Jgroup Tutorial and Programmer's Manual

Alberto Montresor [*] Hein Meling [‡]

February 2002

Abstract

This tutorial gives a brief introduction to Java Remote Method Invocation, an overview and some details of the services provided by the Jgroup Group Communication System and explains how to develop distributed applications using Jgroup. It also gives a brief overview of the Autonomous Replication Management framework.

1 Introduction

Building distributed applications is a complex task and in recent years the emergence of programming environments to simplify development of distributed applications have been introduced. These programming environments are often refered to as a *middleware* between the application itself and the operating system. Middleware provides various services that applications can use to provide end-user services. There are several types of middleware that support different approaches, such as computational middleware and message oriented middleware. This tutorial cover only Java Remote Method Invocation (RMI) [17], and in particular the Jgroup Group Communication System (GCS), which can be classified as a computational middleware. Computational middleware is characterized by transparency at the level of method invocation (function call), and this approach is also used in CORBA [12]. Java RMI is also the technology used in Jini [4] and J2EE [18].

In order to abstract the complexity of distributed systems and to promote modularity and reusability, most middleware platforms are based on object-oriented concepts like abstraction, encapsulation, inheritance and polymorphism. Furthermore, they enable client/server interactions among distributed objects: server objects encapsulate an internal state and make it accessible through a set of well-defined interfaces; 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, often refered to

[*]Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy), Email: montresor@CS.UniBO.IT

[‡]Department of Telematics, Norwegian University of Science and Technology, O.S. Bragstadsplass 2A, N-7491 Trondheim (Norway), Email: meling@item.ntnu.no

as *stubs*, that deal with all low-level details of an invocation, such as communication and marshalling of invocation parameters.

Existing object-oriented middleware environments focus their attention on improving portability, interoperability and reusability of distributed software components and applications. Unfortunately, none of them provide adequate support for the development of dependable applications in the presence of partial failures. The main problem is the lack of "one-to-many" (multicast) 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 [11]. In the absence of any systematic support, building applications able to deal with partial failures such as crashes and network partitionings is an error-prone and time-consuming task.

In an effort to fill this void, the *object group* paradigm has been proposed [10]. 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* [9, 13, 19, 1, 7], 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 Revisiting Java Remote Method Invocation

The purpose of providing a framework for remote method invocation is primarily to make the programming style for building distributed systems, as much as possible, like the one for building non-distributed systems. That is, to make a remote method invocation appear as if it is just a local method invocation. The reasoning behind this is that working with distributed systems is a complex task, even without having to thing about message exchange.

The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. RMI provides for remote communication between programs written in the Java programming language.

The main difference between RMI and local invocations is that special handling is required to obtain object references for the remote objects. This is typically accomplished using a naming service. A naming service is a very simple database of name-to-object reference mappings.

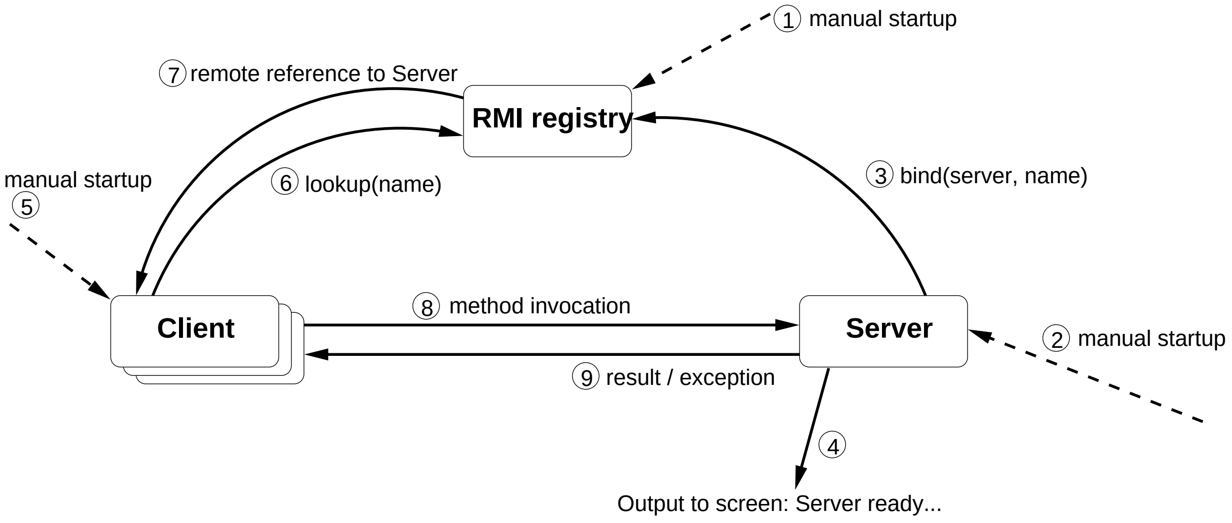

Figure 1: Overview of interactions in the RMI model.

Java RMI Architecture Overview

Figure 1 gives an overview of the workings of Java RMI, describing the sequence in which tasks are typically executed. As mentioned above, to run a distributed service we need to have a naming service, also called a registry. Initially, the RMI registry is thus started manually (①), followed by the starting of the server (②). During its initialization phase, the server will register (bind) its remote object reference (*(Remote) this*) with the RMI registry (③), associating the server with a *name*. After the initialization phase (④), the server will simply wait for clients to issue remote method invocations on it. Assuming that a server has been able to register its reference in the registry, a client can be started (⑤) and it can perform a lookup (⑥) operation, using the *name* of the service to obtain a reference (⑦). Given a reference to a remote object, as with a local object, the client can then perform method invocations on the object (⑧), and obtain results (⑨).

2 Jgroup Overview

The Jgroup toolkit integrates object group technology and distributed objects based on Java RMI [17]. In Jgroup, client objects interact with an object group implementing some distributed service through an *external group method invocation* (EGMI) facility, as shown in Figure 2. Jgroup hides the fact that services may be implemented as object groups rather than single objects so that clients using them through EGMI need not be reprogrammed. Servers making up the object group cooperate in order to provide a dependable version of the service to their clients. This cooperation has to maintain the consistency of the replicated service state and is achieved through an *internal group method invocation* (IGMI) facility. Strong guarantees provided by Jgroup for both EGMI and IGMI in the presence of

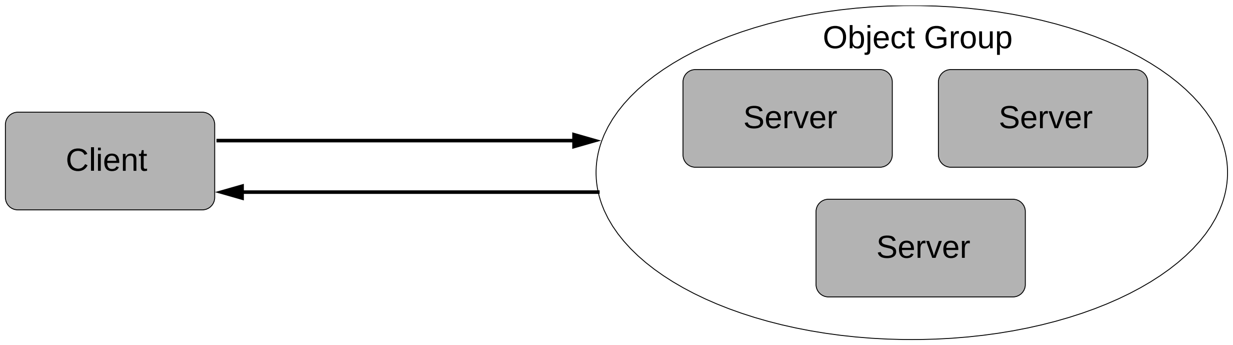

Figure 2: Client to server object group communication using EGMI.

failures and recoveries (including partitioning and merging of the communication network) greatly simplify the task of application developers.

Jgroup includes numerous innovative features that make it interesting as a basis for developing modern network services:

- It exposes network effects to applications, which best know how to handle them. In particular, operational objects continue to be active even when they are partitioned from other object group members. This is in contrast to the *primary partition* approach, that hides as much as possible network effects from applications by limiting activity to a single *primary* partition while blocking activity in all other partitions. An important property of Jgroup is providing each object a consistent view of all other objects that are in the same partition as itself. This knowledge is essential for *partition-aware* application development where the availability of services is dictated by application semantics alone and not by the underlying system.

- In Jgroup, all interactions within an object group implementing some service and all requests for the service from the outside are based on a single mechanism — remote method invocations. Jgroup is unique in providing this uniform object-oriented interface for programming both servers and clients. Other object group systems typically provide an object-oriented interface only for client-server interactions while server-server interactions are based on message passing. This heterogeneity not only complicates application development, it also makes it difficult to reason about the application as a whole using a single paradigm.

- Jgroup includes a *state merging service* as systematic support for partition-aware application development. Reconciling the replicated service state when partitions merge is typically one of the most difficult problems in developing applications to be deployed in partitionable systems. This is due to the possibility of the service state diverging in different partitions because of conflicting updates.

- The Autonomous Replication Management (ARM) [14] framework simplifies building replicated distributed applications. This is achieved through a replication manager that ensures to maintain specified replication levels, even in presence of failures.

3 Jgroup Specification

The Jgroup toolkit is composed by three integrated facilities: the *partition-aware group membership service* (GMS), the *group method invocation service* (GMI service) and the *state merging service* (SMS). In this section, we informally specify their behavior. The formal specification may be found in a companion work [16].

3.1 The Partition-aware Group Membership Service

Groups are collections of server objects that cooperate in providing distributed services. For increased flexibility, the group composition is allowed to vary dynamically as new servers are added and existing ones removed. Servers desiring to contribute to a distributed service become a *member* of the group by *joining* it. Later on, a member may decide to terminate its contribution by *leaving* the group. At any time, the *membership* of a group includes those servers that are operational and have joined but have not yet left the group. Asynchrony of the system and possibility of failures may cause each member to have a different perception of the group's current membership. The task of a PGMS is to track voluntary variations in the membership, as well as involuntary variations due to failures and repairs of servers and communication links. All variations in the membership are reported to members through the *installation* of *views*. Installed views consist of a membership list along with a unique view identifier, and correspond to the group's current composition as perceived by members included in the view.

A useful PGMS specification has to take into account several issues. First, the service must track changes in the group membership accurately and in a timely manner[1] such that installed views indeed convey recent information about the group's composition within each partition. Next, we require that a view be installed only after agreement is reached on its composition among the servers included in the view. Finally, PGMS must guarantee that two views installed by two different servers be installed in the same order. These last two properties are necessary for server objects to be able to reason globally about the replicated state based solely on local information, thus simplifying significantly their implementation. Note that the PGMS we have defined for Jgroup admits co-existence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications.

[1]Being cast in an asynchronous system, we cannot place time bounds on when new views will be installed in response to server joins, leaves, crashes, recoveries or network partitionings and merges. All we can guarantee is that new view installations will not be delayed indefinitely.

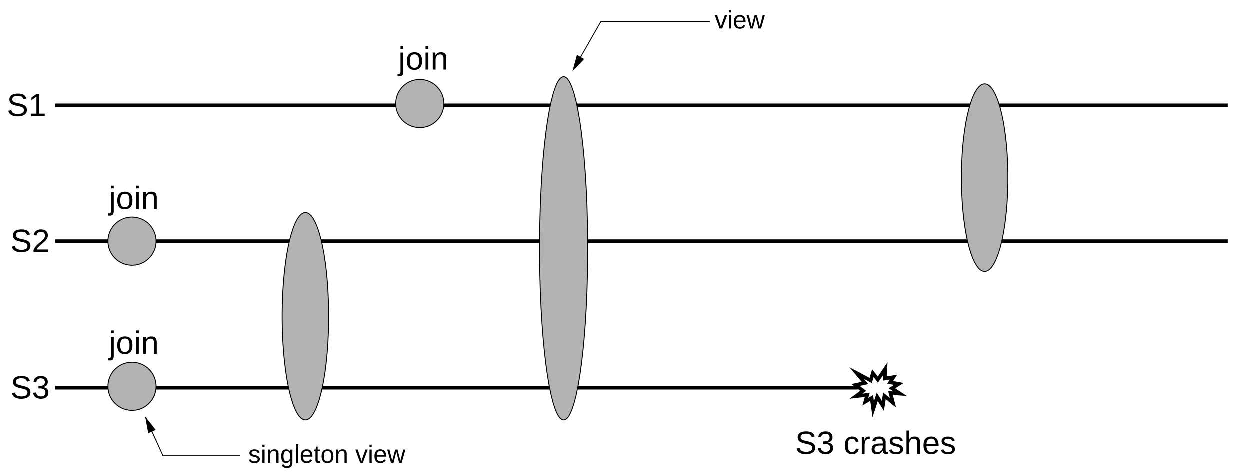

Figure 3: Overview of how the membership service works with respect to crash failures.

