added or removed from views without any justification. The third trivial solution lacks liveness. As an example, the specification of the Newtop group membership service [28] may be satisfied by a protocol where each member p initially installs a view composed only of p and then stops executing new views. Thus, Newtop suffer of capricious view splitting and capricious view installation; moreover, it lacks liveness. The specification of Transis [25] can be satisfied by a protocol that installs an *a priori* agreed sequence of views: every member p installs a singleton view composed only of p and maintains this view for most of the time; periodically, p installs a fixed sequence of views including all of the subsets of the set of members taking part in the computation. Then, p returns to its singleton view. This solution suffer of capricious view splitting and capricious view installation.

The main properties that make our specification non-trivial are Property GM2 and Property GM3. Property GM2 prevents capricious view splitting by requiring that mutually reachable members appear together in a single view. Capricious view installation is prevented by requiring that the composition of installed views be a function of the actual unreachability pattern that occurs during an execution, which cannot be predicted ahead of time. At the same time, Property GM2 and Property GM3 guarantee liveness since in both cases view installations cannot be delayed indefinitely under stable conditions.

It is important to note that Property GM2 also prevents the following subtle combination of the first and the second trivial solutions: alternating installation of normal views related to the actual partitioning pattern and singleton views composed of only the installing member. Specifications requiring the eventual installation of a new view containing permanently reachable members are not sufficient to rule out this trivial solution. For this reason, we require that from some time onwards, all views installed by one member will contain the other, as long as they remain reachable.

7.1.1 The Horus Project

Horus [76] (together with its evolution Ensemble [39]) is one of the most important experimental group communication service. The Horus specification [34] has many similarities with our proposal, particularly with respect to safety properties such as View and Message Agreement. However, there are also important differences with respect to the specification of non-triviality properties: the Horus specification is conditional to the output of the failure detector present in the system. This approach is suggested also in the work of Anceaume et al. [4], that criticizes the formal model of Transis because in the specification "failure suspicions are generated but then completely ignored". In our opinion, it is highly desirable that the specification be formulated in terms of the actual execution characterization and not in terms of failure detector suspicions [67, 34]. Otherwise, the correctness of the specification itself would be conditional on the properties of the failure detector producing the suspicions. It is reasonable for the correctness of the partitionable group membership service *implementation*, but not its *specification*, to be conditional on the correctness of the failure detector that it is based on.

The Horus specification presents also another interesting characteristic. It permits to choose among two different models of virtual synchrony, called respectively *strong virtual synchrony* and *weak virtual synchrony*. Strong virtual synchrony guarantees that a message be delivered in the context of the view in which it was sent. This property may be exploited by applications to minimize the amount of context information to be sent with each message. Unfortunately, strong virtual synchrony cannot be implemented without discarding messages or blocking sending of messages for a certain time period before a new view is installed. For this reason, Horus provides also weak virtual synchrony. In this model, every installation of a view v is preceded by at least one *suggested view* event. The membership of the suggested view is guaranteed to be a superset of v. Weak virtual synchrony guarantees that every message sent in the suggested view is delivered in the next regular view. In Jgroup, a similar approach is taken. Every installation is preceded by a prepare upcall; messages sent before the end of this upcall will be delivered in the view that is current. Messages sent after this upcall will be delivered in the next view.

7.1.2 The Transis Project

The specification of the Transis system has been modified several times. A flaw was discovered by Anceaume et al. [4] in the first version of the specification [24]. Subsequent versions presented similar problems [25, 26]. As said above, the Transis specification can be implemented by installing a pre-determined sequence of views, independent from the actual execution. Recently, the Transis group has designed two new membership protocols called Congress and Moshe [5]. Congress provides a simple group membership protocol, while Moshe, which extends Congress, provides a full group communication service. The specification of Moshe has many similarities with our proposal. Moshe includes properties such as *View Identifier Local Monotony*, *Self Inclusion* and *View Synchrony*, that can be compared to Property GM5, Property GM6 and Property GMI3. Property GMI2 is enclosed in Properties *Self Delivery* and *Termination of Delivery* of Moshe. Property GMI4 and Property GMI1 are not included in Moshe; note that the lack of Merging Rule is counterbalanced by the extended virtual synchrony model [62], in which *transitional sets* are delivered together with views. Transitional sets convey the information needed to solve problems generated in undesirable scenarios, as those pointed out in Chapter 4.

The non-triviality requirements are the main difference between Moshe and our proposal. Moshe includes a property called *Agreement on Views*, that may be compared with the requirements of Properties Property GM2, Property GM3 and Property GM4. This property forces a set of members S to install the same sequence of view only if there is a time after which every member in S is correct, mutually reachable with any member in S, mutually unreachable from any object not in S and not suspected by any member in

S. As in our proposal, this requirement may be relaxed by requiring that the condition expressed in it holds for a sufficiently long period of time, and not forever; moreover, a service including this requirement cannot be implemented in a truly asynchronous system. In spite of these similarities, the non-triviality requirement in Moshe and in our proposal present an important difference: in Moshe, two mutually reachable members are not required to install a common sequence of views, if the rest of the system is not stable. This design choice is motivated by the will to avoid notifying the application of obsolete views when the network is unstable, thus avoiding extra overhead due to state transfer or state merging. In our proposal, this pair of members install the same sequence of views and may permanently cooperate. In the case of an highly unstable network, opportune tuning mechanisms in the lower layers (e.g., adaptive timeout values) may reduce the number of obsolete views installed.

7.1.3 The VS Service

Fekete et al. [29] present a formal specification for a partitionable, view-oriented group communication service. In the same work, their proposal is used as a building block to construct a totally ordered broadcast algorithm. Their approach to group communication is completely different from every other view oriented paradigm seen before. First of all, they argue that existing specifications are all complicated and difficult to understand. For this reason, one of their goals is the design of a simple specification. To achieve this, their specification "does not describe all the potentially-useful properties of any particular implementation. Rather, it includes only the properties that are needed for the ordered-broadcast application". In other words, their specification is tailored at resolving only the totally-ordered broadcast problem.

7.1.4 Ordering properties

An important aspect of group communication is the order in which messages (in our case, invocations) are delivered to servers. In this dissertation, the problem of ordering has not been discussed. Many ordering semantics may be provided, including FIFO, causal and several variants of total order. All these semantics may be implemented on top of the reliable multicast layer. Future distribution of Jgroup will include additional layers that implement these semantics.

7.2 The Algorithm

The algorithms developed in the context of this thesis should be considered as a proof of concepts. Several optimizations may be applied, in particular with respect to the multi-send layer described in Section 5.3 and the view agreement algorithm described in Section 5.4.

As regards the view agreement algorithm, several proposals have appeared in the literature [56, 24, 49, 39, 43]. The most interesting one has been proposed by Keidar et al. [43]. Keider's work describes a novel scalable group membership algorithm designed for wide-area networks. Since message latency in WANs may be large, their protocol attempts to minimize the number of message rounds exchanged among the servers: once a change in network connectivity is detected, each server multicasts a message to the other servers, containing an estimate of the current membership; if these messages are in agreement, a new view is installed. Thus, if the maximum message latency in the network is δ, the membership algorithm usually terminates within δ time after all servers detect the change in reachability. If temporary lack of symmetry or transitivity in the network causes surviving members to differ too much in their detections of failures and reconnections, then the Keidar's algorithm may be required to run a re-synchronization protocol among the servers. In this case, the algorithm terminates within at most 3δ time once network stabilization occurs and all of the servers correctly detect the network connectivity.

7.3 The Object Group Paradigm

In the last few years, the problem of integrating group communication with distributed object technologies such as CORBA [37] and Java RMI [78, 52] has been the subject of intense investigation.

Examples of experimental distributed object frameworks based on group communication are Electra [47], Orbix+Isis [46], OGS [30], Eternal [64], Newtop [60], Filterfresh [12] and JavaGroups [11]. We can classify these systems using two fundamental parameters: which distributed object model they extend (Java RMI vs CORBA), and which approach they use with respect to partitionings (primary-partition vs partitionable). The primary-partition approach is used in Electra, Orbix+Isis, OGS and Eternal to extend CORBA, and in Filterfresh to extend Java RMI. The primary-partition approach is not satisfactory for the design of high-available applications in partitionable environments, as it requires the existence of a totally-connected majority of correct members, and this requirement may be rarely satisfied in a highly partitionable environment. The absence of a primary partition may lead to the total blocking of a group communication service and the applications based on it. More importantly, even when a primary partition exists, members not belonging to it cannot collaborate until communication with the primary partition is restored, thus precluding continued availability in concurrent partitions. Newtop (which extends CORBA) and Jgroup solve these problems by providing distributed object toolkits based on partitionable group communication service. This feature makes them the right support for the development of partition-aware applications in partitionable distributed systems.

7.3.1 Object Group Systems Based on CORBA

Most of this research effort has been dedicated to CORBA; the resulting object group systems may be classified according to three main categories [30]. The *integration approach* consists in modifying and extending an ORB using existing group communication services; Orbix+Isis [41] and Electra [47] are based on this approach. The former was a commercial product from Iona Technologies and Isis Distributed Systems, which integrates Orbix, a CORBA-compliant development environment, with the group communication facilities of Isis [14]. The latter is an academic CORBA programming environment written in C++, which can be configured for various communication subsystems such as Isis [14] or Horus [76]. Both toolkits allow programmers to treat collections of CORBA objects as if they were a single entity, and clients invoke operations on object groups without needing to know the membership of the group. The integration approach is appealing for its transparency, since clients do not need to know if the service they access is provided by a single object or a group; nevertheless, the resulting ORBs are not CORBA-compliant.

An alternative methodology is the *service approach* which consists in providing group communication as a service on top of the ORB. Clients hold references to an object group service, whose task is to provide primitives to communicate with groups of objects. An example of object group toolkit based on the service approach is the Object Group Service [31, 32, 30]. Although the service approach is CORBA-compliant, clients cannot transparently access a group of objects as if they were a single entity, but they must use the primitives offered by the OGS.

A third approach called *interceptor approach* also make use of an existing object group systems; in this approach, messages issued by an ORB are intercepted and mapped on to calls on the group communication systems. The best known example of this approach is the Eternal system [64, 65, 66, 63]. Eternal replicates objects, maintains the consistency of replicated objects, and distributes objects across the system. The major advantage of this approach is that no modifications to the ORB are required. The shortcomings are that the resulting implementation depends on low-level mechanisms of the underlying operating system, and this compromises its portability; secondly, as the group communication system is not available to CORBA application builders, it can only be used in a fixed manner (replication in the case of Eternal).

Being based on Java RMI, Jgroup does not suffer of any of the problems experienced by these approaches. Java RMI enables the construction of group references, whose task is to manage interactions between clients and remote object groups. The resulting system provides transparent access to object groups and completely satisfies the specification of Java RMI.

7.3.2 Newtop

As we have said, the most important difference between Jgroup and existing object group systems is that Jgroup is aimed at supporting the development of high-available applications to be deployed in partitionable environments. Existing systems are generally based on the primary-partition approach and thus cannot be used to develop applications capable to continue providing their service in every partitions of the system. The only exception is Newtop [60], that is based on the partitionable approach. Newtop object group service has been designed to support a wide variety of group interactions. Objects can simultaneously belong to many groups, group size could be large, and objects could be geographically large separated. Newtop can provide causality preserving total order message delivery to members of a group, ensuring that total order delivery is preserved even for multi-group objects. Clients interact with a server group by forming a group consisting of the server group and itself (a *client/server*) group. When a client no longer requires a service, the client/server group may be disbanded. Client/server groups guarantee that interactions among clients and servers have the same properties as interactions among servers, but the costs of creating and disbanding a group may be too high when the set of clients is large. Finally, Newtop differs from Jgroup also because it does not provide any support for reconciling the states of servers belonging to multiple merging partitions.

7.3.3 Filterfresh

Filterfresh [12] and Jgroup share the same goal, i.e. the integration of the group communication paradigm with the Java distributed object model. Due to the constraints inherent to the architecture of Java RMI, the approaches they follow are similar: both offer a reliable invocation mechanism for remote object groups composed of a collection of remote objects that cooperate through a group communication service, and a distributed implementation of the RMI registry. Nevertheless, there are also important differences. First of all, Filterfresh does not provide the multicast semantics for method invocations issued by clients, but only the anycast semantics. In other word, view synchrony is not extended to method invocations; to circumvent this limitation, an object completing a method invocation should take the responsibility of multicasting it to the other group members. Filterfresh group references contain a reference for only one of the remote objects forming a group. If this remote object is partitioned from the invoking client, the group reference asks the reliable registry for the reference of another remote object belonging to the group. This means that each invocation failure requires an additional RMI interaction with the reliable registry. This slows down the invocation mechanism in highly partitionable distributed systems such as large-scale and wireless networks. In contrast, Jgroup group references contain a reference for each of the replicas forming a group and are able to independently select an operational replica in their partition.

7.3.4 JavaGroups

JavaGroups [11] is a group communication toolkit written in Java providing reliable multicast communication. Its first version uses the group communication protocols provided by Ensemble [39]. Ensemble offers to Javagroups a rich variety of communication paradigm, but limits its portability. Furthermore, the group method invocation facilities of JavaGroups are not transparent, as they are based on the exchange of objects that encode method invocation descriptions.

7.3.5 The OMG Request for Proposals on Fault-Tolerance

In April 1998, the OMG issued a *Request for Proposals* (RFP) entitled *Fault-Tolerant CORBA Using Entity Redundancy* [36]. The objective of this document is the following:

> This RFP addresses the need for standard CORBA functionality supporting fault tolerant applications, such that clients of these applications will be largely insulated from such details as management of redundant copies, failure masking, and recovery. This will greatly reduce the burden on the designers and implementors of fault tolerant applications, who must currently manage replicas, and respond to faults by switching services form the faulty component to replicated components, often using proprietary extensions to an ORB.

The RFP does not specify the unit of redundancy, but uses the term *entity* and presents the concepts of *entity group*. Among the mandatory requirements of the RFP is the specification of the unit of redundancy; definition of interfaces for passive entity groups; support for state synchronization; support for dynamic group management; interfaces that allow entity redundancy to be transparent to clients; ability of entity groups to determine failed members; interfaces for recovery control. In addition, the RFP defines some optional requirements, such as support for active entity groups; interfaces for using failure detection and notification mechanisms; support for suppressing multiple responses from an entity group to a single request.