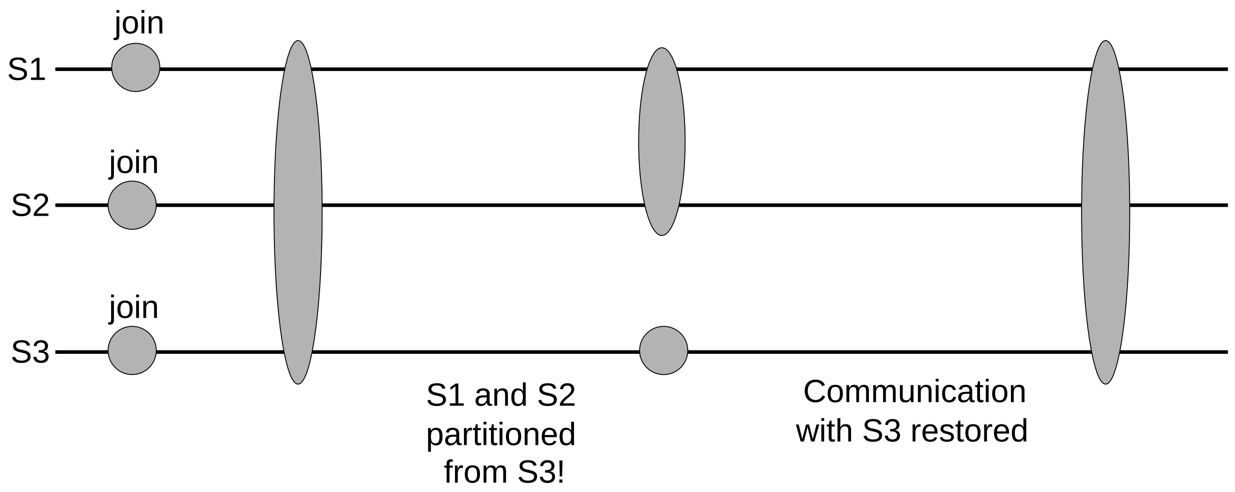
join
S1
join
S2
join
S3
S1 and S2 partitioned from S3!
Communication with S3 restored

Figure 4: Overview of how the membership service works with respect to partitioning failures.

3.2 The Group Method Invocation Service

Jgroup differs from existing object group systems due to its uniform communication interface based entirely on group method invocations. Clients and servers alike interact with groups by remotely invoking methods on them. In this manner, benefits of object-orientation such as abstraction, encapsulation and inheritance are extended to internal communication among servers.

Although they share the same intercommunication paradigm, we distinguish between *internal group method invocations* (IGMI) performed by servers and *external group method invocations* (EGMI) performed by clients. There are several reasons for this distinction:

- *Visibility:* Methods to be used for implementing a replicated service should not be visible to clients. Clients should be able to access only the "public" interface defining the service, while methods invoked by servers should be considered "private" to the implementation.

- *Transparency:* Jgroup strives to provide an invocation mechanism for clients that is completely transparent with respect to standard RMI. This means that clients are not required to be aware that they are invoking a method on a group of servers rather than a single one. Servers, on the other hand, that implement the replicated service may have different requirements for group invocations, such as obtaining a result from each server in the current view.

- *Efficiency:* Having identical specifications for external and internal group method invocations would have required that clients become members of the group, resulting in poor scalability of the system. In Jgroup, external group method invocations have semantics that are slightly weaker than those for internal group method invocations. Recognition of this difference results in a much more scalable system by limiting the higher costs of full group membership to servers, which are typically far fewer in number than clients.

When developing dependable distributed services, internal methods are collected to form the *internal remote interface* of the server object, while external methods are collected to form its *external remote interface.* A proxy object capable of handling group method invocation, will be generated dynamically (at runtime) based on the remote interfaces of the server object. This proxy enables a client (or server) object to communicate with the entire group of server objects, as if it was a local or remote method invocation.

In order to perform an internal group method invocation, servers must obtain an appropriate group proxy from the Jgroup runtime running in the local Java virtual machine. Clients that need to interact with a group, on the other hand, must request a stub from a *registry service*, whose task is to enable servers to register themselves under a group name. Clients can then look up desired services by name in the registry and obtain their stub. Currently, Jgroup support two registry services. The first one is called *dependable registry* [15], and it derives from the standard registry included in Java RMI, while the

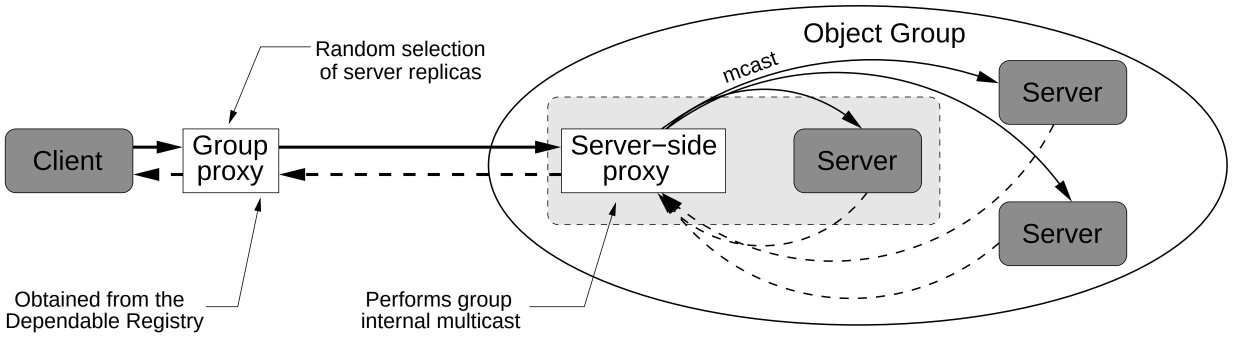
Random selection of server replicas
Object Group
mcast
Server
Client
Group proxy
Server−side proxy
Server
Server
Obtained from the Dependable Registry
Performs group internal multicast

Figure 5: Illustration of how the external group method invocation with multicast semantics is performed using the group proxy obtained from the dependable registry and its server-side counterpart. It is the server group members that bind with the dependable registry whom then can generate the group proxy based on its known members.

second is based on the Jini lookup service. The dependable registry service is an integral part of Jgroup and is implemented as a replicated service using Jgroup itself. The choice of which registry service to use is left to developers; the dependable registry service is simpler than the Jini based lookup service, as it requires complex deployment mechanisms of Jini.

In the following sections, we discuss how internal and external group method invocations work in Jgroup, and how internal invocations substitute message multicasting as the basic communication paradigm. In particular, we describe the reliability guarantees that group method invocations provide. They are derived from similar properties that have been defined for message deliveries in message-based group communication systems [7]. We say that an object (client or server) *performs* a method invocation at the time it invokes a method on a group; we say that a server *completes* an invocation when it terminates executing the associated method. Method invocations are uniquely identified such that it is possible to establish a one-to-one correspondence between performing and completing them.

3.2.1 Internal Group Method Invocations

Unlike traditional Java remote method invocations, internal group method invocations (IGMI) return an array of results rather than a single value. IGMI comes in two different flavors: *synchronous* and *asynchronous*. In synchronous IGMI, the invoker remains blocked until an array containing results from each server that completed the invocation can be assembled and returned to it (from which servers result values are contained in the return array is discussed below). There are many programming scenarios where such blocking may be too costly, as it can unblock only when the last server to complete the invocation has produced its result. Furthermore, it requires programmers to consider issues such as deadlock that may be caused by circular invocations. In asynchronous IGMI, the invoker does not block but specifies a *callback* object that will be notified when return values are ready from servers completing the invocation.

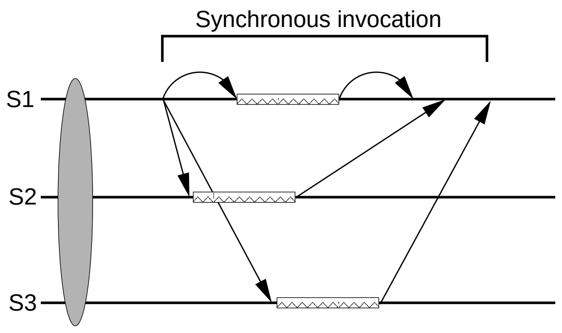

Figure 6: Synchronous IGMI method invocation.

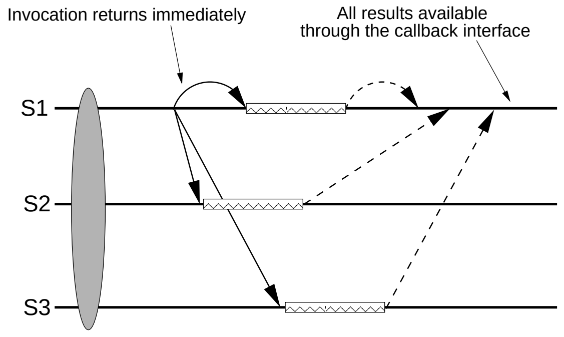
Invocation returns immediately
All results available through the callback interface
S1
S2
S3

Figure 7: Asynchronous IGMI method invocation with callback.

If the return type of the method being invoked is `void`, no return value is provided by the invocation. The invoker has two possibilities: it can specify a callback object to receive notifications about the completion of the invocation, or it can specify `null`, meaning that it is not interested in knowing when the method completes.

Completion of IGMI by the servers forming a group satisfies a variant of "view synchrony" that has proven to be an important property for reasoning about reliability in message-based systems [8]. Informally, view synchrony requires two servers that install the same pair of consecutive views to complete the same set of IGMI during the first view of the pair. In other words, before a new view can be installed, all servers belonging to both the current and the new view have to agree on the set of IGMI they have completed in the current view. This enables a server to reason about the state of other servers in the group using only local information such the history of installed views and the set of completed IGMI. Clearly, application semantics may require that servers need to agree on not only the set of completed IGMI but also the order in which they were completed. In Jgroup, different ordering semantics for IGMI completions may be implemented through additional layers on top of the basic group method invocation service.

We now outline some of the main properties that IGMI satisfy. First, they are *live*: an IGMI is guaranteed to terminate either with a reply array (containing at least the return value computed by the invoker itself), or with one of the application-defined exception contained in the *throws* clause of the method. Furthermore, if an operational server S completes some IGMI in a view, all servers included in that view will also complete the same invocation, or S will install a new view. Since installed views represent the current failure scenario as perceived by servers, this property guarantees that an IGMI will be completed by every other server that is in the same partition as the invoker. IGMI also satisfy "integrity" requirements whereby each IGMI is completed by each server at most once, and only if some server has previously performed it. Finally, Jgroup guarantees that each IGMI be completed in at most one view. In other words, if different servers complete the same IGMI, they cannot complete it in different views. In this manner, all result values that are contained in the reply array are guaranteed to have been computed during the

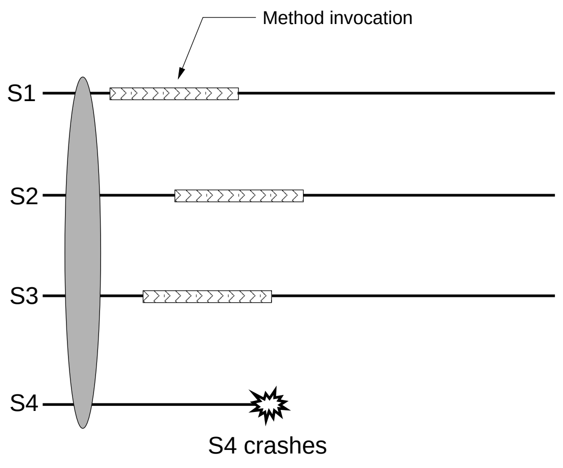

Figure 8: Invalid view synchrony execution.

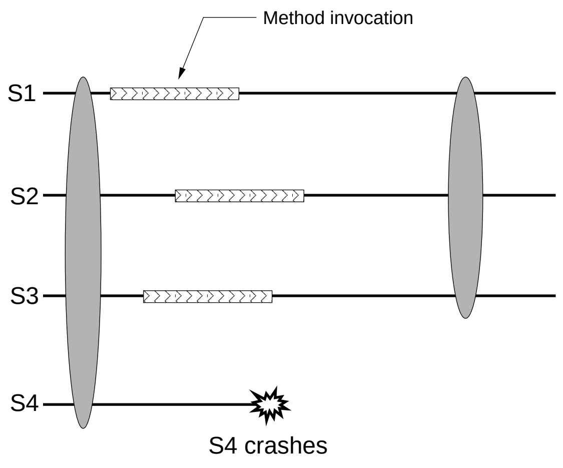

Figure 9: A valid view synchrony execution.

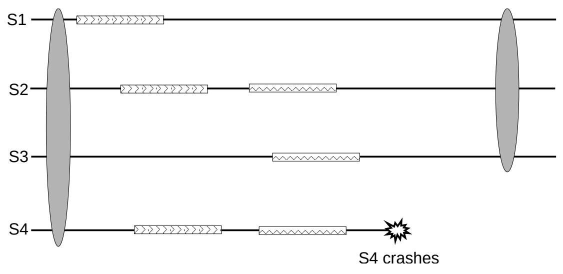

Figure 10: Invalid view synchrony execution.

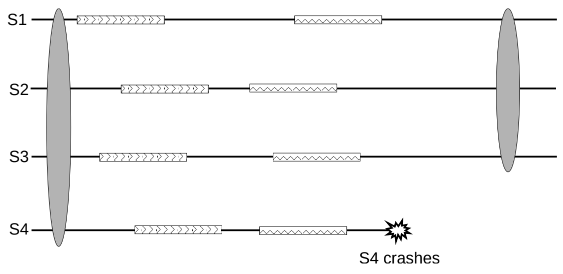

Figure 11: A valid view synchrony execution.

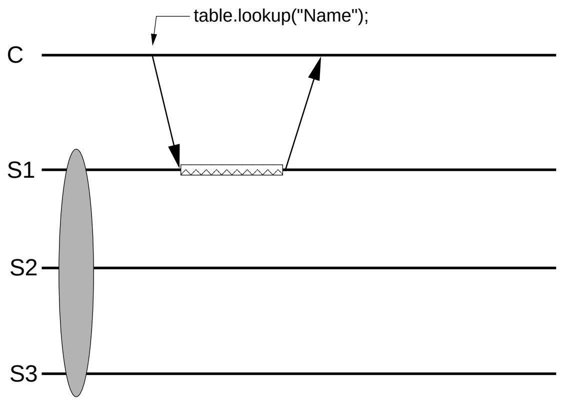

Figure 12: EGMI anycast method invocation (read operation).

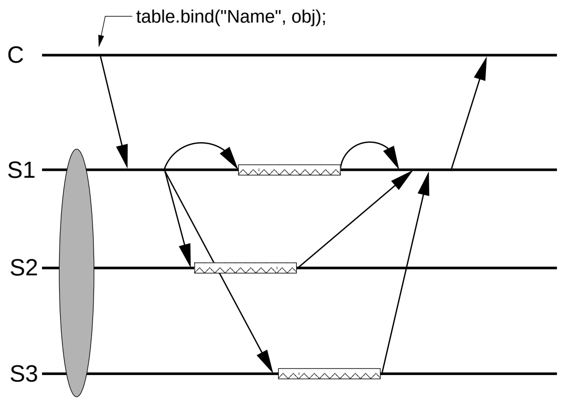

Figure 13: EGMI multicast method invocation (write operation).

same view.

3.2.2 External Group Method Invocations

External group method invocations (EGMI) that characterize client-to-server interactions are completely transparent to clients that use them as if they were standard remote method invocations. When designing the external remote interface for a service, an application developer must choose between the *anycast* and the *multicast* invocation semantics. An anycast EGMI performed by a client on a group will be completed by at least one server of the group, unless there are no operational servers in the client's partition. Anycast invocations are suitable for implementing methods that do not modify the replicated server state, as in query requests to interrogate a database. A multicast EGMI performed by a client on a group will be completed by every server of the group that is in the same partition as the client. Multicast invocations are suitable for implementing methods that may update the replicated server state.

The choice of which invocation semantics to associate with each method rests with the programmer of the distributed service when designing its external remote interface. The default semantics for an external method is anycast. Inclusion of the tag McastRemoteException in the *throws* clause of a method signals that it needs to be invoked with multicast semantics. When generating the stub for an external interface, the group manager will analyze the *throws* clause using reflection (*cf.* `java.lang.reflect`) and produces the appropriate code for the stub object.

Our implementation of Jgroup guarantees that EGMI are live: if at least one server remains operational and in the same partition as the invoking client, EGMI will eventually complete with a reply value being returned to the client. Furthermore, an EGMI is completed by each server at most once, and only if some client has previously performed it. These properties hold for both anycast and multicast versions of EGMI. In the case of multicast EGMI, Jgroup also guarantees view synchrony as defined in the previous section.

Internal and external group method invocations differ in an important aspect. Whereas an IGMI, if it completes, is guaranteed to complete in the same view at all servers, an EGMI may complete in several different concurrent views. This is possible, for example, when a server completes the EGMI but becomes partitioned from the client before delivering the result. Failing to receive a response for the EGMI, the client's stub has to contact other servers that may be available, and this may cause the same EGMI to be completed by different servers in several concurrent views. The only solution to this problem would be to have the client join the group before issuing the EGMI. In this manner, the client would participate in the view agreement protocol and could delay the installation of a new view in order to guarantee the completion of a method in a particular view. Clearly, such a solution may become too costly as group sizes would no longer be determined by the number of server objects (degree of replication of the service), but by the number of clients, which could be very large.

The fact that EGMI may complete in several different concurrent views has important consequences for dependable application development. Consider an EGMI that is indeed completed by two different servers in two concurrent views due to a partition as described above. Assume that the EGMI is a request to update part of the replicated server state. Now, when the partition is repaired and the two concurrent views merge to a common view, we are faced with the problem of reconciling server states that have evolved independently in the two partitions. The problem is discussed in length below but what is clear is that a simple-minded merging of the two states will result in the same update (issued as a single EGMI) being applied twice. To address the problem, Jgroup assigns each EGMI a unique identifier. In this manner, the reconciliation protocol can detect that the two updates that are being reported by the two merging partitions are really the same and should not both be applied.

One of the goals of Jgroup has been the complete transparency of server replication to clients. This requires that from a clients perspective, EGMI should be indistinguishable from standard Java RMI. This has ruled out consideration of alternative definitions for EGMI including multi-value results or asynchronous invocations.

3.3 The State Merging Service

While partition-awareness is necessary for rendering services more available in partitionable systems, it can also be a source of significant complexity for application development. This is simply a consequence of the intrinsic availability-consistency tradeoff for distributed applications and is independent of any of the design choices we have made for Jgroup.

Being based on a partitionable GMS, Jgroup admits partition-aware applications that have to cope with multiple concurrent views. Application semantics dictates which of its services remain available where during partitionings. When failures are repaired and multiple partitions merge, a new shared state has to be constructed. This new state should reconcile, to the extent possible, any divergence that may have taken place during partitioned operation.

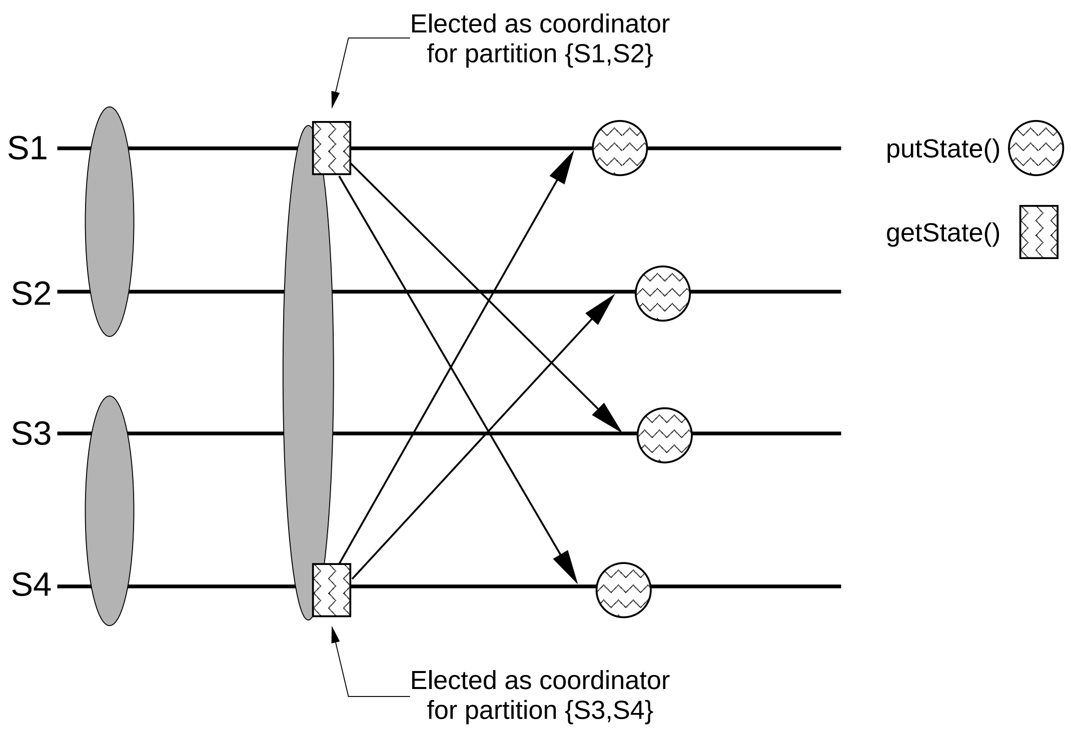
Elected as coordinator for partition {S1,S2}
S1
S2
S3
S4
putState()
getState()
Elected as coordinator for partition {S3,S4}

Figure 14: A state merge scenario without failure.

Generically, state reconciliation tries to construct a new state that reflects the effects of all non-conflicting concurrent updates and detect if there have been any conflicting concurrent updates to the state. While it is impossible to automate completely state reconciliation for arbitrary applications, a lot can be accomplished at the system level for simplifying the task [5]. Jgroup includes a state merging service (SMS) that provides support for building application-specific reconciliation protocols based on stylized interactions. The basic paradigm is that of full information exchange — when multiple partitions merge into a new one, a coordinator is elected among the servers in each of the merging partitions; each coordinator acts on behalf of its partition and diffuses state information necessary to update those servers that were not in its partition. When a server receives such information from a coordinator, it applies it to its local copy of the state. This one-round distribution scheme has proven to be extremely useful when developing partition-aware applications [6, 15].