Several submissions have responded to this RFP [74, 73, 21, 27, 68]. Several companies have proven to be interested in the development of a specification for a fault-tolerant CORBA, meaning that there is a great interest in the problem. Submitting companies include Ericsson, Eternal Systems, Inprise Corporation, Iona Technologies PLC, Lucent Technologies, Oracle and Sun Microsystems.

In October 1999, a joint revised submission has been issued [38]. The joint revised submission is based on the work of Eternal, and supports a wide range of fault-tolerant strategies, including request/retry, redirection on alternative servers, passive (primary/backup) replication, and active replication based on the primary-partition which provides more rapid recovery from faults. At the time of this writing, the joint revised submission is being considered in order to be adopted as a standard by the OMG.

Chapter 8

Conclusions

Java RMI [78, 52] do not specify any abstraction for supporting the development of dependable applications in spite of object failures and network partitionings. Group communication, on the other hand, has proven to be an adequate paradigm for building reliable and high-available distributed systems. The integration of these complementary technologies in order to obtain a programming framework capable to support the design and the implementation of object-oriented, dependable distributed systems is an open research area [47, 46, 12, 49, 63, 60, 11].

In this thesis, we have presented the design and the implementation of Jgroup, an extension of the Java distributed object model based on the group communication paradigm. Differently from other group-oriented extensions of existing distributed object models, Jgroup is expressly aimed at supporting the development of reliable and high-available distributed applications in partitionable environments. Jgroup enables the creation of groups of remote objects that cooperate towards some common goal using a partitionable group communication service. Remote object groups simulate the behavior of standard remote objects by implementing a set of remote interfaces and by enabling clients to remotely invoke the methods defined in these interfaces through the standard Java RMI mechanism. For the first time, communication among servers forming a group is based on group method invocations as well, offering a completely object-oriented framework for programmers. Finally, Jgroup is completed by a state merging service, which simplifies the development of state merging protocols needed to re-establish a consistent state after the disappearing of a partitioning.

The design and the development of Jgroup has presented both design and engineering issues. When we have started the Jgroup project, the idea was simply to implement our group communication specification using Java and use the resulting implementation in order to validate our design choices. Soon after, the prototype was extended by enabling clients to perform external group method invocations with anycast semantics. When we have decided to introduce invocations with multicast semantics, we have realized that the method invocation paradigm could be appropriate also for internal communication among groups, as the complete object-oriented environment offered by Jgroup has interesting implications with respect to abstraction, portability, and reusability. At that point, our task has been to identify the right semantics for internal group method invocations.

With respect to engineering issues, by implementing a prototype we have tried to answer the following question: how can group communication be integrated with a client/server communication paradigm as Java RMI? The first concern of this integration is transparency for clients, which should not be required to be aware of the fact that the service they are accessing is composed of a collection of replicated servers. Transparency has been obtained partly using the existing mechanisms included in Java RMI in order to allow users to extend the protocol, and partly by adopting some workarounds where the support provided by Java RMI has proven to be not sufficient. The resulting experience in extending standard Java RMI has been found of interest by Sun Microsystems, which plans to completely re-design Java RMI in order to enhance the extension support provided to users. On this regard, our research group at the University of Bologna and Sun have started a collaboration in order to share this experience.

Jgroup has proven to be an useful support for the development of partition-aware distributed applications. Examples of applications based on Jgroup can be found in Chapter 6, where a dependable registry service, a partitionable execution service and a partitionable allocation service have been presented. However, several issues still remain open. Some ideas of extension for Jgroup are outlined below.

Algorithm Optimization

The current implementation of Jgroup has been developed as a proof of concept. Several optimizations may be applied to it. In particular, the multi-send layer described in Section 5.3 should be completely rewritten, as it does not make use of any broadcasting capability of the underlying network. Furthermore, the partitionable group membership layer could be optimized as well. Currently, our view agreement algorithm terminates in four communication rounds; as message latency in WAN may be large, this may be very costly. Future version of Jgroup will include a view agreement algorithm that terminates in a single communication round *in most cases* [43].

Security Extension

Recently, Javasoft has released a draft for the specification of security extension for Java RMI [54]. The current version of Java RMI already includes security mechanisms, as the possibility of customizing the sockets using for communication for encrypting data. As standard Java RMI, Jgroup enables socket customization. However, neither Java RMI nor Jgroup provide support for authentication and complex authorization policies.

The Java 2 platform provides mechanisms for defining and using fine-grained access control policies, built around the notion of what code is being executed. Permission is granted to code, based on what location the code came from and who signed that code. Missing from the security architecture is any notion of who is executing the code. The Java Authentication and Authorization Service [53] extends the security architecture, providing mechanism to authenticate subjects, execute code on behalf of subjects, and grant permissions to subjects.

The RMI Security Extension [54] further extends the security architecture to distributed systems, providing mechanisms to mutually authenticate client and server subjects during a remote invocation, protect the communication from third parties, and execute code in the server on behalf of the client's subject.

Future version of Jgroup will integrate the Java RMI Security Extension. In Jgroup, authentication of server subjects will be substituted with the authentication of server group subjects. Furthermore, security mechanisms will be provided also among servers composing a group.

Extension to Jini

Another important technology recently introduced by Sun is Jini [55]. A Jini system is a distributed system based on the idea of federating collections of users and the resources required by those users. The overall goal is to turn the network into a flexible, easily administered tool on which resources can be found by human and computational clients.

Resources can be implemented as either hardware devices, software programs, or a combination of the two. The focus of the system is to make the network a more dynamic entity that better reflects the dynamic nature of the workgroup by enabling the ability to add and delete services flexibly.

The Jini systems consists of a set of components that provides an infrastructure for federating services in a distributed system, and a programming model that supports and encourages the production of distributed services. However, no support is given for dependability of these services. Future extension of Jgroup will be able to register object groups as Jini services.

References

[1] O. Amir, Y. Amir, and D. Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop*, Le Mont Saint-Michel, France, June 1993.

[2] Y. Amir, G. Chockler, D. Dolev, and R. Vitenberg. Efficient State Transfer in Partionable Environments. In *Proceedings of the 2nd European Reasearch Seminar on Advances in Distributed Systems (ERSADS)*, Zinal, Switzerland, March 1997.

[3] Y. Amir and J. Stanton. The Spread Wide-Aread Group Communication System. Technical report, Center of Networking and Distributed Systems, Johns Hopkins University, Baltimore, Mariland, April 1998.

[4] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the Formal Specification of Group Membership Services. Technical Report TR95-1534, Department of Computer Science, Cornell University, August 1995.

[5] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable Group Membership Services for Novel Applications. In *Proceedings of the DIMACS Workshop on Networks in Distributed Computing*, pages 23–42. American Mathematical Society, 1998.

[6] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[7] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Membership and View Synchrony in Partitionable Asynchronous Systems: Specifications. *ACM SIGOPS Operating Systems Review*, 31(2):11–22, April 1997. Also appears as Department of Computer Science, Technical Report UBLCS-95-18, University of Bologna, Italy, September 1996.

[8] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionable Group Membership: Specification and Algorithms. Technical Report UBLCS-97-1, Department of Computer Science, University of Bologna, January 1997.

[9] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 2000. To appear. Also appears as Department of Computer Science, Technical Report UBLCS-98-01, University of Bologna, Italy, April 1998 (Revised October 1999).

[10] Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 184–191, Amsterdam, The Netherlands, May 1998.

[11] Bela Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Department of Computer Science, Cornell University, July 1998.

[12] A. Baratloo, P. Emerald Chung, Y. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Hot Replication of Java RMI Server Objects. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[13] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[14] K. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[15] K.P. Birman and T.A. Joseph. Exploiting Virtual Synchrony in Distibuted Systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, 1987.

[16] N. Brown and C. Kindel. Distributed Component Model Protocol DCOM/1.0. Technical report, Microsoft Corp., 1996.

[17] Patrick Chan and Rosanna Lee. *The Java Class Libraries*. Addison-Wesley, 1997.

[18] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–330, May 1996. Also available as technical report TR95-1533, Department of Computer Science, Cornell University.

[19] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, March 1996.

[20] G. Collson, J. Smalley, and G.S. Blair. The Design and Implementation of a Group Invocation Facility in ANSA. Technical Report MPG-92-34, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster, UK, 1992.

[21] Highlander Communication, Inprise Corporation, Lockheed Martin Corporation, Lucent Technologies, and TIBCO Inc. Fault Tolerant CORBA Using Entity Redundancy: Joint Initial Submission. Technical Report orbos/98-10-09, Object Management Group, Framingham, MA, October 1998.

[22] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997. Also available as Technical Report TR96-1608.

[23] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996.

[24] D. Dolev, D. Malki, and R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[25] D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[26] D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, May 1996.

[27] Ericsson, IONA, and Nortel. Fault Tolerance Joint Initial Submission. Technical report, Object Management Group, Framingham, MA, 1998.

[28] P.E. Ezhilchelvan, R.A. Macêdo, and S.K. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS)*, Vancouver, BC, Canada, June 1995.

[29] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and Using a Partitionable Group Communication Service. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997.

[30] P. Felber. *The CORBA Object Group Service: a Service Approach to Object Groups in CORBA.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, January 1998.

[31] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the 15th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 150–159, Niagara-On-The-Lake, Canada, October 1996.

[32] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, January 1998.

[33] M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[34] R. Friedman and R. Van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, Department of Computer Science, Cornell University, March 1995.

[35] R. Friedman and A. Vaysburg. Fast Replicated State Machines over Partitionable Networks. In *Proceedings of the 16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[36] Object Management Group. Fault Tolerant CORBA Using Entity Redundancy. Technical Report orbos/98-04-01, Object Management Group, Framingham, MA, April 1998.

[37] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 2.3.* OMG Inc., Framingham, Mass., March 1998.

[38] Object Management Group. Revised Joint Fault Tolerance Submission. Technical report, Object Management Group, Framingham, MA, 1999.

[39] M. Hayden. *The Ensenble System.* PhD thesis, Department of Computer Science, Cornell University, January 1998.

[40] M. Hiltunen and R. Shlichting. Properties of Membership Services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, 1995.

[41] Iona and Isis. *An Introduction to Orbix+Isis.* Iona Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[42] I. Keidar and D. Dolev. Efficient Message Ordering in Dynamic Networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, Philadelphia, PA, May 1996.

[43] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Technical Report MIT-LCS-TM-593, Massachussets Institute of Technology, Boston, MA, June 1999.

[44] R. Khazan. Group Communication as a Base for a Load-Balancing, Replicated Data Service. Master's thesis, Departement of Electrical Engineering and Computer Science, Massachussets of Thecnology, Cambridge, MA, June 1998.

[45] R. Khazan, A. Fekete, and N. Lynch. Multicast Group Communication as a Base for a Load-Balancing Replicated Data Service. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC)*, August 1998.

[46] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[47] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1st USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Monterey, CA, June 1995.

[48] S. Maffeis. The Object Group Design Pattern. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, June 1996.

[49] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large-Scale Networks.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1996.

[50] C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *Proceedings of the 2nd Open Workshop of the ESPRIT Project Broadcast*, Grenoble, France, July 1995.

[51] Sun Microsystems. Why developers should not write programs that call "sun" packages. Url: `http://java.sun.com/products/jdk/faq/faq-sun-packages.html`.

[52] Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.50.* Sun Microsystems, Inc., Mountain View, California, October 1998.

[53] Sun Microsystems. *Java Authentication and Authorization Service Specification.* Sun Microsystems, Inc., Mountain View, California, September 1999.

[54] Sun Microsystems. *Java Remote Method Invocation Security Extension Specification, Early Look Draft 2.* Sun Microsystems, Inc., Mountain View, California, September 1999.

[55] Sun Microsystems. *Jini Architectural Overview.* Sun Microsystems, Inc., Mountain View, California, March 1999.

[56] S. Mishra, L. Peterson, and R. Schlichting. A Membership Protocol Based on Partial Order. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 137–145, Tucson, AZ, February 1991.

[57] A. Montresor. A Dependable Registry Service for the Jgroup Distributed Object Model. In *Proceedings of the 3rd European Reasearch Seminar on Advances in Distributed Systems (ERSADS)*, Madeira, Portugal, April 1999.

[58] A. Montresor. The Jgroup Reliable Distributed Object Model. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 389–402, Helsinki, Finland, June 1999.

[59] A. Montresor, R. Davoli, and Ö. Babaoğlu. Group-Enhanced Remote Method Invocations. Technical Report UBLCS-99-5, Department of Computer Science, University of Bologna, April 1999.

[60] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and Implementation of a CORBA Fault-Tolerant Object Group Service. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 361–374, Helsinki, Finland, June 1999.

[61] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Group Communication System. *Communications of the ACM*, 39(4), April 1996.

[62] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, Poznan, Poland, June 1994.

[63] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent Object Replication in the Eternal System. *Distributed Systems Engineering*, 4(2):81–92, January 1998.

[64] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Consistency of Partitionable Object Groups in a CORBA Framework. In *Proceedings of the 30th Hawaii International Conference on System Sciences (HICS)*, pages 120–129, January 1997.

[65] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault-Tolerance. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 81–90, Portland, OR, June 1997.

[66] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Replica Consistency of CORBA Objects in Partitionable Distributed Systems. *Distributed Systems Engineering*, 4(3):139–150, September 1997.

[67] G. Neiger. A New Look at Membership Services. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, May 1996.

[68] Oracle. Oracle Initial Fault Tolerance Submission. Technical Report orbos/98-10-13, Object Management Group, Framingham, MA, 1999.

[69] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A Dynamic View-Oriented Group Communication Service. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, June 1998.

[70] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 341–352, August 1991.

[71] R. Sessions. *COM and DCOM: The Microsoft's Vision for Distributed Objects.* Wiley's and Son, 1997.

[72] J. Sussman and K. Marzullo. The *Bancomat* Problem: an Example of Resource Allocation in a Partitionable Asynchronous System. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC)*, 1998.