SMS drives the state reconciliation protocol by calling back to servers for "getting" and "merging" information about their state. It also handles coordinator election and information diffusion. To be able to use SMS for building reconciliation protocols, servers of partition-aware applications 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 replicated state and be able to provide state information when requested by SMS;

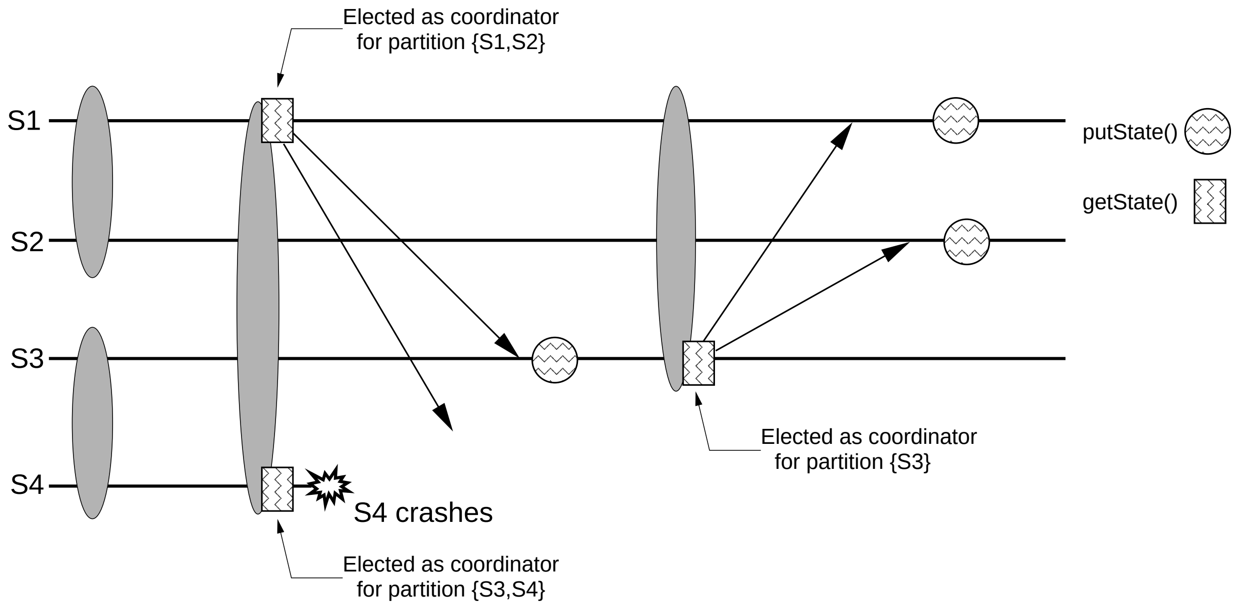
Elected as coordinator for partition {S1,S2}
S1
S2
S3
S4
putState()
getState()
S4 crashes
Elected as coordinator for partition {S3}
Elected as coordinator for partition {S3,S4}

Figure 15: State merge scenario with coordinator failure.

- a server must be able to apply any incoming updates to its local state.

These assumptions restrict the applicability of SMS. For example, applications with high-consistency requirements may not be able to apply conflicting updates to the same record. Note, however, that this is intrinsic to partition-awareness, and is not a limitation of SMS.

In order to elect a coordinator, SMS requires information about "who can act on behalf of whom". At a given time, we say that server s_1 is *up-to-date* with respect to server s_2 if all information known by s_2 is known also by s_1. A server s_1 may act as a coordinator on behalf of a server s_2 if s_1 is up-to-date with respect to s_2. 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 can become up-to-date with respect to these servers. On the other hand, a server ceases to be up-to-date with respect to other servers upon the installation of a new view excluding them. Consider for example a server s_1 installing a view v that excludes server s_2. Since the state of s_2 may be evolving concurrently (and inconsistently) with respect to s_1, SMS declares s_1 as being not up-to-date with respect to s_2. The main requirement satisfied by SMS is *liveness*: 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 through different servers that may be elected coordinators and provide information on behalf of one of the two servers). Another important property is *agreement*: servers that install the same pair of views in the same order are guaranteed to receive the same state information through invocations of their "merging" methods 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, SMS satisfies an integrity property such that SMS will not initiate a state reconciliation protocol without reasons (e.g., if all servers are already up-to-date).

4 Building and Running Jgroup Applications

While developing Jgroup, we make use of the Jakarta project's build tool called *Ant* [2]. This allows us to both compile and run our software with very little hassel. In this section we will try to explain how to use Ant to build and run Jgroup applications.

To check if your local machine has Ant installed, you may type the following command:

```
ant -help
```

If the above command reports something like, *command not found*, you will need to download[2] and install the ant tool on your local machine before continuing. The Ant program can be compared with the UNIX `make` tool, but it is specially designed for building Java programs and you specify all rules and targets using XML. To use Ant for building a project, the root directory of the project should contain the `build.xml` file, describing the various targets needed to compile a project, entirely or in part.

4.1 Building Jgroup

The most important build (compile) targets in the Jgroup `build.xml` file is summarized below:

- build target will check which files needs to be built and builds only those. build is the default target, and will be used by ant, if no target is specified.
- all is used to rebuild all files, even those that do not need to be rebuilt.
- clean has the obvious meaning to remove all files relevant to the build process.
- doc is used to create javadoc documentation for Jgroup.
- cleandoc removes all files related to javadoc documentation.

So in order to rebuild Jgroup from scratch, the following command must be executed from the Jgroup root directory:

```
ant all
```

You may also execute this from other directories, but then you must specify which `build.xml` file to use:

```
ant -buildfile ~/Jgroup2/build.xml all
```

[2]Ant can be downloaded from `http://ant.jakarta.apache.org/`

4.2 Running Jgroup Applications

To simplify running the various Jgroup applications provided, we have specified targets for the programs included with Jgroup, some of which are summarized below:

- dregistry creates an instance of the dependable registry provided with Jgroup on the local machine. More than one instance of the registry may be started within the distributed system, thus increasing the fault tolerance of the registry service.
- helloserver start an instance of the hello server class on the local machine. More than one instance of the hello server may be executing in the distributed system, and they will all join the same object group.
- helloclient executes a hello client on the local machine. This will result in the client contacting one of the hello server instances that are running in the distributed system.

Note that most of the programs provided with Jgroup requires an XML configuration file describing the distributed system in which the Jgroup application should execute. The location of this file is typically in the `Jgroup2/config` directory, but you may change this location for all applications using the `system.configuration.url` property in the `build.xml` file, or you may edit each target separately.

An example run with the helloserver

To demonstrate a very simple application that makes use of Jgroup, first ensure that the `Jgroup2/config/config.xml` file contains the correct domain and a set of hosts on which you wish to run this application. Only one domain is required, but at least three hosts are needed to run this example.

Initially, the dependable registry should be started on one of the hosts specified in the distributed system configuration:

```
ant dregistry
```

Note that additional replicas (instances) of the registry may be started on other machines in the distributed system, but this is not required for the example. Next, two or more hello server repliacs must be started on distinct hosts that is specified in the distributed system.

```
ant helloserver
```

Assuming that both the dependable registry and all hello server replicas are running, we are now ready to start any number of hello clients, to invoke the hello server replicas. Note that the host on which to run the hello clients does not need to be specified in the distributed system configuration.

```
ant helloclient
```

4.3 Adding a new Jgroup Application

If you wish to develop a new application based on Jgroup, the following will explain how you can add a new run target to the `build.xml` file. We do not explain how to compile your application, but there are numerous approaches you can take; the simplest being to add your application under the Jgroup directory tree, and it will be compiled together with Jgroup itself. Another approach is to add a separate compile target for your application. For details see the build target in the `build.xml` file, or refer to the Ant documentation [3].

The simplest approach for adding a new run target is to cut and paste an existing target; thus it is convenient to look at the target for the hello server to determine which parts needs to change for your application:

```
<target name="helloserver" depends="build">
  <java classname="jgroup.test.hello.HelloServer"
        fork="true" classpathref="run.path">
    <sysproperty key="java.compiler" value="${java.compiler}"/>
    <sysproperty key="jgroup.debug.dir" value="${debug.dir}"/>
    <sysproperty key="jgroup.debug.fileLevel" value="${debug.fileLevel}"/>
    <sysproperty key="jgroup.debug.outLevel" value="${debug.outLevel}"/>
    <arg value="${system.configuration.url}"/>
  </java>
</target>
```

The target name is a unique name within a build file and serves to identify the target. Also notice that the helloserver target depends on the build target, which basically means that Ant will check if it needs to compile something before trying to run the helloserver. Within the helloserver target, we use the java tag, specifying the class name for the hello server. For details about the java tag, please refer to the Ant documentation [3]. The sysproperty is mainly used to pass various attributes to the jgroup debugger, while the arg is used to pass command line arguments to the hello server. The parts that needs to be modified for a new application will typically involve the target name and classname, and possibly add more arg value lines. Most other parts will remain identical for all run targets.

The classpathref attribute of the java tag is used to specify the class path for the hello server application. This is done in a separate part of the `build.xml` file, and this is used by most other run targets as well:

```
<path id="run.path">
  <pathelement location="classes"/>
  <pathelement location="${lib}/crimson.jar"/>
</path>
```

This is simply a listing of all the class path elements required to run the hello server application. If your application requires another library (jar file) or if you keep your application in another directory than the classes directory in which the Jgroup classes resides, you will need to add your own directory or jar file to the run.path.

5 Application Examples

In this section we present two Jgroup example applications.

```
package jgroup.test.hello;

public interface Hello
  extends jgroup.core.ExternalGMIListener
{
  Answer sayHello()
    throws java.rmi.RemoteException;
}
```

Figure 16: The external Hello interface.

```
package jgroup.test.hello;

public interface InternalHello
  extends jgroup.core.InternalGMIListener
{
  public Object time()
    throws java.rmi.RemoteException;
}
```

Figure 17: The InternalHello interface.

5.1 The Hello Example

The hello application is a very simple example meant to get you started with the most basic Jgroup services. The application is composed of a set of servers that accepts client requests and responds with a hello message, the time of the reply and the name of the server that responded. To enable clients to communicate with a group of hello servers, the servers must bind their local reference to a group proxy that is stored in the registry service.

5.1.1 The Hello Server

Designing a replicated Jgroup service requires that all methods of the service be added to an external interface, allowing clients to invoke these methods on the group of servers. External GMI interfaces must extend the ExternalGMIListener, and each of its methods must specify the exception RemoteException in its *throws* clause. The Hello interface shown in Figure 16, contains only method sayHello and returns an Answer object (see Figure 18) to clients invoking that method.

Notice that the Answer class implements the Serializable interface. This is the standard approach for passing an object "by value" over the network. The process of serializing something is merly a way to convert an object instance (in memory form) into a stream of bytes that can be passed over the network. By implementing the Serializable interface, the Answer class actually inherits a few methods from the Object class which provides the default serialization. The default serialization provided by the Object class, will serialize all the fields of the Answer class so that they can be transfered over the network, and be reinstated in an empty Answer object once it has been transfered.

Figure 19 shows the code for the hello server. The server class has been divided into sections to simplify its understanding. As can be seen from the *implements* clause, the server implements the Hello interface (Figure 16) and the InternalHello interface (Figure 17). These are used for external and internal group method invocations, respectivly. In addition, the server also implements the MembershipListener interface, enabling the server to be notified of changes in the membership of the group.