[73] Eternal Systems, Sun Microsystems, and University of California at Santa Barbara. Fault Tolerance for CORBA - Version 1.0 - Initial RFP Submission. Technical Report orbos/98-10-08, Object Management Group, Framingham, MA, October 1998.

[74] Objective Interface Systems. Fault Tolerant CORBA Using Entity Redundancy: Initial Submission. Technical Report orbos/98-10-03, Object Management Group, Framingham, MA, October 1998.

[75] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.

[76] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[77] R. Vitenberg, I. Keidar, G.V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report MIT-LCS-TM-593, Massachussets Institute of Technology, Boston, MA, September 1999.

[78] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, June 1996.

Appendix A

Proof of Correctness

In this chapter, we prove that the algorithms illustrated in Chapter 5 are correct, i.e. they satisfy the specifications illustrated in Chapter 4. Before starting, we introduce some simplifying notation. Since we need to refer to several objects, we index variables names with object names they are local to (for example, the variable *reachable* of MSL at object p is denoted with $reachable_p$). Moreover, we denote variables as functions of time so as to refer to their value at a particular time (for example, $reachable_p(t)$).

A.1 The Multi-Send Layer

In this section, we prove that the algorithm presented in Section 5.3 is correct.

Theorem A.1 (MSL Specification)

The algorithm presented in Section 5.3 satisfies Property MS1.

Proof:

(a) We must show that if a server q is continuously unreachable from p, then eventually p will continuously m-suspect q. Let Θ and $\Psi = \Pi - \Theta$ be two sets of servers such that there is a time t_0 after which the servers in Ψ are permanently unreachable from servers in Θ. We must show that there is a time after which every event $msuspect(P)$ generated by correct server $p \in \Theta$ are such that $\Psi \subseteq P$. By Strong Completeness, there is a time t_1 after which every correct server $p \in \Theta$ will permanently suspect the servers in Ψ. It follows that every message $\langle M, A, F \rangle$ sent by each $p \in \Theta$ after t_1 is such that $\Psi \subseteq F[q]$. The proof is by induction on $|\Theta|$. If $|\Theta| = 1$, after t_1 the set $fd_p[p]$ (and so $reachable_p$) is permanently equal to $\{p\}$; thus, the claim trivially holds. If $|\Theta| = n$, let Θ' denote the set $\{q | q \in \Theta \wedge \exists \bar{t}, \forall t > \bar{t} : \Psi \subseteq fd_p[q](t)\}$. If $\Theta = \Theta'$, then there is a time after which $reachable_p \cap \Psi = \emptyset$. If, on the other end, $\Theta' \subset \Theta$, none of the messages sent by servers in $\Theta - \Theta'$ after t_1 is received by servers in Θ'; by Eventual Symmetry, there is a time t_2 after which Θ' and $\Psi' = \Psi \cup (\Theta - \Theta')$ are definitely unreachable. Since $|\Theta'| < |\Theta|$, the proof is concluded by induction.

(b) Let q be a server always reachable from p after time t_0; we must show that there is a time after which given a server r, for each event *msuspect*(P) generated by r, $q \in P$ if and only if $p \in P$. Let Θ denote the set of servers such that there is a time t_1 after which every message $\langle M, A, F \rangle$ sent by a server in Θ is such that $q \notin F[p]$. By Eventual Strong Accuracy, at least p and q belong to Θ. We claim that none of the servers not in Θ receives any message from servers in Θ sent after t_1. By contradiction, suppose a server $r \notin \Theta$ receives a message $\langle M, A, F \rangle$ sent by a server in Θ after t_1. Thus, $q \notin F[p]$ and r removes q from $fd_r[p]$ after t_1. Since q cannot be inserted again in $fd_r[p]$, there is a time after which all messages $\langle M, A, F \rangle$ sent by r are such that $q \notin F[p]$, and thus $r \in \Theta$, a contradiction. By the Eventual Symmetry and Fair Channels properties, Θ and $\Pi - \Theta$ are permanently unreachable after t_1. If $r \notin \Theta$, from (a) it follows that there is a time after which every event *msuspect*(P) generated by r is such that $p, q \in P$. If $r \in \Theta$, there are two possibilities. If r generates an event *msuspect*(P) for q after t_1 due to a change in $reachable_r$, P must contains also p (by $reachable_r$ construction). If r generates a *msuspect* event for q after t_1 due to the receipt of a $(\text{SUSPECT}, -, G)$ tuple, it is easy to see that $p, q \in G$ and so r generates a *msuspect* event for both.

(c) We must show that a server generates a *mrecv* event for a message m at most once and only if some server q actually m-sent it earlier to a set of servers containing p. Before generating a *mrecv* event for a message m m-sent by a server q, p checks that m has not been delivered yet, by verifying that the sequence number of m immediately follows the value stored in $ack_p[p][q]$; since after the first *mrecv* event, $ack_p[p][q]$ is set equal to the sequence number of m, p generates a *mrecv* event for m at most once. As regards the second part, p generates an event *mrecv*(m, q) at time t_0 only after executing an event *msend*(m, G) such that $p \in G$, or after the receipt of a message $\langle M, A, F \rangle$ such that $(\text{MSEND}, m, -) \in M[q][p]$. The first case is trivial; in the second case, there is a time $t_1 < t_0$ at which $msg_q[q][p]$ contained $(\text{MSEND}, m, -)$; this implies that q has executed an event *msend*(m, G) at time $t_2 < t_1$ such that $p \in G$.

(d) We must show that messages from the same sender are m-received in FIFO order. Let m be a message m-sent by a server q. When p has m-received m, it has set $ack_p[p][q]$ equal to the sequence number associated with it. By construction, all the messages from q m-received by p after m have a sequence number greater than the number associated with m and thus, have been m-sent after m.

(e) Let p be a correct server that m-sends at time t_0 a message m to a set of servers containing q; we must show that either (i) q will eventually generate a *mrecv* event for m, or (ii) p will eventually generate a *msuspect* event for q. By contradiction, suppose the claim is false. This implies that m will never be removed from $msg_p[p][q]$ after t_0. Let Θ denote the set of servers that never receive a message $\langle M, A, F\rangle$ such that $(\text{MSEND}, m, n) \in M[p][q]$. By definition, all servers not in Θ receive at least one message $\langle M, A, F\rangle$ such that $(\text{MSEND}, m, n) \in M[p][q]$. Since q never performs an *mrecv* event for m, q never generates an acknowledgement for m, and thus there is a time $t_1 > t_0$ after which all messages $\langle M, A, F\rangle$ sent by servers not in Θ, $\langle \text{MSEND}, m, n\rangle \in M[p][q]$ holds. By the Fair Channels property, servers in Θ are permanently unreachable from servers not in Θ after t_1. By definition, p is not in Θ. There are two possibilities: if $q \in \Theta$, by (a) it follows that there is a time after which every event *msuspect*(P) generated by p is such that $q \in P$, a contradiction. Thus, suppose $q \notin \Theta$. By hypothesis, p never m-suspects q after having m-sent m. Thus, p never inserts a $(\text{SUSPECT}, n', P)$ tuple in $msg_p[p][q]$ such that $n' > n$. The proof continues by induction on n. If $n = 1$, q must m-receive m (a contradiction), since q cannot receive a tuple SUSPECT with a value greater than n. Then, suppose $n > 1$. This implies that q cannot m-receive the message since its variable $ack_q[q][p]$ is blocked on a value $n' < n - 1$. There are two possibilities: the value $n' + 1$ is associated either with a regular message m', or to a tuple SUSPECT. In the first case, by inductive hypothesis q will m-receive m' or p will m-suspect q after having m-sent m' (and thus q will m-receive a SUSPECT tuple with a value greater than $n' + 1$); this implies that the value of $ack_q[q][p]$ will eventually increase, a contradiction. In the second case, q will m-receive a SUSPECT tuple containing $n' + 1$, and the proof terminates as in the previous case.

(f) We must show that no server generates *msuspect* events for itself. There are two cases in which a server p generates a *msuspect* event: when a change occurs in the $reachable_p$ set, in which case p generates a *msuspect* event for the set $\Pi - reachable$, which cannot contain p, or after receiving a message $\langle M, A, F\rangle$ where $(\text{SUSPECT}, -, G) \in M[q][p]$

for some server q. Since q has inserted the SUSPECT tuple in $msg[q][p]$ immediately after having excluded p from $reachable_q$ and G is equal to $reachable_q$, G does not contains p. So, p never generates a *msuspect* event for itself.

(g) Let p be a server that never generates a *msuspect* event for q after time t_0; we must show that there is a time after which q never generates a *msuspect* event for p. By contradiction, suppose this is false. By the algorithm, there is a time $t_1 > t_0$ at which $msg_q[q][p]$ contains a $(\text{SUSPECT}, n, -)$ tuple not yet received by p. Let Θ be the set of servers r such that there is a time at which $(\text{SUSPECT}, n, -) \in msg_r r[p][q]$. If $p \in \Theta$, then p eventually generates a *msuspect* event for q after t_0 as a result of the receipt of $(\text{SUSPECT}, n, -)$. If, on the other end, $p \notin \Theta$, since the sets Θ and $\Pi - \Theta$ are permanently unreachable and $q \in \Theta$, from (a) it follows that p generates a *msuspect* event for q after time t_0.

A.2 The Group Membership Layer

In this section we illustrate the correctness of the PGML algorithm presented in Section 5.4. In this section, we refer to the s-phase (ee-phase, respectively) *of a view* v to denote the s-phase (ee-phase, respectively) started *during* view v. Finally, we say that a message $\langle \text{VIEW}, w, C \rangle$ *contains* a view v if v is either equal to w or v is a partial view obtained from w and C.

First of all, we must prove that any correct server that enters s-phase will eventually install a new view. This termination property is fundamental, since will be used in the proofs of Property GM2, Property GM3 and Property GM4. The proof is divided in two parts: (i) if a correct server enters s-phase of a view v, it will eventually enter ee-phase of v itself; (ii) if a correct server enters ee-phase of a view v, it will eventually install a view after v. Since the second part is needed in the proof of the first one, we first prove (ii) and then (i).

Lemma A.1

The number of PROPOSE *messages m-sent by a server during a view is bounded.*

Proof: Apart from the PROPOSE message m-sent by p immediately after entering ee-phase of a view, a server m-sends a PROPOSE message only when it modifies its variable *estimate*.

By construction, no server may be added to *estimate* during a view after its initialization. Since the starting cardinality is bounded, the number of PROPOSE messages m-sent by a server during a view is bounded as well.

Lemma A.2

If a correct server p enters ee-phase of a view v, then p will eventually install a new view after v.

Proof: By contradiction, suppose p installs a bounded number of views and v is the last view installed by p. Let V_p denote the value of $agreed_p$ when p entered ee-phase of v. By Lemma A.1, the number of different PROPOSE messages m-sent by p during v is bounded. Let $\langle \text{PROPOSE}, s_p \rangle$ be the last PROPOSE message m-sent by p (such message exists, since p m-sends at least one PROPOSE message at the beginning of the ee-phase). Let P_p denote the set $s_p.estimate$. By Property MS1(f) and since only m-suspected servers may be removed from variable *estimate*, P_p is not empty. Since no server in P_p can be m-suspected from p during v, by Property MS1(a) every server in P_p is correct; moreover, every message m-sent by p during v to a server in P_p will eventually be m-received.

For each server $r \in P_p$, the version number $V_p[r]$ is generated at the beginning of the s-phase of a view v_r. First of all, we must prove that:

Claim: *Each server $r \in P_p$ will eventually enter ee-phase during v_r.*

By contradiction, suppose the claim is false. Thus, r is blocked in s-phase of v_r. At the beginning of ee-phase of view v, p has m-sent to r a message $\langle \text{ESTIMATE}, V_p, - \rangle$. When r m-receives it, there are two possibilities:

- If r has excluded p from its view estimate, it m-sends a SYMMETRY message to p containing $V_p[p]$; by Property MS1(g) and (e), p will eventually m-receive such a message and exclude r from its view estimate, a contradiction.
- Otherwise, r accepts the ESTIMATE message and enters ee-phase. This concludes the claim.

The next step consists on showing that all servers in P_p are blocked in ee-phase:

Claim: *Each server $r \in P_p$ is blocked in the ee-phase of its view v_r.*

By contradiction, suppose the claim is false. Thus, r installs another view after v_r. Note that p cannot m-receive a SYNCHRONIZE or a SYMMETRY message from r containing a version number array V such that $V[r] > V_p[r]$; otherwise, p would exclude r from its view estimate, a contradiction. By Property MS1(g), there is a time t_1 after which r stops m-suspecting p. There are two possibilities:

- If r installs an unbounded number of views, p will m-receive an unbounded number of different SYNCHRONIZE messages containing increasing version numbers for r, a contradiction.

- If r installs a bounded number of views, let v'_r be the last view installed by r. First of all, we must show that r will enter s-phase of v'_r. If $v'_r.comp$ does not contain p, r enters s-phase immediately after having installed v'_r, or when it stops m-suspecting p. If $v'_r.comp$ contains p, r forwards the VIEW message containing v'_r to p. There are two possibilities:

  - If this message is lost, by Property MS1(e) r will m-suspect p during v'_r and enter s-phase;

  - if p m-receives the message, by hypothesis p discards it; but this implies that the agreement on v'_r has been reached with a PROPOSE message of p m-sent before the installation of v. Thus, r will eventually enter s-phase.

  At the beginning of the new s-phase, r generates a new version number n_r greater than $V_p[r]$. If this happens before t_1, p will m-receive from r a SYMMETRY message containing n_r; otherwise, if this happens after t_1, p will m-receive from r a SYNCHRONIZE message containing n_r. In both cases, we have obtained a contradiction that concludes the claim.

Given that all servers in P_p are blocked in ee-phase, by Lemma A.1 the number of different PROPOSE messages m-sent by each server $r \in P_p$ during v_r is bounded. Let $\langle \text{PROPOSE}, s_r \rangle$ be the last PROPOSE message m-sent by r, and let P_r denote the set $s_r.estimate$.

Claim: *For each server $r \in P_p$, p belongs to P_r.*
By contradiction, suppose the claim is false. By Property MS1(g), there is a time after which r stops m-suspecting p; thus, p will m-receive a message $\langle \text{ESTIMATE}, V, P \rangle$ such that $V[r] = V_p[r]$ and $p \notin P$; so, p will remove r from its view estimate, a contradiction.