The only tasks of the main method of the HelloServer is to parse the system configuration file, and to create an instance of the HelloServer on the local machine. The constructor will prepare the answer message, and will obtain a reference to the local services (layers) from the group manager. In particular the membership and external GMI services are used by

```
package jgroup.test.hello;

public class Answer
  implements java.io.Serializable
{
  private String message;

  /** Time of the last view that has been installed by the group members */
  private Object[] time;

  public Answer(String message)
  {
    this.message = message;
  }

  public void setTime(Object[] timeValues)
  {
    time = timeValues;
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    buf.append(message);
    if (time != null) {
      for (int i = 0; i < time.length; i++) {
        buf.append(time[i]);
        buf.append(" ");
      }
    }
    return buf.toString();
  }
}
```

Figure 18: The Answer class.

```
package jgroup.test.hello;

import java.rmi.*;

import jgroup.util.*;
import jgroup.core.*;

public class HelloServer
  implements Hello, InternalHello, MembershipListener
{
  /** Stores the answer for replying to external invocations */
  private Answer answer;

  /** The internal group proxy for the HelloServer */
  private InternalHello internalHello;

  ////////////////////////////////////////////////////////////////////////////
  // Main method (initialize the HelloServer object)
  ////////////////////////////////////////////////////////////////////////////

  public static void main(String argv[])
    throws Exception
  {
    String configURL = null;

    for (int i = 0 ; i < argv.length ; i++) {
      if (argv[i].startsWith("-")) {
        System.err.println("Unknown option " + argv[i]);
        Abort.usage("helloserver <configURL>");
      } else {
        if (configURL != null) {
          System.err.println("Hostfile already specified");
          Abort.usage("helloserver <configURL>");
        }
        configURL = argv[i];
      }
    }
    if (configURL == null) {
      System.err.println("You must specify an configURL");
      Abort.usage("helloserver <configURL>");
    }
    HelloServer obj = new HelloServer(configURL);
  }

  ....
```

Figure 19: The HelloServer class.

```
....

////////////////////////////////////////////////////////////////////////////
// Constructor for the HelloServer
////////////////////////////////////////////////////////////////////////////

public HelloServer(String configURL)
  throws Exception
{
  /* Prepare the answer for replying to external invocations. */
  answer = new Answer("Hello from " + Network.getLocalHostName());

  /* Obtain the group manager for this server object */
  GroupManager gm = GroupManager.getGroupManager(configURL, this);

  /* Obtain proxies for the services required by the HelloServer */
  MembershipService pgms =
    (MembershipService) gm.getService(MembershipService.class);
  ExternalGMIService egmis =
    (ExternalGMIService) gm.getService(ExternalGMIService.class);
  internalHello = (InternalHello) gm.getService(InternalHello.class);

  /* Join the group and bind the server in the dependable registry */
  pgms.join(12);
  egmis.bind("Jgroup/HelloServer");
  System.out.println("Server ready and bound to the reliable registry");
}

////////////////////////////////////////////////////////////////////////////
// Methods from the Hello interface (External Group Method Invocation)
////////////////////////////////////////////////////////////////////////////

public Answer sayHello()
  throws RemoteException
{
  return answer;
}

////////////////////////////////////////////////////////////////////////////
// Methods from the InternalHello interface (Internal Group Method Invocation)
////////////////////////////////////////////////////////////////////////////

public Object time()
  throws RemoteException
{
  return new Long(System.currentTimeMillis());
}

....
```

Figure 20: The HelloServer class.

```
  ....

  ////////////////////////////////////////////////////////////////////////////
  // Methods from MembershipListener
  ////////////////////////////////////////////////////////////////////////////

  public void viewChange(View view)
  {
    System.out.println("View id: " + view.getVid());
    MemberId[] members = view.getMembers();
    for (int i = 0; i < members.length; i++)
      System.out.println("Member[" + i + "]: " + members[i]);

    try {
      /*
       * The time() method is defined in the InternalHello interface and
       * is marked as a group internal method.  By definition, all group
       * internal methods will return an array of values instead of a
       * single value as with standard remote (or external group) method
       * calls.
       */
      Object[] objs = (Object[]) internalHello.time();
      for (int i = 0; i < objs.length; i++)
        System.out.println("Time: " + objs[i]);
      answer.setTime(objs);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void prepareChange() { }

  public void hasLeft() { }
}
```

Figure 21: The HelloServer class.

```
package jgroup.test.hello;

import jgroup.util.*;
import jgroup.core.registry.*;

public class HelloClient
{

  public static void main(String argv[])
    throws Exception
  {
    String configURL = null;

    for (int i = 0 ; i < argv.length ; i++) {
      if (argv[i].startsWith("-")) {
        System.err.println("Unknown option " + argv[i]);
        Abort.usage("helloclient <configURL>");
      } else {
        if (configURL != null) {
          System.err.println("configURL already specified");
          Abort.usage("helloclient <configURL>");
        }
        configURL = argv[i];
      }
    }
    if (configURL == null) {
      System.err.println("You must specify an configURL");
      Abort.usage("helloclient <configURL>");
    }

    /*
     * Obtain a proxy for the depedable registry running in the
     * distributed system described in the configURL on the specified port;
     */
    DependableRegistry registry = RegistryFactory.getRegistry(configURL);

    /*
     * Retrieve a proxy for an object group implementing the Hello
     * interface registered under the name "Jgroup/HelloServer".
     */
    Hello server = (Hello) registry.lookup("Jgroup/HelloServer");

    /*
     * Invoking method sayHello, which returns an object of type Answer
     * containing a string produced by the contacted object plus
     * the time at which the groups members installed the last view.
     */
    Answer answer = server.sayHello();
    System.out.println(answer);
  }
}
```

Figure 22: The HelloClient class.

the server join its object group and bind its reference with the dependable registry, respectivly. Notice that the binding of the server reference is performed through the external GMI layer instead of directly through a registry instance. This is basically a convenient approach, since we need to access all the dependable registry replicas. In addition to the above, the constructor also obtains a proxy for the HelloServer group, enabling them communicate with each other through group internal method invocations.

The implementation of the external method sayHello of the Hello interface is simply to return the already prepared answer, while implementing the internal method time of the InternalHello interface is to return the local time of this HelloServer replica. So when another HelloServer replica invokes the time method, it will obtain one reply from each replica. This last part is exactly what is done in the implementation of the MembershipListener method viewChange. For each view change of the hello server group, the internal method time is invoked and an array of results are returned. These are incorporated with the local Answer object through the setTime method. Thus, the latest time values (the time when the last view was installed) can also be seen by clients invoking the sayHello method on the external Hello interface.

The HelloClient implementation is very simple (Figure 22). Similarly to the server, also the client needs to parse the command line arguments to obtain the system configuration file, which is needed in order to determine the location of the dependable registry. Once a reference to the dependable registry has been obtain, the client may perform a lookup in order to obtain a reference to the server based on the pre-assigned name ("Jgroup/HelloServer"). After obtaining this reference, the client may invoke the method sayHello at will. The HelloClient will simply print the Answer object obtained from the sayHello invocation and exit.

5.1.2 The workings of internal group method invocation

Below we detail the inner workings of the internal group method invocation (IGMI) layer of Jgroup. Figure 23 sketch how Jgroup handles an invocation on an internal proxy object. Once the HelloServer requests a group manager object using the static method GroupManager.getGroupManager(), the group manager will examine the interfaces implemented by the HelloServer and determine that it in fact implements an internal interface. This is since the InternalHello interface extends the InternalGMIListener interface. Given this fact, the group manager will construct an IGMI proxy dynamically (at runtime) that implements the InternalHello interface. The exact details of how this proxy is generated is outside the scope of this tutorial; Jgroup simply use the proxy mechanism found in java.lang.reflect.Proxy. This proxy is shown with shaded background in Figure 23. Next, the HelloServer can obtain a reference to this dynamically generated proxy by invoking the getService(InternalHello.class) method on the group manager.

Now that the server has an IGMI reference, it can perform invocations on all servers using a single method invocation like this one:

```
Object[] objs = (Object[]) internalHello.time();
```

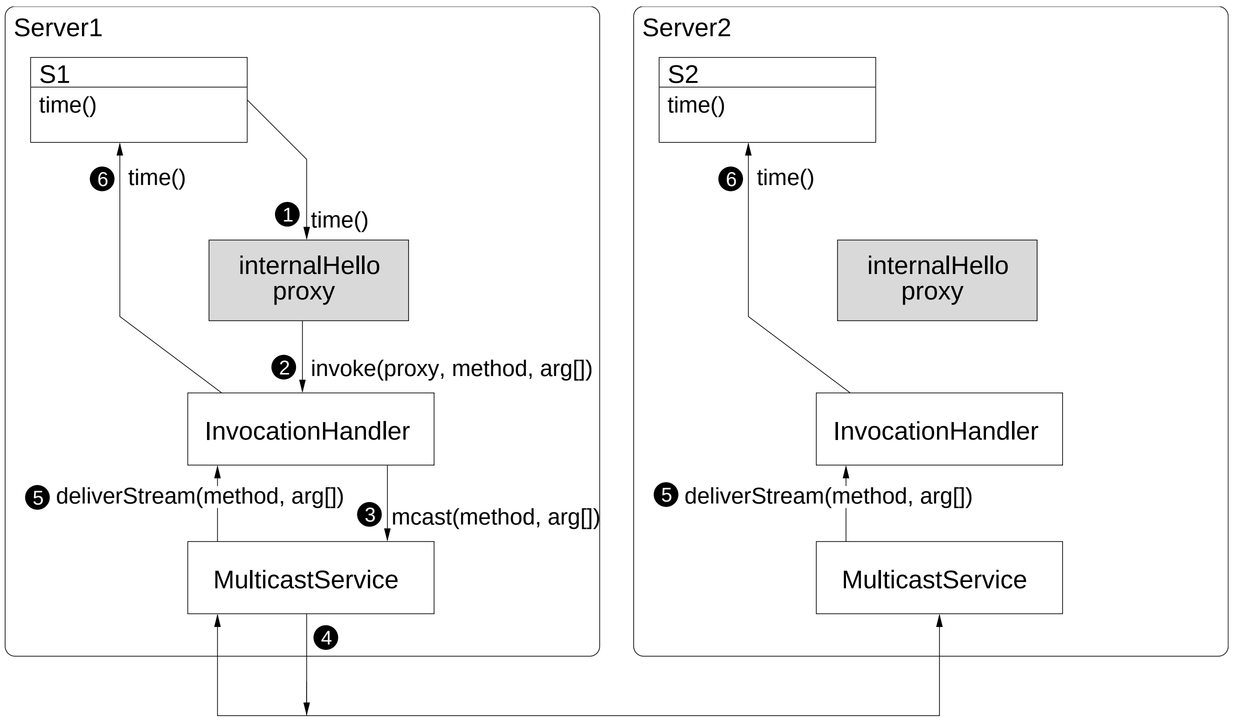

Figure 23: Details of the workings of the InternalHello proxy.

Such an invocation will cause a chain of events in the Jgroup layers, as shown in Figure 23. The actual invocation occur at ❶. The dynamically generated proxy simply converts the time() invocation into an invoke() method invocation on an InvocationHandler implementation (❷). The InvocationHandler used for IGMI is called IntGroupHandler and is specialized for handling internal method invocation within an object group. It will perform a multicast (❸,❹) to all IntGroupHandler layers that is member of this group (❺). Once a member's IntGroupHandler has received such an invocation, it will perform the actual invocation on the server implementation (❻), and return the results back following the reverse path.

5.2 A Dependable Computation Service

In the following, we present a simple application example exploiting most of the characteristics of Jgroup. The service being performed is a dependable computation service that executes arbitrary tasks requested by clients. The service is composed of a group of servers that accept request from clients and coordinate the execution of tasks in order to guarantee that each task is completed and no task is executed twice (whenever possible). When a task is completed, one of the servers calls back the client that requested the execution and notifies the result to it.

Section 5.2.1 and Section 5.2.2 will show how to write the server and the client ocde of

```
import java.rmi.*;
import jgroup.*;

public interface ComputeService extends ExternalGMIListener {

  public void compute(Task t, ResultListener rl)
    throws RemoteException, McastRemoteException;

}

public interface Task {

  public void init(String[] args);

  public Object run();

}

public interface ResultListener extends Remote {

  public void result(Object result)
    throws RemoteException;

}
```

Figure 24: The ComputeService interface and its related interfaces.

the computation service application. The purpose of this example is to show how to use the Jgroup API to build a simple replicated application, and not how to build a production version of such application. In other word, the rest of this section is concentrated on the details of the Jgroup API, and not on the details of the application. A more complete version of this example will eventually be available for download and testing.

5.2.1 The Computation Server

The first step to accomplish when designing a replicated Jgroup service is to create the external interface containing the methods that can be invoked by clients. In our example, this interface is called ComputeService and is shown in Figure 24. In order for an interface to be an external GMI interface, it must extend ExternalGMIListener; furthermore, each of its methods must contain exception RemoteException in its *throws* clause. Interface ComputeService contains only method compute, used by clients to request the execution of a task. By adding exception McastRemoteException to the *throws* clause of method compute, we force its execution following the multicast semantics. In this way, servers in the same partition will receive the same set of compute invocations, and will be able to maintain the same collection of tasks to be performed. Having the same knowledge about tasks, servers may exploit the state merging service in order to reconstruct a consistent state after the end of a partitioning; furthermore, the workload can be easily subdivided among servers in each of the partitions.