Note that the claims we have developed so far can be applied to each of the servers in P_p. Given a server r, we can show that:

- each server $s \in P_r$ installs a bounded number of views and is blocked in ee-phase of its last view v_s
- r knows the last version number of each server in P_r.

Thus, all servers in P_p know the same version number for each of the servers in P_p. By Property MS1(g), each server $r \in P_p$ will accept at least one message $\langle \text{ESTIMATE}, -, P_p \rangle$ from p and p will m-receive and accept at least one message $\langle \text{ESTIMATE}, -, P_r \rangle$. Since we have supposed that for each server $r \in P_p$, P_r is the last view estimate, then for each server $r \in P_p$ we have that $P_r \subseteq P_p$ and $P_p \subseteq P_r$. Thus, all values P_r are equal to P_p.

This implies that all servers in P_p maintain the same view estimate P_p and know the same agreed version number array (restricted to the servers in P_p). By construction, they m-send their last PROPOSE messages to the same coordinator; since no server in P_p can m-suspect another server in P_p, the coordinator will m-receive all these messages and observe an agreement. Thus, the coordinator will m-send a VIEW message to p, that m-receives it and installs a new view, a contradiction that concludes the proof.

Lemma A.3 *If a correct server p enters s-phase of a view v, then p will eventually enter ee-phase of v.*

Proof: By contradiction, suppose p never enters ee-phase of v. Under this assumption, during the s-phase no server can be added to *estimate* or removed from *synchronized* (note that servers may be added to *estimate* or removed from *synchronized* during a s-phase, but only just before entering ee-phase of v). Thus, there is a time t_1 after which p never modifies its variables $estimate_p$ and $synchronized_p$. Let n_p denote the final value of $version_p[p]$ (recall that a server generates new version numbers only when enters a new s-phase), and let q be a server contained in $estimate_p(t_1)$, but not in $synchronized_p(t_1)$ (such server must exist, otherwise p will enter ee-phase of v). By hypothesis, p cannot m-suspect q after having entered s-phase of v. By Property MS1(a), q is correct; by Property MS1(e), q will eventually m-receive the SYNCHRONIZE message containing n_p m-sent by p at the beginning of the s-phase of v and it will store n_p in $version_q[p]$ at time t_2. If q m-suspects p after t_2, by Property MS1(g) and (e) p will eventually m-receive a message $\langle$SYMMETRY$, V, -\rangle$ such that $V[p] = n_p$ and it will remove q from $estimate_p$, a contradiction. Thus, suppose q never m-suspects p after t_2. There are two possibilities:

- If q is either in idle- or in s-phase at time t_2, by construction and by Property MS1(e) p will eventually m-receive a message $\langle$SYNCHRONIZE$, n_p, -, -\rangle$; thus, p will insert q in $synchronized_p$, a contradiction.

- Otherwise, suppose q is in ee-phase at time t_2. By Lemma A.3, q will eventually install a new view w after t_2, due to the m-receipt of a message $\langle$VIEW$, w, C\rangle$. There are two possibilities.

  - If $w.comp$ does not contains p, the new view does not correspond to the current estimate of the reachability set of q (p is not m-suspected). Thus, after the installation of w, variable *stable* is set to false.
  - If $w.comp$ contains p, $C[q].version[p]$ is different from $version_q[p] = n_p$ (since p has never m-sent a PROPOSE message containing n_p). Thus, after the installation of w, variable *stable* is set to false.

  In both cases, q enters again s-phase and m-sends a SYNCHRONIZE message containing n_p to p; by Property MS1(e), p will m-receive it and add q to $synchronized_p$, a contradiction.

Corollary A.1 *If a correct server p enters s-phase of view v, then it will eventually install a new view after v.*

Proof: From Lemmas A.2 and A.3.

The next property to prove is View Accuracy. In the following, we say that a server r excludes a server s from its view estimate *indirectly* if the exclusion follows the m-receipt of a $\langle$ESTIMATE$, -, P\rangle$ message such that $r \in P$ and $s \notin P$. Otherwise, we say that r excludes a server s from its view estimate *directly* (for example, due to a m-suspect or a SYMMETRY message).

Lemma A.4 *Let v be a view installed by a server p, and let t be the time at which p entered s-phase of v. Let q be a server from which p accepts a message $\langle$ESTIMATE$, -, P\rangle$ 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 n_p. By Lemma A.3, p will eventually enter ee-phase of v. There are two possibilities:

- p enters ee-phase of v after having m-received a SYNCHRONIZE message containing n_p from each server not m-suspected after the beginning of the s-phase of v. By construction, all servers in $estimate_p$ were in s-phase when they m-sent the SYNCHRONIZE reply containing n_p. Thus, they enter ee-phase of their view after t.

- p enters ee-phase of v after having m-received a ESTIMATE message containing n_p m-sent by a server q_1 that has entered ee-phase after storing n_p in its array $version_{q_1}$. In this case, we have again two possibilities: q_1 has entered ee-phase either after having m-received a SYNCHRONIZE message from each server not m-suspected during s-phase, or after having m-received a ESTIMATE message m-sent by a server q_2 that has entered ee-phase. By iterating the reasoning, we obtain a finite chain $p, q_1, \ldots, q_n$ such that each server has entered ee-phase after having m-received a ESTIMATE message from the following one, apart from q_n that has m-received a SYNCHRONIZE message from each server not m-suspected during s-phase. Let t_i denote the time at which q_i m-sends the ESTIMATE message; by construction, we have that $estimate_{q_i}(t_i)$ is contained in $estimate_{q_{i+1}}(t_{i+1})$, for each $i = 1 \ldots n-1$. Thus, $p \in estimate_{q_i}(t_i)$, for each $i = 1 \ldots n$. Moreover, we have that $version_{q_i}[p](t_i)$ is equal to $version_{q_{i+1}}[p](t_{i+1})$, for each $i = 1 \ldots n-1$. This implies that $version_{q_n}[p]$ is equal to n_p, and that q_n has entered ee-phase of its view after t.

Theorem A.2 (View Accuracy)

The PGML algorithm satisfies Property GM2.

Proof: Let p be a correct server 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. Suppose p installs a bounded number of views; in this case, the last view installed by p must contain q (since otherwise p would enter again s-phase and by Corollary A.1 would install a new view, impossible by hypothesis). Thus, suppose p installs an unbounded sequence of views.

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

By Property MS1(b), there is a time t_1 after which each server r m-suspects both or none of p and q. Since each message $\langle$SYMMETRY$, -, P\rangle$ m-sent by a server s at time t is such that P is equal to $reachable_s(t)$, it follows that there is a time t_2 after which all messages $\langle$SYMMETRY$, -, P\rangle$ m-received by a server r are such that P contains both or none of p and q. Moreover, we must prove that there is a time t_3 after which if a server r excludes q from its view estimate due the m-receipt of a SYNCHRONIZE message, it excludes p as well. Recall that r may exclude q due the m-receipt of a message $\langle$SYNCHRONIZE$, V_p, V_s, -\rangle$ from a server s only if $agreed_r[s] < V_s$; by Property MS1(d) and by algorithm construction, this can happen only if r never m-receives an ESTIMATE or VIEW message m-sent by s before the SYNCHRONIZE message. Suppose there exists a server s from which r m-receives an unbounded number of messages $\langle$SYNCHRONIZE$, V_p, V_s, P\rangle$ inviting r to exclude q from $estimate_r$ (otherwise the claim is trivially concluded). By Property MS1(e), s will m-suspect r an unbounded number of times. Obviously, s will stop m-suspecting r an unbounded number of times as well. Thus, s will modify its variable $symset_s[r]$ after time t_1. This implies that there is a time t_3 after which all messages $\langle$SYNCHRONIZE$, -, -, P\rangle$ m-sent by s to q are such that p, q belong both or none to P. This concludes the claim.

By Eventual Symmetry, there is a time after which p is always reachable from q. By repeating the reasoning of the previous claim, we obtain that there is a time t_q after which no server different from q can directly exclude p from its view estimate without excluding q as well.

The next step consists on showing that p and q eventually stop directly excluding each other:

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

By Property MS1(f) and (b) there is a time after which p cannot m-suspect q and q cannot m-suspect p. This implies that there is a time after which (i) p cannot m-receive and accept a SYMMETRY or a SYNCHRONIZE message from a server distinct from q inviting p to exclude q (such messages cannot be m-sent to p); (ii) p cannot m-receive a SYMMETRY message from q (since they stop m-suspecting themselves); and (iii) p cannot m-receive a SYNCHRONIZE message from q with a version number for q greater than $agreed_p[q]$ (by Property MS1, this is possible only if some messages from q to p have been lost; but we know that there is a time after which p is permanently reachable from q). In a symmetric way, we can prove that q cannot directly exclude p after t_4. This concludes the claim.

By hypothesis, p will install an unbounded number of views after t_4. In order to conclude the proof, we must show that:

Claim: *There is a time t_{acc} after which all messages $\langle \text{VIEW}, -, C \rangle$ m-received and accepted by p are such that q belongs to $C[p].estimate$ (and vice versa).*
Let P_p be the set of servers that participate with p in the agreement of an unbounded number of views. This implies that all servers in P_p are correct and invoke the agreement protocol an unbounded number of times. By definition, there is a time t_5 after which p does not participate to the agreement of a view with a server not included in P_p. Consider a time t_6 at which all servers in P_p have installed at least one view after time $Max(t_4, t_5)$. Let v be a view installed by p after t_6. By Lemma A.4, all servers that participate with p in the agreement of the next view enter ee-phase of their view after t_6. By construction, all these servers have entered s-phase of their views after t_4. Thus, all messages $\langle \text{ESTIMATE}, -, P \rangle$ m-received by p during v are such that P contains both or none of p and q. This implies that p never removes q (directly or indirectly) from its view estimate during v. All $\langle \text{PROPOSE}, s \rangle$ messages m-sent by p during v are such that $q \in s.estimate$. Thus, the next message $\langle \text{VIEW}, w, C \rangle$ m-received by p is such that $q \in C[p].estimate$. This concludes the first part of the claim. In a symmetric way, we can prove that all messages $\langle \text{VIEW}, -, C \rangle$ m-received and accepted by q after t_{acc} are such that q belongs to $C[q].estimate$.

To conclude the proof, we must show that eventually each partial view installed by p contains q. Consider a message $\langle \text{VIEW}, v, C \rangle$ m-received and accepted by p; let v_p and v_q denote the values $C[p].view.id$, and let $v_p.comp$ and $v_q.comp$ denote the values $C[p].cview.comp$ and $C[q].cview.comp$. Suppose the VIEW messages containing v_p and v_q have been m-received by p and q after t_{acc}, respectively. By contradiction, suppose v_p is different from v_q; since p, q belong to both $v_p.comp$ and $v_q.comp$, at the m-receipt of $\langle \text{VIEW}, v, C \rangle$ p will install a partial view not containing q. p and q must have participated in the agreement of both v_p and v_q, but the corresponding VIEW messages must have been generated from two distinct coordinators c_p and c_q, respectively. Suppose p has m-sent the PROPOSE message for the agreement on v_p before the PROPOSE message for v_q; by monotony of variable $estimate_p$ and by the use of function *Min* to select the coordinator, c_p cannot belong to $v_q.comp$. Thus, p has not m-received the $\langle \text{VIEW}, v_p, - \rangle$ message directly from c_p, but from a server $p_1 \in v_p.comp$. Since the $\langle \text{VIEW}, v_p, - \rangle$ message has been m-received after the m-sending of the PROPOSE message for the agreement on v_q, by monotony of *estimate* server p_1 must belong to $v_q.comp$. Thus, p_1 has participated in the construction of both v_p and v_q by m-sending two PROPOSE messages to c_p and c_q in this order, and has m-received a VIEW message containing v_p after the m-sending of the PROPOSE message for the agreement on v_q. By iterating the reasoning, we obtain an unbounded chain $p_1, \ldots, p_n, \ldots$ of servers belonging to $v_p.comp$. This is a contradiction, since the cardinality of $v_p.comp$ is finite and all these servers are different (each server m-sends the same VIEW message at most once). Now, suppose p has m-sent the PROPOSE message to c_q before the PROPOSE message to c_p; by monotony of *estimate*, q must have m-sent the PROPOSE messages to c_p and c_q in the same order. But this leads to a contradiction as in the previous case and concludes the proof of the theorem.

The next property to prove is View Completeness. As the reader can note, its proof is simpler than the previous one, since if a server p m-suspects another server q, p will permanently remove q from its view estimate and no server can force p to insert q again.

Theorem A.3 (View Completeness)
The PGML algorithm satisfies Property GM3.

Proof: Let Θ be a set of servers such that there is a time after which all servers in Θ are unreachable from $\Pi - \Theta$. Let p be a correct server in Θ and let q be a server in $\Pi - \Theta$; we must prove that there is a time after which the current view of p never contains q. Suppose p installs a bounded number of views; the last view installed by p cannot contain q, otherwise p will enter again s-phase and by Corollary A.1 it will install a new view after the last. Thus, suppose p installs an unbounded sequence of views. By Property MS1(a), there is a time t_1 after which p permanently m-suspects q; this implies that after t_1, q is permanently excluded from $reachable_p$. This implies that there is a time $t_2 \geq t_1$ after which all messages $\langle \text{PROPOSE}, s_p \rangle$ m-sent by p during a view are such that $q \notin s_p.estimate$. Thus, all views installed by p after t_2 cannot contain q.

Now, we must prove that our algorithm satisfies View Order and View Integrity. Note that we do not provide a proof for Property GM1, as in our algorithm we assume that the there exist only one group and its membership is already formed when our algorithms are executed.

Theorem A.4 (View Order)

The PGML algorithm satisfies Property GM5.