Method compute has two arguments; the first is a task object containing the code to be computed, the second is an object that will listen for the result of the task. These

```
import java.rmi.*;
import jgroup.*;

public interface InternalComputeService
extends InternalGMIListener {

  public void completed(IID id, Object result);

}
```

Figure 25: The InternalComputeService class.

objects must implement interfaces Task and ResultListener, respectively (see Figure 24). Task extends interface Serializable, meaning that task objects are passed "by value" to the computation service. ResultListener, on the other hand, extends interface Remote, meaning that result listeners are passed "by reference" to the computation service. After having created a task, a client invokes method init on it in order to initialize it with an array of strings. This design enables us to write a generic client capable to request the execution of arbitrary tasks initialized with command-line arguments. Method run of Task is invoked by the server after having received the task through a compute invocation. Method run performs some computation and returns an object containing the result. This result is delivered to the client by performing a remote method invocation of result on the result listener provided when invoking compute.

After having defined the external interface, the next step is to define the internal interface containing the methods invoked by servers to communicate among themselves. Figure 25 shows interface InternalComputeService used in our example. This interface contains only method completed, used to communicate the result of a task to the other servers in a group. A result is maintained by a server until it is delivered to the corresponding result listener. Tasks are identified using invocation identifiers IID provided by Jgroup.

At this point, we are ready to write the server implementation class, which is illustrated in Figure 26. Our implementation takes advantage of each of the facilities provided by Jgroup, i.e. the group membership service, the state merging service, and the internal and external group method invocation service. In order to receive event notifications originated by these services, a set of interfaces needs to be implemented by the server class. Interface MembershipListener is used to receive view changes, while interface MergingListener contains the callbacks used by SMS. Interfaces ComputeService and InternalComputeService are used to receive group method invocations performed by clients and servers, respectively.

The Constructor

The constructor for a computation server is very simple. First of all, a new Workload object is initialized. Tasks to be computed will be stored in this data structure. In order to access the services provided by Jgroup, a server must first notify Jgroup of its existance, and it must provide one or more objects capable to listen to Jgroup event notifications. This is performed by invoking static method getGroupManager on class GroupManager. This method takes two parameters: the former is an array of configuration objects, while the latter is an array of listener objects.

For the purpose of this example, the configuration array contains only a DistributedSystem object. A distributed system describes the set of hosts in which the servers will be run. Several constructors for distributed systems exists (see Section 6.4.1); here, we use a constructor taking a simple string argument containing the name of an *hostfile*. An hostfile is a text file containing a list of host names separated by a carriage return.

The listener array contains only a reference to the server itself using the keyword this. In this way, each event notification related to the group membership service, the state merging service and the group method invocation service (i.e., the interfaces implemented by ComputeServer) are intercepted by the server.

The next step is to obtain explicit references to the services requested by invoking method getService on the group manager. These references will be used in order to join and leave the group, to obtain information about external group method invocations and to perform internal group method invocations. In the example, the method is invoked four times to obtain references to GMS, the external and internal GMI services and a proxy for the internal interface defined earlier. The class object of the requested service interface is passed as an argument to method getService. Note that the no reference for SMS is requested. The reason for this is that servers cannot voluntarily interact with SMS, but only listen to callback invocations originated by it.

After having obtained these references, a group must be explicitly joined. This operation is subdivided in two parts. First, we must inform the other servers in the group to be joined. This operation is performed by invoking method join on the GMS reference. Second, we must inform possible clients that this server is part of a group and is available to accept group method invocations. This is performed by invoking method bind on the external GMI service reference, which takes care of registering the server under a group name on the dependable registry service running in the distributed system specified when invoking method getGroupManager.

Methods of the External and Internal Interfaces

As explained above, ComputeServer must implement method compute contained in the external interface of the service. The code associated to this method looks as follows:

```
public void compute(Task t, ResultListener rl) throws RemoteException {
  IID id = externalService.getIdentifier();
  workload.insert(t, id, rl);
}
```

The first action of compute is to obtain an identifier for the invocation by invoking method getIdentifier on the external GMI service. This identifier is used among servers composing the group to uniquely identify a task. Then, the task, the invocation identifier and the result listener are inserted in the workload object initialized at the beginning.

When a server completes the execution of a task (see Section 5.2.1), it informs other servers in the group by performing an internal invocations of method completed and speci-

```
import java.rmi.*;
import jgroup.*;

public class ComputeServer
implements MembershipListener, MergingListener, ComputeService,
           InternalComputeService {

  MembershipService membershipService;
  ExternalGMIService externalService;
  InternalGMIService internalService;
  InternalComputeService groupProxy;

  // Constructor

  public ComputeServer()
    throws Exception
  {
    Workload workload = new Workload();

    Object[] listeners = new Object[] { this };
    Object[] config = new Object[] { new DistributedSystem("hostfile") };
    GroupManager gm = GroupManager.getGroupManager(config, listeners);
    membershipService =
      (MembershipService) gm.getService(MembershipService.class);
    externalService =
      (ExternalGMIService) gm.getService(ExternalGMIService.class);
    internalService =
      (InternalGMIService) gm.getService(InternalGMIService.class);
    groupProxy =
      (InternalComputeService) gm.getService(InternalComputeService.class);

    membershipService.join();
    externalService.bind("ComputeService");
  }

  // Methods from MembershipListener
  public void viewChange(View v) { /* */ }

  // Methods from MergingListener
  public Object getState(MemberId[] dests) { return null;/* */ }
  public void putState(Object status, MemberId[] sources) { /* */ }

  // Methods from ComputeService
  public void compute(Task t, ResultListener r)
    throws RemoteException, McastRemoteException { /* */ }

  // Methods from InternalComputeService
  public void completed(IID id, Object result) { /* */ }

}
```

Figure 26: The ComputeServer class.

fying the identifier and the result of the completed task. Method completed stores the result in the workload object:

```
public void completed(IID id, Object result) {
  workload.addResult(id, result);
}
```

The Execution Thread

In order to execute the requested tasks, an execution thread needs to be initialized. Aim of this thread is to request a task to be performed from the workload object, execute it and communicate the result to the other servers. The run method of the execution thread look as follows:

```
public void run() {
  while (true) {
    IID id = workload.next();
    Task t = workload.getTask(id);
    ResultListener rl = workload.getListener(id);
    Object result = t.run();
    rl.result(result);
    groupProxy.completed(id, result);
  }
}
```

The first instruction in the loop obtains the identifier of the task to be computed. We assume that method next of Workload is capable to distribute the task executions in order that two servers always reachable between themselves never execute the same task. Once obtained the identifier, we can request the actual task object and the result listener to which the result has to be notified. After the result has been computed, it is transmitted to the client by invoking result on the result listener, and to the other servers by invoking completed on the server proxy containing the object. After these operations, the loop starts again.

Methods of the MembershipListener Interface

When a new view is installed, method viewChange is invoked on each computation server. A new View object is delivered to the application, containing information on which members are in the current view. In our example, method viewChange notifies the Workload object, in order that the workload is redistributed among the members currently operational and reachable. After this, it obtains the list of members in the current view by invoking getMembers on the View object and print it on the console for logging purposes.

```
public Object viewChange(View view) {
  workload.redistribute(view);
  MemberId[] members = view.getMembers();
  for (int i=0; i < members.length; i++)
    System.out.println(members);
}
```

Methods of the MergingListener Interface

In order to use SMS, methods getState and putState of MergingListener must be implemented. Their implementation looks as follows:

```
public Object getState(MemberId[] dests) {
  return workload;
}

public void putState(Object status, MemberId[] sources) {
  workload.merge((Worload) status);
}
```

Method getState is invoked when a server has been elected coordinator for a partition, i.e. responsible for update servers that were previously partitioned. In a production version of this application, only the information needed to update members listed in dests needs to be provided. Here, we simply return the entire workload object. This object is communicated to other servers through the invocation of method putState, which merge the information contained in the received workload with that contained in its local workload.

The Workload class

In this example, we have hidden the application details in the implementation of the Workload class. This class must be able to maintain information about the set of completed task; to store this information on stable storage, if necessary; to distribute the work among operational and reachable servers, avoiding duplicated executions of the same task whenever possible. Aim of this tutorial is to show the basic API of Jgroup, and not to discuss which is the best implementation of the Workload class. Interested reader may download the complete application and/or read related documentation [6].

5.2.2 The Client

The client class (illustrated in Figure 27) is very simple. The code is enclosed in the main method. In the same way as every client accessing a non-replicated remote object, the first action to accomplish is to obtain a reference to the computation service. This is done by obtaining a reference to a dependable registry service through the invocation of static method getRegistry of class RegistryFactory, and then invoking method lookup on this reference. getRegistry requires a string representing the name of an hostfile containing the description of the distributed system in which the dependable registry service is running.

Once obtained this reference, the code of the client does not differ from the code of a client accessing a non-replicated service with the same interface as our computation service. First, a task object is created, using the class name provided in the command line. This task is initialized using the following arguments. Then, a result listener object is created. Class ResultListenerImpl implementing interface ResultListener is illustrated in the same figure. Method result of ResultListenerImpl is invoked by the computation service when the result has been computed; method getResult is invoked by the client and waits until a result is available.

```
import java.rmi.*;
import jgroup.core.registry.*;

public class ComputeClient {

  public static void main(String[] args) throws Exception {

    DependableRegistry reg = RegistryFactory.getRegistry(args[0]);
    ComputeService srv = (ComputeService) reg.lookup("ComputeService");

    Task t = (Task) Class.forName(args[1]).newInstance();
    String[] argv = new String[args.length-2];
    for (int i=0; i < argv.length; i++)
      argv[i] = args[i+2];
    t.init(argv);

    ResultListenerImpl rl = new ResultListenerImpl();
    srv.compute(t, rl);
    Object result = rl.getResult();
    System.out.println(result);
  }
}

public class ResultListenerImpl extends UnicastRemoteObject
  implements ResultListener {

  Object result;

  public ResultListenerImpl() throws RemoteException;

  public synchronized void result(Object result) throws RemoteException {
    this.result = result;
    notify();
  }

  public synchronized Object getResult() {
    while (result == null)
      try { wait(); } catch (Exception e) { };
    return result;
  }
}

public class CalcFactorial implements Task, Serializable {

  private int value;

  public void init(String[] args) {
    value = Integer.parseInt(args[0])
  }

  public Object run() {
    double result = 1;
    for (int i=1; i<=value; i++)
      result *= i;
    return new Double(result);
  }
}
```

Figure 27: The client code.

Once created a result set, the next step is to invoke method compute on the computation service, passing the task and the result listener. Then, the client invokes getResult to obtain the result and print it on the console.

5.2.3 Deploying the Computation Service Application

In order to deploy an application based on Jgroup, several steps needs to be performed. The first step is to start one or more instances of the dependable registry service. This may be obtained by executing the `dregistry` ant target, as follows:

```
ant dregistry
```

Once having started an appropriate number of registry replicas (depending on the degree of fault tolerance needed), one or more computation servers have to be started. Before you can start the compute server, you must add the computeserver target to your build.xml file. To start an application server on a machine, the following command has to be executed from the directory containing the class files of the computation server application:

```
ant computeserver
```

For this simple application, we suggest to use the same hostfile for both the dependable registry replicas and the computation servers. In this way, computation servers may locate the dependable registry service by simply inspecting machines included in their own hostfile. The Jgroup API enables application developers to use different hostfile for the dependable registry service and their applications. It is important to note that even when the hostfiles are the same, it is not necessary that machines running application servers run also a dependable registry replica, or vice versa.

The computation servers form a group and start waiting computation requests from clients. To start a client, the following command has to be executed:

```
ant computeclient CalcFactorial <n>
```

Once again, the `hostfile` contains the description of the distributed system in which application servers can be executed.

The client instantiates a `CalcFactorial` object initialized with a number for which the factorial need to be computed, and invokes method `compute` on the computation server. One of the servers takes care of computing the `CalcFactorial` object, and notifies the result to the provided `ResultListener`. If more than one client requires the execution of a task, the workload is subdivided among the computation servers forming the group.