Proof: Let v, w be two views. We must prove that if $v \prec^* w$, then $w \not\prec^* v$. By contradiction, suppose there exist two chains of views, $v \equiv v_1 \prec_{p_1} v_2 \prec_{p_2} \ldots \prec_{p_{n-1}} v_n \equiv w$ and $w \equiv v_{n+1} \prec_{p_{n+1}} v_{n+2} \prec_{p_{n+2}} \ldots \prec_{p_{m-1}} v_m \equiv v$. Let t_i denote the time at which the coordinator c_i generated the VIEW message containing v_i, and let t'_i be the time at which p_i installed v_i; obviously, $t_i < t'_i$. When p_i m-receives the VIEW message that contains v_{i+1}, it verifies that the agreement on v_{i+1} has been reached with information m-sent by p_i in v_i and therefore $t'_i < t_{i+1}$; thus, $t_i < t_{i+1}$. By transitivity, we can state that $t_1 < t_n$ and $t_n < t_m = t_1$, a contradiction that concludes the proof.

Theorem A.5 (View Integrity)

The PGML algorithm satisfies Property GM6.

Proof: Let p be a server. We must prove that each view installed by p contains p itself. The first view installed by p is equal to $\{p\}$. Before installing any other view v, p must m-receive a message $\langle \text{VIEW}, w, C \rangle$. By Property MS1(a), p belongs to the destination set of the VIEW message; by construction, this set coincides with $C[p].estimate$. Since v contains all servers in $C[p].estimate$ that have m-received the VIEW message containing the predecessor of v at p, then p belongs to $v.comp$.

Finally, we must prove that our algorithm satisfies the View Coherency property. First of all, we prove that a server will eventually enter s-phase if a server in its current view v never installs v. Then, we use this lemma to show that our algorithm satisfies the three parts of View Coherency.

Lemma A.5 *If a correct server p installs a view v, then for every server $q \in v.comp$ either (i) q also install v, or (ii) p will eventually enter s-phase during v.*

Proof: By contradiction, suppose q never installs v and p never enters s-phase of v. Thus, p never m-suspects q after the installation of v. By Property MS1(a), q is correct; by Property MS1(e), q will eventually m-receive the VIEW message forwarded by p before installing v. Since q does not install v, there are two possibilities: q discards the VIEW message containing v either because it has installed a new view, or because p does not belong to $estimate_q$. In both cases, q must have excluded p from $estimate_q$ after m-sending the PROPOSE message with which the agreement on v was reached; after the exclusion, q has installed a view w not containing p. By Property MS1(g), there is a time after which q stops m-suspecting p; so, q will eventually enter s-phase and m-send a SYNCHRONIZE message to p. By Property MS1(e), p will eventually m-receive the message and enter s-phase of v, a contradiction that concludes the proof.

Theorem A.6 (View Coherency)

The PGML algorithm satisfies Property GM4.

Proof: (i) Let v be a view installed by a correct server p and let q be a server in $v.comp$ that never installs v. The proof follows from Lemma A.5 and Corollary A.1.

(ii) Let v be a view installed by two servers p and q, and suppose p changes view after having installed v. If q is correct, we must prove that q will eventually install an immediate successor to v. By Corollary A.1, it is sufficient to prove that p will eventually enter s-phase of v. Suppose q stops m-suspecting p after having installed v (otherwise the proof is trivial). By Property MS1(a), p is correct; by Property MS1(g), there is a time t_1 after which p stops m-suspecting q. There are two possibilities:

- If p installs a bounded number of views, let w be the last view installed by p. By hypothesis, $v \prec^* w$. Process q must belong to $w.comp$, since otherwise p would enter again s-phase and install a new view. By Lemma A.5 and Theorem A.4, q will install w after v (since by hypothesis p does not install other views after w) and the lemma is proved.

- If p installs an unbounded number of views, q will m-receive an unbounded number of SYNCHRONIZE messages m-sent by p after t_1; this implies that q enters s-phase of v.

(iii) Let p be a server that installs a view v as well as its immediate successor w, both containing q. We must prove that p installs w only after q has installed v. Note that $w.comp$ is obtained from the last VIEW message by excluding all servers in $v.comp$ whose last view is different from v. Since q belongs to both $v.comp$ and $w.comp$, the last view of q must be equal to v.

A.3 The Reliable Multicast Layer

In this section, we prove that the algorithm presented in Section 5.5 is correct. First of all, we must prove that the changes we have done to the agreement protocol do not invalidate the proof of correctness of Section A.2. First of all, we have changed method *sendEstimate* by avoiding the sending of a the PROPOSE messages if all responses for messages multicast during the current view have not been received. In order to not invalidate the liveness lemmas, we must prove that after a server enters ee-phase for a view v, at least one PROPOSE message is sent in v. In fact, our proof is based on the assumption that we can take the last PROPOSE message sent during the view. By contradiction, suppose that no PROPOSE message is sent in v. This means that there is a server q and a message m such that m has been delivered in v, q is never removed from the current estimate and no RESPONSE message for m is ever m-received by p from q. First of all, we claim that q is correct and is also blocked in the ee-phase of v; otherwise, q would have been m-suspected or would have m-sent a SYMMETRY message to p; in both cases, it would have been removed from the current estimate, a contradiction. By algorithm construction, p has m-sent to q a MULTICAST and a DELIVERED message containing m. By Property MS1(e), q has delivered both these messages; otherwise, p would eventually m-suspect q and this would contradict our hypothesis. Thus, q has delivered m to the upper layer and has a collected a response. This response is m-sent to p through a RESPONSE message. By Property MS1(e), the RESPONSE will eventually be m-received by p, contradicting the hypothesis.

Furthermore, a new condition has been added in *checkAgreement*: before declaring the reach of an agreement, servers that survive from the previous views to the new one must have delivered the same set of messages. By contraction, suppose the an agreement protocol is blocked forever due to the fact that there are two servers q,r permanently included in the estimate for the next view maintained by a coordinator p, such that q and r have installed the same view, and there is a message m delivered by q in v, but never delivered by p. At the beginning of the ee-phase, q has m-sent a DELIVERED message to r. This message cannot be m-received by r, otherwise r would have delivered m, a contradiction. So, q m-suspects r, removes p from its estimate and m-sends a ESTIMATE message to p, asking p to remove r. By Property MS1(e), this message is eventually delivered, and r is removed from the estimate of p, contradicting the hypothesis.

Lemma A.6

A server q delivers multicast m in view v only if there is a server q that has m-sent a message $\langle \text{MULTICAST}, v, m \rangle$ in view v earlier.

Proof: If p delivers m when executing a *mcast* primitive in idle- or s-phase of view v, then p has m-sent a message $\langle \text{MULTICAST}, v, m \rangle$ in the same method. If p delivers m when m-receiving a message $\langle \text{MULTICAST}, v, m \rangle$, the lemma follows from Property MS1(e). If p delivers m when m-receiving a message VIEW, then there are two possibilities: p has executed a *mcast* primitive in the ee-phase of the previous view, or p has m-received a message $\langle \text{MULTICAST}, v, m \rangle$ in ee-phase of previous view v. In both cases, the proof is similar to the previous two points. To conclude the proof, suppose p delivers m when m-receiving a message DELIVERED m-sent by a server p_1. Server p_1 has previously delivered m, so we may apply the reasoning done so far, and say that (i) there is a server q that m-sent a message $\langle \text{MULTICAST}, v, m \rangle$ earlier, or (ii) p_1 has delivered m when m-receiving a message DELIVERED from a server p_2. In this way, we obtain a chain $p_1, p_2, \ldots$ such that each server p_i has m-received a DELIVERED message from server p_{i+1}. This chain cannot be unbounded, as the number of server in a group is bounded, and no server may appear twice in the chain (as messages cannot be delivered twice). So, there is a server q that m-sent a message $\langle \text{MULTICAST}, v, m \rangle$ earlier.

Theorem A.7 (Message Integrity)

The RML algorithm satisfies Property RM4.

Proof: We most prove that each server delivers a message only if some server actually multicast it earlier. Let p be a server that delivers a message m. By Lemma A.6, there is a server q that has executed event $msend(\langle \text{MULTICAST}, v_q, m \rangle, v_q.comp)$ such that $v_p.comp$ contains p. This implies that q has multicast message m.

Theorem A.8 (Message Uniqueness)

The RML algorithm satisfies Property RM3.

Proof: We most prove that each multicast message, if delivered at all, is delivered in exactly one view. By contradiction, suppose there is a message m that has been delivered by a server p in view v_p, and by a server q in view v_q and $v_p \neq v_q$. By Lemma A.6, there are two servers p', q' which have executed two events $msend(\langle \text{MULTICAST}, v_p, m\rangle, v_p.comp)$ and $msend(\langle \text{MULTICAST}, v_q, m\rangle, v_q.comp)$, respectively. These events are distinct, as they have been executed in two different views. By algorithm construction, these events corresponds to two different $mcast(m)$ events; this is a contradiction, as we have assumed that each multicast request is unique.

Theorem A.9 (Liveness)
The RML algorithm satisfies Property RM1.

Proof: Let p be a correct server that delivers message m in view v that includes some other server q. We must prove that if q never delivers m, then p will eventually install a new view w as the immediate successor to v. Suppose q never delivers m. By Lemma A.6, there is a server r that has m-sent a message $\langle \text{MULTICAST}, v, m\rangle$ in view v. By algorithm construction, this message has been m-sent to all servers contained in v, and thus also to v. There are two possibilities:

- If q never m-receives the MULTICAST message, then by Property MS1(e) r will m-suspect p and enter s-phase of v. By Corollary A.1, r will install a new view w after v. By Theorem A.6, p will also install a new view, as server r is included in v and has installed a new view after v. This concludes the proof of the case.

- If q m-receives the MULTICAST message, the message should have been discarded. If q never installed v, then the message cannot be delivered. But in this case, by Theorem A.6, p will install a new view after v. So, assume that q installs v. Let w the view in which the MULTICAST message has been delivered.

Theorem A.10 (Message Agreement)
The RML algorithm satisfies Property RM2.

Proof: Let v, w be two views such that $v \prec w$ and let c be the coordinator that has observed the agreement on w. Server c has m-received from each server $p \in v.comp \cap w.comp$ a message $\langle \text{PROPOSE}, s_p\rangle$ such that $s_p.buffer = M$. Every message M have been delivered by each server in $p \in v.comp \cap w.comp$ in v. By contradiction, assume that there is a server $q_1 \in v.comp \cap w.comp$ which delivers a message $m \notin M$ in v. Such delivery could have happened only after having m-sent message $\langle \text{PROPOSE}, s_{q_1} \rangle$, and thus only due to the m-receipt of a DELIVERED message from a server q_2 included in $estimate_{q_1}$ when the message has been m-received. Since during ee-phase variable *estimate* is monotone non-increasing, also q_2 must have delivered m after having sent its message $\langle \text{PROPOSE}, s_{q_1} \rangle$. By iterating this construction, we obtain an unbounded chain of distinct servers that have delivered m after having m-sent message PROPOSE with which agreement on w has been reached. This contradiction concludes the proof.

Theorem A.11 (Multicast Termination)
The RML algorithm satisfies Property RM5.

Proof: We must prove that a correct server always delivers its own multicast messages. Furthermore, we must prove that if a message is multicast in a view v before the $\overline{prepare}(v)$ event, the message will be delivered in v; otherwise, if it is multicast after the $\overline{prepare}(v)$ event, it will terminate in the successor view. If a server multicast a message in the idle or s-phase of a view v, the code associated with method mcast will locally deliver the message in the same view. If a server multicast a message in the ee-phase of a view v, a pair $(\bot, m)$ is inserted in *suspended*. Since p is correct and has entered ee-phase of view v, by Lemma A.2 p will eventually install a successor view w of v. After having installed this view, p will locally deliver all messages m' such that $(\bot, m')$ is in *suspended*. Thus, p will deliver m in view w.

Theorem A.12 (Response Integrity)
The RML algorithm satisfies Property RM6.

Proof: We must prove that each server p receives a response message only if some server actually sent it earlier to p in the same view, and that each server sends a response for a message m only if it actually delivered m earlier in the same view. The former part follows by algorithm construction and by Property MS1(c). The latter part follows directly by definition: we say that a server send a response r for a message m when r is the return value obtained by the execution of upcall dlvr(m). So, events $\overline{dlvr}(q, m, r)$ represents both the end of the dlvr upcall and the sending of the response message.

Theorem A.13 (Response Liveness)
The RML algorithm satisfies Property RM7.

Proof: Let p be a correct server that sends a response r in view v to correct server q, and suppose q never delivers r. We must prove that q will eventually install a new view w excluding p as the immediate successor to v. Since p sends a response message r to q for a message m, q must have multicast m earlier. By Theorem A.11, q locally delivers m. By Theorem A.8, q must have delivered m in v. When the execution of upcall dlvr at p terminates, p m-sends a RESPONSE message to q. Server q can never m-receive such message; otherwise, q would deliver r. By Property MS1(e), p will eventually m-suspect q and enter s-phase. By Lemma A.2, p will eventually install a successor view of v. By Theorem A.6, q will also install a new view w. By contradiction, suppose $w.comp$ contains p. This means that p has never been excluded from the estimate of q. Furthermore, we know that q has m-sent at least one PROPOSE message in order to reach an agreement on v. But this is not possible, because every server in the estimate set must have responded to all messages delivered in the previous view.

Theorem A.14 (Merging Rule)

The RML algorithm satisfies Property GMI4.

Proof: We must prove that two views merging into a common view must have disjoint compositions. By contradiction, suppose two servers q and r have installed a common view v after having installed two intersecting views v_p and v_q, respectively. The new view v could be either partial (i.e. containing only a subset of the new view composition contained in message VIEW), or total. The former case is impossible, as a partial view contains only those servers that have transmitted views with the same identifiers. Also the second case leads to a contradiction, as the a total view cannot be installed because q and r have installed different previous views.

Theorem A.15 (Integrity of Blocking Period)

The RML algorithm satisfies Property GMI5.

Proof: We must prove that when a server p installs a view w as the immediate successor to view v, then p executed event $\overline{prepare}(v)$ in v. Furthermore, events $prepare(v)$ and $\overline{prepare}(v)$ may be executed only in view v. The proof follows directly by algorithm construction. In fact, before installing a view, each server must enter ee-phase of previous one. At the beginning of ee-phase of v, upcall prepare(v) is executed, thus generating event $\overline{prepare}(v,-)$. Moreover, upcall prepare(v) corresponding to events $prepare(v)$ and $\overline{prepare}(v)$ is executed only if variable *view* is equal to v. This may happen only if the current view of p is v.