In order to test the fault tolerance of this application, the reader may try to force the crash of servers forming the group, or use the partition simulator (see Section 6.13 for details) to simulate partitions among computation servers.

6 The Jgroup API

In the previous section, we have illustrated how to use some of the classes and interfaces defined in the Jgroup toolkit. Here we provide the complete specification of the Jgroup API. Additional information can be obtained from the Javadoc documentation included in the Jgroup distribution.

6.1 Taxonomy of the Jgroup API

The interfaces and the classes included in the Jgroup API may be subdivided in three categories: *service interfaces*, *listener interfaces* and *helpers*.

- A service interface specifies the methods that can be invoked by a server in order to access the facilities of a Jgroup service. The MembershipService (Figure 36) is one example of a Jgroup service interface, which is associated to the group membership service and includes methods that can be invoked by servers to join or leave a group.
- A listener interface contains a set of methods that must be implemented by a server in order to receive event notifications from one of the services provided by Jgroup. The MembershipListener (Figure 36) is one example of a listener interface, which includes methods invoked by the group membership service to notify a server that a new view has been installed.
- Helpers are additional classes or interfaces that performs some useful task or maintain some useful information. An example helper class is the member table, which can be used to manage the information about the current state of members with respect to installed views.

Each Jgroup service is associated to exactly one service interface and exactly one listener interface. The interfaces associated to a service may contain no methods, in which case they serve just as a marker. For instance the state merging service, which is based only on event notifications performed through the listener interface, use an empty service interface as a marker.

6.2 Package Structure

The Jgroup API may be subdivided in two main packages: jgroup.core and jgroup.relacs. Package jgroup.core contains the actual Jgroup API, while package jgroup.relacs contains an implementation of the Jgroup specification called Relacs. This separation is motivated by the will of clearly separating the specification from the implementation, in order to enable the use of alternative implementations.

6.3 The GroupManager Class

In order to access the services provided by Jgroup, a server must first notify Jgroup of its existence and provide a listener object for each of the services required by the server. This is

```
package jgroup.core;

public class GroupManager
{
  public Object getService(Class cl)

  public static GroupManager getGroupManager(Object listener)
    throws JgroupException

  public static GroupManager getGroupManager(String configURL, Object listener)
    throws JgroupException
}
```

Figure 28: The jgroup.core.GroupManager class.

performed by invoking static method getGroupManager on the GroupManager class. As shown in Figure 28, this method comes in two variants. Both methods require a listener object (commonly the server object) to be able to construct a group manager. In addition, one of the methods also need a URL pointing to the configuration file, leaving it to the group manager to parse the system configuration. Both methods returns a GroupManager object that can be used (by the server object) to obtain references to layer objects implementing the service interface of the requested services. This can be achieve through the getService method of the group manager.

The format of the XML configuration file specified through the URL string, is detailed in the next section.

The listener (server) object is used to inform Jgroup about the services requested by this server. In order to use a given service, the server object must implement the corresponding listener interface. A server object may implement several listener interfaces, in which case all of the corresponding services will be instantiated; for example, the same object may implement both the GMS and SMS listener interfaces.

Once obtained a group manager object, method getService may be invoked to obtain references to the requested services. To obtain the reference for a particular service, the class object of the service interface must be specified.

6.4 The Configuration Helpers

The current version of Jgroup contains numerous configuration helpers. Here we discuss only two of them; one for configuring the distributed system in which object groups are expected to run, the other is used for configuring the transport layer used by Jgroup to communicate with other servers.

6.4.1 Distributed System Configuration

Upon initialization, the Jgroup runtime needs obtain information about how to locate other Jgroup servers. In general, this information could be in the form of a list of hosts, or a list of network addresses, or a multicast address. The format depends on the Jgroup implementation, and in particular on the transport layer implementation.

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE Configuration SYSTEM "config.dtd">

<Configuration version="1.0">

  <BootstrapRegistry port="1200"/>

  <DependableRegistry locator="jgroup.relacs.registry.RelacsRegistryLocator"/>

  <ReplicaManager name="jgroup.arm.rm.ReplicaManagerImpl" correlationDelay="5000"/>

  <Transport payload="1024" maxTTL="10" TTLWarning="5"/>

  <Services url="file:config/services.xml"/>

  <Applications url="file:config/applications.xml"/>

  <DistributedSystem>
    <Domain name="cs.unibo.it" address="226.1.2.2">
      <Host name="exeon" port="1100"/>
    </Domain>

    <Domain name="item.ntnu.no" address="226.1.2.3" port="6156">
      <Host name="alpha" port="20000"/>
      <Host name="beta" port="30000"/>
      <Host name="gamma" port="40000"/>
    </Domain>
  </DistributedSystem>

</Configuration>
```

Figure 29: The XML configuration for the distributed system.

```
public class DistributedSystemConfig
{
  ////////////////////////////////////////////////////////////////////////////
  // Access methods - client and server side
  ////////////////////////////////////////////////////////////////////////////

  public DomainSet getDomainSet()

  public HostSet getAllHosts()

  ////////////////////////////////////////////////////////////////////////////
  // Access methods - only for server side
  ////////////////////////////////////////////////////////////////////////////

  public Host getLocalHost()
}
```

Figure 30: The class used to obtain information about the distributed system.

In the latest version of Jgroup, the list of domains and hosts of the distributed system is specified using a custom XML format, as shown in Figure 29. The class jgroup.config.ConfigManager is used to parse this configuration file, through the parse() method. Once the distributed system configuration has been parsed, it can also be accessed through the ConfigManager.getConfig() method as follows:

```
/* Obtains configuration information */
DistributedSystemConfig dsc = (DistributedSystemConfig)
  ConfigManager.getConfig(DistributedSystemConfig.class);
TransportConfig transport = (TransportConfig)
  ConfigManager.getConfig(TransportConfig.class);
```

All elements in the XML configuration file has a corresponding class whose name is augmented with Config. For example if the XML tag is titled DistributedSystem, then its associated configuration class will have the name DistributedSystemConfig. As shown in the code sample above, the dsc object will hold the content of the parsed XML data from the DistributedSystem tag. In order to access this data, one needs to know the interface of the DistributedSystemConfig object. The access interface for this configuration class is shown in Figure 30.

The most common access method from the distributed system configuration is the getDomainSet() and getAllHosts(). These methods return a DomainSet and HostSet instance, respectively. They containing the information from the parsed XML configuration file, and the most common thing to do is to iterate over their content in order to probe a given host in the distributed system for availablity or some other specific functionality. However, they provide also a more sofisticated interface, as shown in Figures 31 and 32.

Iterating over either type of set will allow access to either a Domain or Host object. How to access these are shown in Figures 33 and 34. For further details about these classes, please refer to the Jgroup API documentation.

As mentioned above, the DomainSet and HostSet provide more sofisticated functionality, and in particular the addListener() methods enable systems to be built in such a manner

```
public class DomainSet
{
  ///////////////////////////////////////////////////////////////////////////
  // Constructors
  ///////////////////////////////////////////////////////////////////////////

  public DomainSet()

  public DomainSet(int maxEntries)

  public DomainSet(int maxEntries, float loadFactor)

  ///////////////////////////////////////////////////////////////////////////
  // Listener interface
  ///////////////////////////////////////////////////////////////////////////

  public void addListener(DomainListener listener)

  ///////////////////////////////////////////////////////////////////////////
  // DomainSet interface methods
  ///////////////////////////////////////////////////////////////////////////

  public boolean addDomain(String domainName, String mcastAdr, int port)
    throws UnknownHostException

  public boolean addDomain(Domain domain)

  public boolean removeDomain(Domain domain)

  public boolean containsDomain(Domain domain)

  public int size()

  public boolean isEmpty()

  public Iterator iterator()

  public Domain getFirst()
}
```

Figure 31: The DomainSet class used to access the configuration for the distributed system.

```
public class HostSet
{
  //////////////////////////////////////////////////////////////////////////
  // Constructors
  //////////////////////////////////////////////////////////////////////////

  public HostSet()

  public HostSet(int maxEntries)

  public HostSet(int maxEntries, float loadFactor)

  //////////////////////////////////////////////////////////////////////////
  // Listener interface
  //////////////////////////////////////////////////////////////////////////

  public void addHostListener(HostListener listener)

  //////////////////////////////////////////////////////////////////////////
  // HostSet interface methods
  //////////////////////////////////////////////////////////////////////////

  public Host getHost(InetAddress inetAddress)

  public Host getHost(MemberId member)

  public boolean addHost(String hostName, Domain domain, int port)
    throws UnknownHostException

  public boolean addHost(Host host)

  public boolean removeHost(Host host)

  public boolean containsHost(Host host)

  public boolean containsHost(InetAddress inetAddress)

  public int size()

  public boolean isEmpty()

  public Iterator iterator()

  public Host getFirst()
}
```

Figure 32: The HostSet class used to access the configuration for the distributed system.

```
public class Domain
{
  ////////////////////////////////////////////////////////////////////////////
  // Constructors
  ////////////////////////////////////////////////////////////////////////////

  public Domain(String domainName, String mcastAdr, int mcastPort)
    throws UnknownHostException

  ////////////////////////////////////////////////////////////////////////////
  // EndPoint interface methods
  ////////////////////////////////////////////////////////////////////////////

  public InetAddress getAddress()

  public int getPort()

  public boolean isLocal()

  public boolean isMulticastEndPoint()

  public int getIntAddress()

  ////////////////////////////////////////////////////////////////////////////
  // Domain interface methods
  ////////////////////////////////////////////////////////////////////////////

  public String getDomainName()

  public int size()

  public boolean isEmpty()

  public void setLocal(boolean local)

  public HostSet getHostSet()

  public boolean addHost(Host host)

  public Object get(String key)

  public Object put(String key, Object value)

  ////////////////////////////////////////////////////////////////////////////
  // Override methods in Object
  ////////////////////////////////////////////////////////////////////////////

  public boolean equals(Object obj)

  public int hashCode()
}
```

Figure 33: The Domain class used to access the configuration for the distributed system.

```
public class Host
{
  ////////////////////////////////////////////////////////////////////////////
  // Constructors
  ////////////////////////////////////////////////////////////////////////////

  public Host(String hostName, Domain domain, int port)
    throws UnknownHostException

  ////////////////////////////////////////////////////////////////////////////
  // EndPoint interface methods
  ////////////////////////////////////////////////////////////////////////////

  public InetAddress getAddress()

  public int getPort()

  public boolean isLocal()

  public boolean isMulticastEndPoint()

  public int getIntAddress()

  ////////////////////////////////////////////////////////////////////////////
  // Host interface methods
  ////////////////////////////////////////////////////////////////////////////

  public String getCanonicalHostName()

  public String getHostName()

  public String getDomainName()

  public Domain getDomain()

  public Object get(String key)

  public Object put(String key, Object value)

  ////////////////////////////////////////////////////////////////////////////
  // Override methods in Object
  ////////////////////////////////////////////////////////////////////////////

  public boolean equals(Object obj)

  public int hashCode()
}
```

Figure 34: The Host class used to access the configuration for the distributed system.

that hosts and domains can be added and removed from the distributed system. This assumes that the components that handle hosts and domains also implement the required listener interfaces and ensure that they also add and remove items from their local tables. For a more detailed discussion of dynamic host and domain management, see Section 6.4.2.

6.4.2 Dynamic Host and Domain Configuration

Assuming that the system was initialized with a set of domains and hosts within each domain, we may later wish to add new hosts (and domains) to the system at runtime without restarting the system, which would cause unavailability. To facilitate such functionality, the DomainSet and HostSet allows programmers of functionality involving these host configuration classes to become listener to changes (addition and removal) in the host/domain sets.

This is best explained through an example, therefore lets look at the DaemonManager provided with the ARM framework [14]. The DaemonManager passivly monitors hosts within the distributed system, checking if the execution service is available. Suppose that when the DaemonManager was initialized, we did not know all the hosts and domains to be used in the distributed system, but it was needed none the less. To solve this problem the DaemonManager implements the HostListener interface, through the two methods addHost(Host host) and removeHost(Host host). When initializing the internal host tables of the DaemonManager, for each HostSet within the distributed system, we invoke the addListener() method. Thus, if the original host configuration change, one of the two listener methods will be invoked, allowing the DaemonManager to update its internal host tables.

6.4.3 Transport Layer Configuration

Figure 35 shows the jgroup.TransportConfig class, which contains the basic information used to configure the transport layer of any Jgroup implementation. This class contains only one integer field, representing the port number used by the transport layer to communicate with transport layers of other servers in different hosts. Two constructors are provided; the default one constructs a transport layer with the default Jgroup communication port, equal to 28771. The other takes an integer as argument and use it as communication port. Note that if the configuration object array does not contain any TransportConfig object, the default port number is used.

The Relacs implementation includes also a more complex implementation of TransportConfig, called relacs.RelacsConfig (see Figure 35). RelacsConfig enables developers to specify some of the parameters of the transport layer, such as the length of the payload field of UDP packets used by Jgroup to communicate (payload), the number of routing messages that could be lost before a remote host is suspected (lambda) and the time interval between two routing messages sent by the transport layer (delta). More information may be found in the Javadoc API documentation.