Theorem A.16 (Termination of Blocking Period)
The RML algorithm satisfies Property GMI6.

Proof: Let p be a correct server that executes an event $\overline{prepare}(v)$. We must prove that eventually p will install the successor view a v. Since p executes an event $\overline{prepare}$ in view v, p must have entered ee-phase of v. By Lemma A.2, p will eventually install a successor view of v.

A.4 The Group Method Invocation Layer

In this section, we prove that the algorithm presented in Section 5.7 is correct.

Theorem A.17 (Invocation Integrity)
The GMIL algorithm satisfies Property GMI1.

Proof: We must prove that (I1) before executing event $return(m,k,r)$, an object must have performed the corresponding invocation; (I2) each object executes event *return* for a given invocation at most once; (I3) before a server executes event $method(m,k)$, some other object (client or server) must have performed method invocation k; (I4) view installations cannot interrupt the execution of a method: given an event $method(m,k)$ executed by p at time t_1 and an event $\overline{method}(m,k,r)$ executed by p at time t_2, no $vchg(v)$ events can be executed by p in the period occurring between t_1 and t_2. (I1) and (I2) follows directly from the definition of *return* events; (I3) follows from algorithm construction and from Property MS1(c); (I4) follows from algorithm construction and from the fact that upcalls executed on server objects are synchronized, i.e. cannot be executed concurrently.

Theorem A.18 (Invocation Liveness)
The GMIL algorithm satisfies Property GMI2.

Proof: Let p be a correct server p that completes an invocation in a view v. We must prove that each server included in v will eventually complete the same invocation, or p will eventually install a new view after v. The proof is by contradiction. Assume that each server in v never completes the invocation, and v is the last view installed by p. By algorithm construction, before completing the invocation p must have delivered a message EXTERNAL or INTERNAL containing the invocation data. By Theorem A.9, every server in v must have delivered the same message. By Theorem A.3, all servers in v are correct, otherwise p would install another view after v. By algorithm construction, every server in v will eventually complete the invocation, contradicting the hypothesis.

Theorem A.19 (Invocation Agreement)

The GMIL algorithm satisfies Property GMI3.

Proof: We must prove that given two views v, w such that w is an immediate successor of v, all servers belonging to both views complete the same set of invocations in view v. Let p and q be two servers included in $v.comp \cap w.comp$. By algorithm construction, before completing an invocation, p (respectively, q) must have delivered a message EXTERNAL or INTERNAL containing the invocation data. By Theorem A.10, all servers in $v.comp \cap w.comp$ must have delivered the same message and completed the same invocation. The proof is concluded, has we have proved that p has completed all the invocations completed by q and vice versa.

Theorem A.20 (Properties Shared with RML)

The GMIL algorithm satisfies Properties GMI4, GMI5 and GMI6.

Proof: The proof follows directly from Theorems A.14, A.15 and A.15 and from the fact that vchg and prepare upcalls are forwarded directly to the member.

Theorem A.21 (Internal Invocation Termination)

The GMIL algorithm satisfies Property IGMI1.

Proof: We must prove that a correct server performing an internal invocation will eventually obtain a reply set for that invocation. If the invocation has been performed in view v before event $\overline{\text{prepare}}(v)$, it will terminate in v; otherwise, if it has been performed in view v after event $\overline{\text{prepare}}(v)$, it will terminate in the successor view. When performing the invocation, the server multicast a message containing the invocation data. By Theorem A.11, p will eventually deliver the message. The message will be delivered in v, if the the invocation has been performed in view v before event $\overline{prepare}(v)$; otherwise, it will be delivered in the successor view w. Let v' be the view in which the message has been delivered. At this point, p waits until all servers in its current view responds to the invocation message with a return value message, or a new view is installed. In the former case, p returns a set containing a return value for each of the server in $v'.comp$. Otherwise, by Theorem A.13 p will eventually install a new view, and the set of return values obtained so far is returned to p.

Theorem A.22 (Internal Invocation Validity)

The GMIL algorithm satisfies Property IGMI2.

Proof: We must prove that if the pair (p, r) is included in the reply set R obtained by a server q, then server p must have previously returned $r$through an event $\overline{method}(m, k, r)$. The proof follows from Theorem A.12 and from algorithm construction.

Theorem A.23 (Internal Invocation Non-Triviality)

The GMIL algorithm satisfies Property IGMI3.

Proof: The proof is similar to the proof of Theorem A.21.

Theorem A.24 (Internal Invocation Uniqueness)

The GMIL algorithm satisfies Property IGMI4.

Proof: We must prove that an internal invocation is completed in at most one view. This property follows directly from (i) each internal invocation is unique; (ii) an internal invocation is completed when a INTERNAL message is delivered; (iii) by Theorem A.8, the INTERNAL message will be delivered in at most one view.

A.5 The State Merging Layer

In this section, we prove that the algorithm presented in Section 5.8 is correct. In order to simplify presentation, local variables are subscripted to indicate the process to which they belong. Furthermore, we write $mset_q(v)$ and $up\text{-}to\text{-}date_q(v)$ to indicate the value of variables $mset_q$ and $up\text{-}to\text{-}date_q$ immediately after the termination of the upcall $vchg(v)$ generated by the group membership layer at member q.

Lemma A.7 *If v_1 is a view that is installed at two members p and q, then $p \in mset_q(v_1)$ if and only if $q \in up\text{-}to\text{-}date_p(v_1)$.*

Proof: Since $p \in mset_q(v_1)$, by Property IGMI1 q must have installed a sequence of views $v_n \prec_q v_{n-1} \prec_q \ldots \prec_q v_1$, such that $\forall i = 1 \ldots n : p, q \in v_i.comp$, and q must have completed in view v_n a method invocation $update(-, S)$ performed in the same view such that $q \in S$. By Properties GM4, GM5 and GMI4, p has installed exactly the same sequence of views. By Property GMI3, p has completed the same invocation in view v_n as well; thus, p has inserted q in $up\text{-}to\text{-}date_p$ during v_n. Since q belongs to all views installed by p between v_{n-1} and v_1, q cannot be excluded from $up\text{-}to\text{-}date_p$ during any of these views. Thus, q belongs to $up\text{-}to\text{-}date_p(v_1)$. The proof for the other direction is symmetric.

Lemma A.8 *Let v_1 be a view installed by two processes p and q, and let r be a server in $v_1.comp$. If $q \in mset_p(v_1)$ and $r \notin mset_p(v_1)$, then $r \notin mset_q(v_1)$.*

Proof: Since $q \in mset_p(v_1)$, by Property IGMI1 p must have installed a sequence of views $v_{n_p} \prec_p v_{n_p-1} \prec_p \ldots \prec_p v_1$, such that $\forall i = 1 \ldots n_p : p, q \in v_i.comp$, and p must have completed in view v_n a method invocation $update(-, S)$ performed in the same view such that $p \in S$. By Properties GM4, GM5 and GMI4, p has installed exactly the same sequence of views. Suppose by contradiction that $r \in mset_q(v_1)$. Thus, by Property IGMI1 q must have installed a sequence of views $v_{n_q} \prec_q v_{n_q-1} \prec_q \ldots \prec_q v_1$ such that $\forall i = 1 \ldots n_q : q, r \in v_i.comp$, and q must have completed in view v_n a method invocation $update(-, S)$ performed in the same view such that $q \in S$. There are two possibilities:

- If $v_{n_q} \prec^* v_{n_p}$ or $v_{n_q} \equiv v_{n_p}$, then we have that $\forall i = 1 \ldots n_p : r \in v_i.comp$. Thus, when p has set $mset_p$ equal to $view.comp_p$ during v_{n_p}, r has been inserted in $mset_p$ and then never removed. Therefore, $r \in mset_p(v_1)$, a contradiction.

- If $v_{n_p} \prec^* v_{n_q} \prec^* v_1$, note that server q has inserted r in $mset_q$ during v_{n_q} due the completion of an invocation $update(-, S_{n_q})$ such that $q \in Sn_q$. By Properties GMI3 and IGMI4, p has delivered the same message in view v_{n_q}. We claim that $p \in Sn_q$. Note that by Property IGMI1 the invocation $update$ due to which r

has been inserted in $mset_q$ during v_{n_q} has been performed by a server q_1 in view v_{n_q} itself. Property IGMI4, q_1 has installed v_{n_q}; by algorithm construction, $q \in$ *up-to-date*$_{q_1}(v_{n_q})$. By lemma A.7, q_1 belongs to $mset_q(v_{n_q})$. Thus, by Property IGMI1 q must have installed a sequence of views $v_{n_1} \prec_q v_{n_1 - 1} \prec_q \ldots \prec_q v_{n_q}$ such that $\forall i = n_1 \ldots n_q : q_1 \in v_i.comp$, and q must have completed in view v_{n_1} an invocation *update*$(-, S)$ performed in the same view such that $q \in S$. By Properties GM4, GM5 and GMI4, p has installed exactly the same sequence of views. Once again, there are two possibilities:

– If $v_{n_1} \prec^* v_{n_p}$ or $v_{n_1} \equiv v_{n_p}$, q_1 belongs to and has installed view v_{n_p}. By Property GMI3, q_1 has delivered in view v_{n_p} an invocation *update*$(-, S)$ such that $p \in S$; this implies that q_1 has inserted p in *up-to-date*$_{q_1}$. Since p belongs to all views installed by q_1 between v_{n_p} and v_{n_q}, p belongs to *up-to-date*$_{q_1}(v_{n_q})$; so, when q_1 performs the method invocation *update*$(-, S_{n_q})$ in view v_{n_q}, p belongs to S_{n_q} and the claim is proved.

– Otherwise, suppose $v_{n_p} \prec^* v_{n_1}$. Let q_2 be the invoker of the *update* invocation performed in view v_{n_1} due to which q_1 has been inserted in $mset_q$. By algorithm construction, q belongs to *up-to-date*$_{q_2}(v_{n_1})$; by lemma A.7, $q_2 \in mset_q(v_{n_1})$. Thus, by Property IGMI1 q must have installed a sequence of views $v_{n_2} \prec_q v_{n_2 - 1} \prec_q \ldots \prec_q v_{n_1}$ such that $\forall i = n_2 \ldots n_1 : q_2 \in v_i.comp$, and q must have completed in view v_{n_2} an invocation *update*$(-, S)$ performed in the same view such that $q \in S$. By iterating the reasoning, we obtain a set of views $v_{n_h}, v_{n_{h-1}}, \ldots, v_{n_1}$ such that $v_{n_h} \prec^* v_{n_p} \prec^* v_{n_{h-1}} \prec^* \ldots \prec^* v_{n_1} \prec^* v_{n_q}$. Since q_h belongs to v_{n_p}, p has been inserted in *up-to-date*$_{q_h}$ during v_{n_p} and it still belongs to *up-to-date*$_{q_h}$ when q_h performed the *update* invocation during $v_{n_{h-1}}$. Thus, p is inserted in *up-to-date*$_{q_{h-1}}$ during $v_{n_{h-1}}$. By iterating this construction, we can prove that the invocation *update*$(-, S_{n_q})$ performed by q_1 during v_{n_q} is such that $p \in S_{n_q}$, and this concludes the claim.

Since $p \in S_{n_q}$, r has been inserted in $mset_p$ during v_{n_q}, and then never removed. Therefore, r is in $mset_p(v_1)$, a contradiction that concludes the proof.

Lemma A.9 *Let p be a correct server and let v be a view installed by p. Let q be a process contained in $v.comp$ and in all views installed by p after v. If q does not belong to*

$mset_p(v)$, then p will eventually complete an invocation $update(-, S)$ such that $p \in S$.

Proof: The proof is by induction on the size of the variable $mset_p$. If $|mset_p(v)| = 1$, p performs a method invocation $update(-, S)$ such that $p \in S$ during the installation of v (since $mset_p(v)$ is different from $view.comp_p$ and $p = Min(mset_p(v))$). By Property GMI2, p will eventually complete the *update* invocation. This concludes the base step of the induction. Let $|mset_p(v)|$ be equal to l. Since q is in $view.comp_p$, but not in $mset_p(v)$, p elects a server c that will perform the *update* invocation on its behalf. If c does not install v, by Property GM4 p will install a new view v' excluding c; if c becomes partitioned or crashed before being able to perform the invocation, by Property GM3 p will install a new view v' excluding c. If the message multicast by c is lost, by Property GMI2 and Property GM4 p will install a new view v' excluding c. In all these cases, c will be removed from $mset_p$. Since v' and all subsequent views installed by p contains both p and q and $|mset_p(v')| \leq l - 1$, the lemma is proved by inductive hypothesis. Thus, assume c is correct, reachable and has installed v. Since $c \in mset_p(v)$ and $q \notin mset_p(v)$, by lemma A.8 $q \notin mset_c(v)$.Thus, when installing v, c assigns to a process c' the task of performing the *update* invocation. If $c' = c$, c will perform a method invocation $update(-, S)$ with $p \in S$ during the upcall of $vchg(v)$. By Property GMI2, p will eventually complete the *update* invocation. Otherwise, suppose that $c' \neq c$. By lemma A.8, c' must belong to $mset_p(v)$. But this implies that $c' = Min(\{c, c'\})$ and $c = Min(\{c, c'\})$, a contradiction that concludes the proof.

Theorem A.25 (Subview Composition)

The SML algorithm satisfies Property SMS1.

Proof: The proof follows directly by the algorithm construction and by the definition of update set.

Theorem A.26 (State Merging Integrity)

The SML algorithm satisfies Property SMS3.

Proof: We must prove that (i) a member p executes an event $getState(m, D)$ during a view v only if there is a server q in $v.comp$ not up-to-date with respect to p when p installs v; (ii) a member p executes an event $mergeState(m, S)$ only if some other member included in S has previously performed an event *getState*(m, D). As regards the former property, it is sufficient to note that variable *mset* maintained by p contains exactly those servers that are included in the current view and are up-to-date with respect to p. A state merging protocol is started if and only if this variable does not correspond to *vcomp*. The latter property, instead, follows directly by algorithm construction and by Property IGMI2.

Theorem A.27 (State Merging Agreement)

The SML algorithm satisfies Property SMS3.

Proof: We must prove that given two views v, w such that w is an immediate successor of v, all servers belonging to both views complete the same set of *mergeState* events in view v. A server executes an event *mergeState* if and only if it completes an *update* invocation. By Property GMI3, all servers belonging to two consecutive views will complete the same set of invocations, and thus will execute the same set of *mergeState* events.

Theorem A.28 (State Merging Termination)

The SML algorithm satisfies Property SMS4.

Proof: Let t_0 be a time at which a server p installs a view containing a server q. Suppose that after t_0, all views installed by p contains q; moreover, that q is not included in the update set of p at time t_0. We must prove that p will eventually execute an event *mergeState*(m, S) such that $q \in S$. By Lemma A.9, p will eventually complete an invocation *update*$(-, S)$ such that $p \in S$. During this invocation will execute an event *mergeState*(m, S).

Appendix B

The Jgroup API

This chapter contains the specification for the Jgroup application programming interface. This specification has been generated using the Javadoc tool and a Latex doclet.

B.1 Package jgroup

Interfaces in package:

Classes in package:

- JgroupException (page 205)
- McastRemoteException (page 205)
- VMID (page 211)

B.1.1 jgroup.Callback

public interface *Callback*

Callback objects are used by servers to listen to return values obtained by asynchronous internal group method invocations.

Methods

results

*public void **results**(java.lang.Object[][] values)*

Upcall to be implemented in order to receive return values obtained by an asynchronous internal group method invocation.

Parameters:

values	Array of return values obtained for the invocation; the first dimension corresponds to host, while the second corresponds to multiple servers executed on a host.

B.1.2 jgroup.Configurator

public *Configurator*
extends Object

The Configurator class contains the basic information used to configure the transport layer of the Jgroup implementation.

Fields

JGROUP_PORT

*public static final int **JGROUP_PORT***

Default Jgroup port

Constructors

Configurator

public ***Configurator****()*

Constructs a Configurator using the default Jgroup configuration parameters.

Configurator

public ***Configurator****(int port)*

Constructs a Configurator using the specified port.

Parameters:

port	The port to be used

Methods

getPort

public int ***getPort****()*

Return the port number registered in this configurator.

B.1.3 jgroup.DistributedSystem

public *DistributedSystem*
extends Object

A DistributedSystem object contains the description of the distributed system in which the distributed computation is carried on.

Constructors

DistributedSystem

public ***DistributedSystem****(java.lang.String hostfile)*

Constructs a distributed system whose description is contained in hostfile.

Parameters:

hostfile	The pathname of the file containing the description of the distributed system.

Methods

getHostfile

public String ***getHostfile***()

Returns the hostfile path.

B.1.4 jgroup.DSException

public *DSException*
extends JgroupException

Raised when an error is found in distributed system configuration.

Constructors

DSException

public ***DSException***()

Constructs a DSException with no detail message.

DSException

public ***DSException***(*java.lang.String s*)

Constructs a DSException with the specified message.

Parameters:

s	detailed message.

B.1.5 jgroup.GMFactory

public *GMFactory*
extends Object

The GMFactory class contains methods to obtain group manager object. The returned group manager depends on the interfaces implementing by the application member object.

Constructors

GMFactory

public ***GMFactory***()

Methods

getGroupManager

public static GroupManager ***getGroupManager****(jgroup.DistributedSystem ds, jgroup.Configurator conf, jgroup.Member member)*
throws JgroupException, IOException

Returns a group manager for member. The characteristics of the returned group manager depends on the interfaces implemented by member. Members implementing only Member and MergingMember will obtain a standard GroupManager object. Members implementing also the McastMember interface will obtain a McastGroupManager object. Members implementing a InternalRemote interface will obtain the corresponding group manager.

The group manager looks for a Relacs daemon running on the communication port specified in conf. If this daemon does not exists, a new daemon is created with the distributed system ds and the communication protocol configuration conf.

Parameters:

ds	the distributed system containing object members.
conf	Configuration parameters for the daemon used by the group manager.
member	the member that will be notified by the group manager whenever a view is installed or a message is delivered.

Throws:

JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

getGroupManager

public static GroupManager ***getGroupManager****(jgroup.DistributedSystem ds, jgroup.Member member)*
throws JgroupException, IOException

Returns a group manager for member. The characteristics of the returned group manager depends on the interfaces implemented by member. Members implementing only Member and MergingMember will obtain a standard GroupManager object. Members implementing also the McastMember interface will obtain a McastGroupManager object. Members implementing a InternalRemote interface will obtain the corresponding group manager.

The group manager looks for a Relacs daemon running on the default communication port for Jgroup (28771). If this daemon does not exists, a new daemon is created with the distributed system ds and the default communication protocol configuration.

Parameters:

ds	the distributed system containing object members.
member	the member that will be notified by the group manager whenever a view is installed or a message is delivered.

Throws:

JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

B.1.6 jgroup.GMHaltedException

public *GMHaltedException*
extends JgroupException

Raised when a method of a GroupManager instance is called and the GroupManager has been previously stopped.

Constructors

GMHaltedException

public ***GMHaltedException****()*

Constructs a GMHaltedException with no detail message.

GMHaltedException

public ***GMHaltedException****(java.lang.String s)*

Constructs a GMHaltedException with the specified message.

Parameters:

s	detailed message.

B.1.7 jgroup.GmiRemote

public interface *GmiRemote*
implements Remote

The GmiRemote interface serves to identify external remote interfaces containing methods that may be invoked on groups of remote objects.

B.1.8 jgroup.GroupManager

public interface *GroupManager*

In order to access the services provided by Jgroup, each member need a GroupManager object. Group managers can be obtained using the getGroupManager methods of the GMFactory class. Each group manager is associated with one and only one member; it instantiates a thread whose task is to issue upcall methods contained in the Member interface. Members must use the methods of their group manager to join and leave groups.

Methods

getLocalId

*public int **getLocalId**()*

Returns the local identifier of the member. A local identifier is unique inside a Java virtual machine.

halt

*public void **halt**()*
throws GMHaltedException

Stops the group manager.

Throws:

GMHaltedException if the GroupManager has been halted.

join

*public void **join**(java.lang.Object gdata, int gid)*
throws GMHaltedException, JgroupException

Request to join a group. After having requested to join a group, a member is not allowed to multicast messages for the group until it will receive the first view notification, that should be interpreted as an acknowledgement that the member has joined the group.

Parameters:

gdata	User-defined data structure associated with the group; whenever the member will receive an upcall for this group, it will receive a reference to this data structure. May be used by servers to easily retrieve information associated with the group.

gid	Group identifier.

Throws:

GMHaltedException	if the GroupManager has been halted.
JgroupException	if the group manager receives a request to join a group when it is already member of a group.

leave

*public void **leave()***
throws GMHaltedException, JgroupException

Request to leave a group. Note that an object member may receive event notifications such as message deliveries and view installation for a group even after having request to leave the group itself. However, the group will eventually leaved and the members will receive a leave() notification. After this event, the member will not receive any event notification related to the group.

Throws:

GMHaltedException	if the GroupManager has been halted.
JgroupException	If the group manager is not member of any group, or is already leaving its current group.

B.1.9 jgroup.IID

public *IID*
extends Object
implements Serializable

A IID is an object that uniquely identifies group method invocations. It is composed by the identifier of the virtual machine that has generated the invocation, plus a progressive counter used to distinguish among invocations performed in that virtual machines.

Methods

getCounter

*public int **getCounter()***

Returns the progressive counter of this invocation.

getVMID

public VMID ***getVMID****()*

Returns the identifier of the virtual machine that has generated this invocation identifiers.

B.1.10 jgroup.JgroupException

---

public *JgroupException*
extends Exception

Base class for the exceptions thrown by Jgroup.

Constructors

JgroupException

public ***JgroupException****()*

Constructs a JgroupExeception with no detail message.

JgroupException

public ***JgroupException****(java.lang.String s)*

Constructs a JgroupExeception with the specified message.

Parameters:

s	detailed message.

B.1.11 jgroup.McastRemoteException

---

public *McastRemoteException*
extends RemoteException

Tag exception used to identify methods that have to be invoked through the multicast semantics.

Constructors

McastRemoteException

public ***McastRemoteException****()*

Constructs a McastRemoteException with no detail message.

McastRemoteException

public ***McastRemoteException****(java.lang.String s)*

Constructs a McastRemoteException with the specified message.

Parameters:

s	detailed message.

B.1.12 jgroup.Member

public interface *Member*

Java objects that want to become member of a group of objects must implement the Member interface. The methods contained in the interface, like dsChange() and vChange(), are used to receive membership change notifications. Members must be able to react in a suitable way to such events.

It is important to note that every event notification method has an user-dependent object as a parameter. This object is set when joining the group and may contain user data structures related to the group.

Methods

dsChange

public void ***dsChange****(java.lang.Object groupdata, java.net.InetAddress[] hosts, int[] oldpos, int index)*

Upcall that is invoked by Jgroup when a change occur in the distributed system in which a distributed computation is running. Currently, this method is invoked only once, when a member joins a group. Eventually, it will be possible to modify the distributed system by adding or removing hosts.

The new distributed system is described by two arrays of the same length: the first array contains a list of Internet addresses composing the new distributed system, while the second array is a list of integers values describing the position of an host with respect to the previous distributed system. A negative value means that the host was not present in the new distributed system. As an example, suppose a distributed system composed of two machines { 130.136.2.2, 130.136.2.15} is substituted with a distributed system composed of { 130.136.2.2, 130.136.2.3, 130.136.2.15 }. The corresponding array of old positions is { 0, -1, 1 }, meaning that the first host has maintained the same position, the second host was not present in the previous distributed system and the third host was in position 1 (the second in an array) in the previous distributed system.

Parameters:

groupdata	user structure containing information associated with the group.
hosts	the array of IP addresses of the hosts composing the new distributed system.

oldpos	array of the old positions
index	my old position

leave

*public void **leave**(java.lang.Object groupdata)*

Upcall that is invoked by Jgroup to acknowledge the fact that an object is not member of a group any more:

Parameters:

groupdata	user structure containing information associated with the group.

vChange

*public void **vChange**(java.lang.Object groupdata, jgroup.View view)*

Upcall that is invoked by Jgroup when a view change occur in the group identified by gid.

Parameters:

groupdata	user structure containing information associated with the group.
view	the new view.

vPrepare

*public void **vPrepare**(java.lang.Object group)*

Upcall that is invoked by Jgroup to notify that an agreement protocol on a new view is going to be executed.

Parameters:

group	

B.1.13 jgroup.MergingMember

public interface *MergingMember*
implements Member

Interface MergingMember extends interface Member by adding two new event notification methods. The first is an extension of the normal install method, and is invoked when the member has been selected as coordinator in the reconciliation protocol. The coordinator has to prepare a status message containing the information needed to members that need to be updated about the status of the partition of the coordinator. The second, called putStatus, is invoked to inform the member about the status of other members belonging to different partitions.

Methods

mergeState

public void ***mergeState****(java.lang.Object group, java.lang.Object status, int[] sources)*

Upcall that is invoked by Jgroup to notify members of a group of reconciliation information coming from other servers.

Parameters:

group	user structure containing information associated with the group.
status	the reconciliation message
sources	members contained in this arrays know all the information contained in status.

vChange

public Object ***vChange****(java.lang.Object group, jgroup.View view, int[] dests)*

The invocation of a vChange method notifies the server that a new view has been installed. Furthermore, the server has been selected as coordinator and is required to return an object containing the information needed to update the servers included in the dests array.

Parameters:

group	user structure containing information associated with the group.
view	the new view.
dests	the destination set of members for the reconciliation message;

Returns:

the reconciliation message

B.1.14 jgroup.RemoteGroupManager

public interface *RemoteGroupManager*

Methods

getIdentifier

public IID ***getIdentifier****()*

If invoked during the execution of an external invocation, this method returns the identifier of the invocation.

join

*public void **join**(int gid, java.lang.Object groupdata, java.lang.String name) throws JgroupException, GMHaltedException*

Used to join a group. Differently from join methods that may be found in interface GroupManager, join methods of a remote group manager have to take into consideration that the external invocations on the group may be performed by clients. While the group identifier is used internally by servers to identify the group, clients normally use names as identifiers in registry services. For this reason, this version of join registers the server on the default dependable registry under name name.

Parameters:

gid	Group identifier.
groupdata	
name	name under which the group will be registered in the registry service.

Throws:

GMHaltedException	if the GroupManager has been halted.
JgroupException	if the group manager receives a request to join a group when it is already member of a group.

join

*public void **join**(int gid, java.lang.Object groupdata, java.lang.String name, jgroup.registry.DependableRegistry drs) throws JgroupException, GMHaltedException*

Used to join a group. Differently from join methods that may be found in interface GroupManager, join methods of a remote group manager have to take into consideration that the external invocations on the group may be performed by clients. While the group identifier is used internally by servers to identify the group, clients normally use names as identifiers in registry services. For this reason, this version of join registers the server on the dependable registry drs under name name.

Parameters:

gid	Group identifier.
groupdata	
name	name under which the group will be registered in the registry service.
drs	dependable registry on which register the group.

Throws:

GMHaltedException	if the GroupManager has been halted.

JgroupException	if the group manager receives a request to join a group when it is already member of a group.

B.1.15 jgroup.View

public interface *View*

The Partitionable Group Membership Service contained in Relacs tracks changes in the group's composition and installs them as views at member objects. Installed views are an abstraction of the current environment with respect to failures and recoveries. Member objects can obtain information about the current view through the View interface.

Methods

getGid

*public int **getGid()***

Returns the identifier of the group in which this view has been generated.

getHostIndex

*public int **getHostIndex()***

Returns the position of the local host in the array of hosts.

getHosts

*public InetAddress[] **getHosts()***

Returns the set of hosts participating in this view. The hosts are identified by their Internet address. A host participates in a view if and only if it contains members participating in that view.

getIncarnations

*public int[] **getIncarnations()***

Returns the incarnation numbers of the hosts composing this view. If the incarnation number of an host included in this view is greater than the incarnation number of the same host received with a previous view, the daemon running in that host has crashed, together with the associated member objects.

getIndexArray

*public int[] **getIndexArray()***

Returns the index of the hosts participating in this view, with respect to the array of hosts delivered with the last dsChange event. For example, if the last dsChange event contains the array { 130.136.2.2, 130.136.2.3, 130.136.2.15 }, a view object containing only the hosts identified by 130.136.2.2 and 130.136.2.15 returns an array { 0, 2 }.

getLocalMembers

public int[] ***getLocalMembers()***

Returns the set of local members participating in this view. Members are identified by an integer value. The method returns an array containing all the identifiers of the members of the local host.

getMembers

public int[][] ***getMembers()***

Returns the set of members participating in this view. Members are identified by an integer value. The method returns an array containing an item for each of the host participating in the view. Each item is an array containing all the identifiers of the members of that host.

getVid

public long ***getVid()***

Returns the identifier of this view.

B.1.16 jgroup.VMID

public *VMID*
extends Object
implements Serializable

A VMID is an identifier that is unique across all Java virtual machines. VMIDs are used by Jgroup to identify client VMs performing external invocations. A VMID is composed of an IP address plus an incarnation number. Class jgroup.VMID is similar to java.rmi.dgc.VMID; but jgroup.VMID enables other objects to inspect the fields composing it.

Methods

getAddress

public InetAddress ***getAddress()***

Returns the IP address of the machine hosting the Java virtual machine.

getIncarnation

public long ***getIncarnation()***

Returns the incarnation number needed to distinguish JVMs running in the same host.

getVMID

public static VMID ***getVMID()***

Returns the VMID for the local Java virtual machine.

getVMID

*public static VMID **getVMID**(jgroup.VMID id)*

Returns a new VMID identifier identical to id

Parameters:

id	the VMID identifier to be copied.

B.2 Package jgroup.registry

Interfaces in package:

Classes in package:

B.2.1 jgroup.registry.DependableRegistry

public interface *DependableRegistry*
implements GmiRemote

For obtaining references to remote objects, RMI includes a registry service, that provides methods for storing and retrieving remote object references. Each registry object acts independently and maintains a different set of bindings (name, remote object), thus constituting a single point of failure. Furthermore, a registry object running on a certain host can be used to register only remote objects running on the same host; thus, in order to obtain a stub for a remote object, a client need to be aware of its location.

The registry implementation included with Jgroup has several advantages over the standard Java RMI registry. First, it enables a set of replicated remote objects to register themselves under the same name. This set forms a remote object group that can be retrieved as a single entity. All the Jgroup registry objects running in a distributed system cooperate in order to maintain a replicated database of bindings and offer a reliable and high-available repository facility. Clients no longer need to be aware of the location of a service: they can simply access the group of registry replicas and obtain a stub for every service registered in the distributed system. A stub for the registry group is obtained through the RegistryFactory.getDependableRegistry methods.

A distributed system can be designed by including a certain number of registry replicas running on different hosts and possibly on distinct portions of the communication network. Client access these replicas through standard RMI interactions as if they were a single registry, and are guaranteed that their invocation will successfully terminate, provided that at least one operational replica is running in their partition.

During a partitioning, the Jgroup registry presents a partitioned behavior reflecting the current failure scenario. The registration of a remote object inside a partition will not affect the registry replicas not contained in that partition, while a retrieval operation will not success if the service we are looking for has been registered in another partition. Nevertheless, the replicas contained in a partition consistently maintain the same set of bindings and act as a single entity; moreover, the disappearance of the partitioning causes the execution of a reconciliation protocol in order to re-establish a consistent set of bindings among the replicas that belonged to different partitions. It is important to note that this behavior is perfectly reasonable for a partitionable distributed system, since clients asking for remote services are interested only in servers running in their current partitions.

The DependableRegistry interface describes the methods of a dependable registry that can be remotely invoked. To create a registry that runs in an application, use one of the RegistryFactory.createDependableRegistry methods. To obtain a reference to a remote object registry, use one of the RegistryFactory.getDependableRegistry methods.

Fields

DEPENDABLE_REGISTRY_NAME

public static final String ***DEPENDABLE_REGISTRY_NAME***

Service name for the dependable registry.

DREGISTRY_BOOTSTRAP_PORT

public static final int ***DREGISTRY_BOOTSTRAP_PORT***

Port number used by the dependable registry for communication among servers registry in a remote machine.

DREGISTRY_JGROUP_PORT

public static final int ***DREGISTRY_JGROUP_PORT***

Port number used by the dependable registry for communication among servers registry in a remote machine.

Methods

bind

public IID ***bind****(java.lang.String name, java.rmi.Remote obj)*
throws RemoteException, McastRemoteException, AccessException, JgroupException, IOException

Add a replicated remote object to the remote object group associated with the specified name in the registry.

Parameters:

name	the name associated with the remote object group

obj	the replicated remote object to be added

Returns:
a BID identifier used to (possibly) remove this replicated remote object from the group

Throws:

RemoteException	Raised when remote operation failed.
AccessException	Raised when this operation is not permitted.
JgroupException	Generic exception related to Jgroup.
IOException	Raised when an I/O error occurs.

list

public String[] ***list()***
throws RemoteException, AccessException

Returns an array containing the remote object group names registered in the registry.

Throws:

RemoteException	If remote operation failed.
AccessException	If this operation is not permitted.

lookup

public Remote ***lookup****(java.lang.String name)*
throws RemoteException, NotBoundException, AccessException

Returns a stub for the remote object group associated with the specified name in the registry.

Parameters:

name	the name associated with the remote object group

Returns:
a stub for the remote object group

Throws:

RemoteException	If remote operation failed.
NotBoundException	if there is no object group with this name in the registry.
AccessException	If this operation is not permitted.

unbind

*public void **unbind**(jgroup.IID bid)*
throws RemoteException, McastRemoteException, NotBoundException, AccessException, JgroupException, IOException

Remove a replicated remote object identified by the BID object obtained when the replica has been registered.

Parameters:

bid	the binding identifier obtained when the replica has been registered

Throws:

RemoteException	If remote operation failed.
NotBoundException	if there is no object with this name in the registry.
AccessException	If this operation is not permitted.
JgroupException	Generic exception related to Jgroup.

B.2.2 jgroup.registry.RegistryFactory

public final *RegistryFactory*
extends Object

The RegistryFactory class is used to obtain a bootstrap reference to the Jgroup dependable registry, or to create a Jgroup dependable registry instance on the local machine. Every instance of the Jgroup registry service is based on two remote registries: the standard local registry and the replicated one. The standard local registry is used to retrieve a stub for the local instance of the replicated remote registry. When the argument rport is used in a getRegistry or a createRegistry method, it specifies the port used by the local registry. The port used by the replicated registry is identified by rport+1 .

Methods

createRegistry

*public static DependableRegistry **createRegistry**(jgroup.DistributedSystem ds)*
throws RemoteException, JgroupException, IOException

Creates and exports a Jgroup registry instance on the local host that listen to incoming requests on an anonymous communication port. The registry instance joins the group of Jgroup registries on the distributed system described by ds, using the default communication port for Jgroup (28771).

Parameters:

ds	distributed system description

Returns:
the new Jgroup registry instance

Throws:

RemoteException	if the registry could not be exported
JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

createRegistry

*public static DependableRegistry **createRegistry**(jgroup.DistributedSystem ds, jgroup.Configurator conf)*
throws RemoteException, JgroupException, IOException

Creates and exports a Jgroup registry instance on the local host that listen to incoming requests on an anonymous communication port. The registry instance joins the group of Jgroup registries on the distributed system described by ds, using the communication port identified by jport and the Jgroup configuration specified in conf.

Parameters:

ds	distributed system description
conf	configuration information

Returns:
the new Jgroup registry instance

Throws:

RemoteException	if the registry could not be exported
JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

createRegistry

*public static DependableRegistry **createRegistry**(jgroup.DistributedSystem ds, jgroup.Configurator conf, int rport)*
throws RemoteException, JgroupException, IOException

Creates and exports a Jgroup registry instance on the local host that listen to incoming requests on the given rport communication port. The registry instance joins the group of Jgroup registries on the distributed system described by ds, using the communication port identified by jport and the Jgroup configuration specified in conf.

Parameters:

ds	distributed system description
conf	configuration information
rport	port used for RMI communications

Returns:
the new Jgroup registry instance

Throws:

RemoteException	if the registry could not be exported
JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

createRegistry

*public static DependableRegistry **createRegistry**(jgroup.DistributedSystem ds, jgroup.Configurator conf, int rport, java.rmi.server.RMIClientSocketFactory csf, java.rmi.server.RMIServerSocketFactory ssf)*
throws RemoteException, JgroupException, IOException

Creates and exports a Jgroup registry instance on the local host that listen to incoming requests on the given rport communication port. The registry instance joins the group of Jgroup registries on the distributed system described by ds, using the communication port identified by jport and the Jgroup configuration specified in conf. The new instance listens to incoming requests using a ServerSocket created from the supplied RMIServerSocketFactory. A client that receives a reference to this registry will use a Socket created from the supplied RMIClientSocketFactory.

Parameters:

ds	distributed system description
conf	configuration information
rport	port used for RMI communications
csf	client-side Socket factory used to make connections to the registry
ssf	server-side ServerSocket factory used to accept connections to the registry

Returns:
the new Jgroup registry instance

Throws:

RemoteException	if the registry could not be exported
JgroupException	Generic exception related to Jgroup.
DSException	raised when the distributed system configuration object contains an error
IOException	if an I/O error occurs when creating communication sockets.

getRegistry

public static DependableRegistry ***getRegistry****(jgroup.DistributedSystem ds) throws RemoteException, IOException*

Returns a reference to the group of dependable registry replicas running in in the specified distributed system.

Parameters:

ds	the distributed system in which the dependable registry replicas are expected to run

Returns:
a reference (stub) to the Jgroup dependable registry

Throws:

RemoteException	if the reference could not be obtained

getRegistry

public static DependableRegistry ***getRegistry****(jgroup.DistributedSystem ds, int rport) throws RemoteException, IOException*

Returns a reference to the group of dependable registry replicas running in in the specified distributed system and listening on the port specified by rport.

Parameters:

ds	the distributed system in which the dependable registry replicas are expected to run
rport	port used by the replicas for RMI communications

Returns:
a reference (stub) to the Jgroup dependable registry

Throws:

RemoteException	if the reference could not be obtained

getRegistry

public static DependableRegistry ***getRegistry****(jgroup.DistributedSystem ds, int rport, java.rmi.server.RMIClientSocketFactory csf)*
throws RemoteException, IOException

Returns a reference to the group of dependable registry replicas running in in the specified distributed system and listening on the port specified by rport. Communication with this remote registry will use the supplied RMIClientSocketFactory csf to create socket connections.

Parameters:

ds	the distributed system in which the dependable registry replicas are expected to run
rport	port used by the replicas for RMI communications
csf	client-side Socket factory used to make connections to the registries. If csf is null, then the default client-side Socket factory will be used in the registry stub.

Returns:
a reference (stub) to the Jgroup dependable registry

Throws:

RemoteException	if the reference could not be obtained

Acknowledgements

Ok. Here we are. The dissertation is completed, and it is time to thank people who helped me, in a way or in another, to write it. The first decision I have to take is: Italian or English? My first answer was English, in order that everybody may understand it. But then I realized that all people I want to thank are italian or are able to understand Italian, and this section is dedicated to them... So, I'm sorry for readers that cannot understand it, but the rest of this section is written in Italian.

La prima persona che devo ringraziare è ovviamente Ozalp Babaoglu, il mio supervisore. Spesso è impossibile capire dove si sta andando mentre si è in cammino, nella vita come nella ricerca. Negli ultimi quattro anni, Ozalp mi ha fatto da guida in questo cammino. Mi ha indicato la strada da seguire, mi ha spronato quando mi sono fermato, e mi ha consigliato quando la strada scelta si è rivelata sbagliata. E' grazie al suo aiuto, ed a tutte le responsabilità che mi ha affidato, se sono stato in grado di scegliere da solo la strada che ha portato a Jgroup.

Se Ozalp mi ha indicato il cammino, Renzo Davoli mi ha pazientemente accompagnato. Il suo aiuto in questi anni è stato fondamentale nel risolvere i problemi che la nostra ricerca mano a mano presentava. Non solo, Renzo in questi anni mi è stato molto vicino anche umanamente. Grazie!

Una parola di ringraziamento va ovviamente ai referee che hanno pazientemente letto e giudicato la mia tesi: Rachid Guerraoui e Keith Marzullo. I loro commenti sono stati preziosi per migliorare sia il contenuto che la forma della tesi.

Scientificamente parlando, voglio ringraziare anche Sape Mullender, che mi ha ospitato per tre magnifici mesi all'Università di Twente. La nostra collaborazione, incentrata su argomenti diversi da quelli di questa tesi, non ha prodotto risultati di rilievo. Ma Sape e il suo gruppo di ricerca mi hanno insegnato un concetto molto importante: la ricerca è fondata sulla curiosità di sapere. Senza la capacitị porsi domande, è inutile cercare le risposte.

Tutte queste persone hanno avuto un ruolo più o meno determinante nella produzione di questa tesi. Ma in questi quattro anni, altri mi sono stati vicino e il mio riconoscimento va anche a loro. Innanzitutto i miei colleghi dottorandi, dai dottorandi-x ai dottorandi-xiv (più una dottoranda-ix che dispone di un ufficio proprio, ma non può fare a meno di stare sempre nel dotto-lab!). Un grazie va anche a tutte le persone del dipartimento che mi hanno aiutato, in particolare il sistemista Amico... vedi per esempio l'installazione di un modulo fronte-retro il giorno stesso di consegna delle tesi!

Alcuni amici mi sono stati accanto negli ultimi anni: Alessandro, Anna, Chiara, Damiano, Dario, Elena, Gianluca, Nicolas, Piero, Riccardo, Serena e Stefano (in ordine alfabetico, così non favorisco nessuno). I fine settimana trascorsi con loro mi hanno dato la serenità per affrontare il mio lavoro.

Come ogni buon italiano, a questo punto devo ringraziare mia madre. Supporto incondizionato, amore smisurato e pranzetti deliziosi: che altro potrei chiedere?

Per ultima Alice, anche se lei per me è sempre la prima. Che posso dire? Lei crede in me, e io ne traggo forza per affrontare questa e altre sfide. Questa tesi è dedicata a te.