```
package jgroup;

public class TransportConfig {

  public TransportConfig();

  public TransportConfig(int port);

  public int getPort();

}

package relacs;

public class RelacsConfig extends TransportConfig {

  public RelacsConfig();

  public RelacsConfig(int port);

  public RelacsConfig(TransportConfig conf);

  public void setPayload(int payload);
  public int getPayload();

  public void setLambda(int lambda);
  public int getLambda();

  public void setAlfa(int alfan, int alfad);
  public int getAlfan();
  public int getAlfad();

  public void setRtimeout(int rtimeout);
  public int getRtimeout();

}
```

Figure 35: The classes needed to configure the transport layer.

6.4.4 Multiple Groups

So far, we have implicitly made the assumption of the existance of a single group in the system, joined by all servers in a distributed system. The reality may be more complex; for example, a server may become member of more than one group; these groups may have different group compositions, but want to share the same transport layer to save the costs of routing and failure detection. Otherwise, different servers may co-exist in the same Java virtual machine, using different distributed systems and transport layers. In this section we explain how configuration helpers may be used to configure these and other scenarios.

First of all, if more than one server co-exist in a Java virtual machine, a separate invocation of method getGroupManager must be performed for each of them. In the same way, a server needing to join more than one group at the same time must invoke getGroupManager once for each of the groups it intends to join.

As explained in the previous section, servers may specify a port number to be used through a jgroup.TransportConfig configuration object. Each specified port number in a Java virtual machine is controlled by a transport layer. Multiple servers in the same Java virtual machine, or a single server joining multiple groups may share the facilities provided by a transport layer by specifying the same communication port when requesting a group manager. In order to use different transport layers, different port numbers must be specified. Note that if no TransportConfig object is specified, the default port number is used; this means that multiple invocations of getGroupManager with no TransportConfig objects lead to multiple group managers sharing the same (default) transport layer.

Given two different invocations of getGroupManager specifying the same port number, the first invocation actually configures the other parameters of the transport layer (such as the distributed system description and the timing parameters described in the previous sections). The second invocation returns a group manager based on the same transport layer, so the other configuration parameters are ignored.

6.5 The Group Membership Service

Figure 36 shows the service and listener interfaces of the group membership service. The facilities provided by GMS may be accessed using the MembershipService interface. Methods in this interfaces enable servers to become members of a group and subsequentially leave it.

As explained above, multiple objects joining different groups may share the facilities provided by a single transport layer. In this case, the group identifier argument of the first version of join enables to distinguish among multiple groups using the same communication port. Otherwise, if each group is associated to a distinct communication port, servers may join the group by invoking the second version of join, without needing to specify a group identifier. In order to subsequentially leave a joined group, method leave could be invoked. Note that an object member may receive event notifications such as view installations for a group even after having request to leave the group itself. When the server eventually leaves the group, method leaved() is invoked on the object(s) implementing the listener

```
package jgroup;

public interface MembershipService {

  public void join(int gid)
    throws JgroupException;

  public void join() throws JgroupException;

  public void leave() throws JgroupException;

  public MemberId getMyIdentifier();

  public MemberTable getMemberTable();

}

public interface MembershipListener {

  public void viewChange(View view);

  public void leaved();

}
```

Figure 36: The interfaces associated to the group membership.

interface. After this invocation, the member will not receive any event notification related to the group.

Each membership service reference may be used to join a single group at a time; in other words, two invocations of the join without a leave between them lead to an exception.

The MembershipService interface defines other two methods; method getMyIdentifier returns a MemberId object that uniquely identifies the server, while getMemberTable returns a MemberTable object that can be used to manage the information about the current state of members with respect to installed views. The functions of these classes are explained in the next section.

Interface MembershipListener contains the methods that must be implemented to receive membership event notifications. When a view change occurs, GMS invokes method viewChange to deliver the new View object to the listener. View objects maintain information on the current installed view, as explained in the next section. As illustrated above, method *leaved* is invoked after a member leaves the group, to acknowledge the effective leaving.

6.6 The View Interface and Related Classes

Installed views consist of a membership list along with a unique view identifier, and correspond to the group's current composition as perceived by members included in the view. The View interface (see Figure 37) enables to obtain the membership list as an array of member identifiers through method getMembers, and the view identifier as a long value through method getVid. Method getGid enables to obtain the identifier of the group to

```
package jgroup;

public interface View {

  int getGid();

  long getVid();

  MemberId[] getMembers();

}

public interface  MemberId {

  public java.net.InetAddress getAddress();

  public int getCounter();

  public boolean isNewer(MemberId id);

  public boolean isNeighbour(MemberId id);

}

public class MemberTable implements MembershipListener {

  public final static int NOTMEMBER   = 0;
  public final static int PARTITIONED = 1;
  public final static int CRASHED     = 2;
  public final static int RECOVERING  = 3;
  public final static int SURVIVED    = 4;
  public final static int MERGING     = 5;
  public final static int NEWMEMBER   = 6;

  public MemberTable();

  public int getState(MemberId id);

  public int getIndex(MemberId id);

  public void put(MemberId id, Object value);

  public Object get(MemberId id);

  public void remove(MemberId id);

  public Object[] elements();

  public MemberId[] members();

  public boolean isMember(MemberId id);

  public void viewChange(View view);
}
```

Figure 37: The View interface and related classes.

which this view is related.

Instances of this class uniquely identify a member object in a group. A member identifier is composed by three parts: an IP address, uniquely identifying the machine hosting the member; the incarnation time of the Jgroup runtime system hosting the member, i.e. the time at which the Jgroup run-time system has been created; and finally, a member counter, which is used to distinguish multiple members running in the same Java virtual machine.

Member ids created in a Java virtual machine share the same IP address and the same incarnation number; thus, identifiers of members hosted in the same Java virtual machine differs only for the member counter. Member ids may be compared in the following ways:

- two members ids are equal (method equals) if and only if they have the same IP address, the same incarnation time and the same member counter;
- two members ids are *neighbor* (method isNeighbour) if and only if they have the same IP address and the same incarnation time; i.e., if they are hosted in the same virtual machine;
- a member id is *newer* than another member id (method isNewer) if and only if their IP addresses are the same and the incarnation time of the former id is greater than the incarnation time of the latter id; in other words, if the Java virtual machine hosting the latter member id has crashed and then recovered.

Apart from methods isNewer and isNeighbour, useful to compare member identifiers, interface MemberId contains also a method getAddress to obtain the IP address of the machine in which the member is hosted, while getCounter returns a counter used to distinguish multiple members running in the same Java virtual machine.

Aim of the MemberTable helper class is to support developers in maintaining information about servers in the group. Member tables can be used as hash tables, to associate application-dependent information to member identifiers. Member tables maintain also information about the state of group members with respect to both the current view and previously installed views. In the computation server example, a member table has been used to store information about other members, and to discover which servers are new servers to redistributed the workload when needed.

Member tables are obtained by invoking method getMemberTable on the group membership service reference, or by using the default constructor of the class. Member tables obtained in the former way are automatically updated by GMS when a new view is installed; otherwise, the application developer must take care of explicitly updating the table by invoking method viewChange on it.

A table maintains information about a member until it crashes. A member is declared crashed when a member identifier from the same host, but with an higher incarnation number is inserted in the table. This means that the JVM hosting the member has crashed, and a new one has started. The information associated to the member is maintained until the next view change, after which the member identifier and all its associated information is removed from the table.

```
package jgroup;

public interface  MergingListener {

  public Object getState(MemberId[] dests);

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

}
```

Figure 38: The interfaces associated to the state merging service.

Method getState returns the state associated to the specified member identifier. The returned value is one of the integer constants defined in the class; possible states are *not member*, *crashed*, *partitioned*, *recovering*, *survived* (from the previous view), *merging*, *new member*. Method getIndex returns the index of the specified member identifier in the view composition array obtained by invoking method getMembers on the current view; returns -1 if the member is not included in that view. Method isMember returns true if and only if the specified member identifier is included in the current view.

Method put associates an object to the specified member identifier. This association is maintained in the table until it is explicitly removed from the table using method remove, or the member is declared crashed. Values may be subsequentially retrieved using method get, which returns the value associated to the specified member id. Returns null if the table contains no mapping for this member id. A return value of null does not necessarily indicate that the table contains no mapping for the key; it is possible that the table explicitly maps the key to null. Method remove removes the object associated to the specified member identifier. Finally, methods elements and members return an array containing the values associated to member identifiers and the members identifiers contained in the member table, respectively.

6.7 The State Merging Service

Figure 38 shows the listener interface of the state merging service. Differently from other facilities included in Jgroup, the SMS command interface is empty. Applications using SMS must provide an object implementing the listener interface, and can only receive notifications from SMS.

When a server is elected coordinator for the state merging protocol, method getState is invoked on its state merging listener. The listener must respond with an object containing a snapshot of its current state. This snapshot may be complete, i.e. contain all information about the state, or may be partial, i.e. contain only the information needed to update servers contained in dests. The choice between returning a complete or partial state is application-dependent, and in particular depends on the size of the state to be returned. The dests array contains the servers that have been partitioned and have not been updated yet.

The state returned by a coordinator is delivered to the other application servers by

```
package jgroup;

public interface ExternalGMIListener extends Listener, Remote {}

public interface InternalGMIListener extends Listener {}

public interface ExternalGMIService {

  public IID getIdentifier();

  public IID bind(String name)
    throws RemoteException, AccessException;

  public IID bind(String name, DependableRegistry registry)
    throws RemoteException, AccessException;

}

public interface InternalGMIService {

  void invoke(Method m, Object[] args, Callback callback)
    throws Exception;

}
```

Figure 39: The interfaces associated to the GMI service.

invoking method putState on the listener objects associated to them. Method putState has two arguments, one containing the object returned by the coordinator when completing method getState, the other containing the list of members for which the coordinator acted as a representative.

6.8 The GMI Service

Figure 39 contains the definition of both the listener and command interfaces for the external and internal GMI services. Listener interfaces ExternalGMIListener and InternalGMIListener are two tag interfaces used to mark application-defined interfaces as external or internal, respectively. They do not contain any method to be implemented.

6.8.1 The External GMI Service

ExternalGMIService is the command interface of the external GMI service. Method getIdentifier returns the *invocation identifier* of the external invocation currently executed. If, on the other hand, getIdentifier is invoked outside an external invocation, null is returned. Invocation identifiers uniquely identify invocations performed by clients, and are used by the external GMI service to detect and discard duplicate executions of the same method on the same server. Application developers may use invocation identifiers to identify operation performed in different partitions. Each identifier is composed of the client identifier (the IP address of the machine hosting the JVM), an *incarnation number* (used to distinguish different virtual machines residing at the same host) and an invocation counter (incremented

```
package jgroup;

public interface  Callback {

  public void result(MemberId id, Object result);

  public void exception(MemberId id, Exception e);

}
```

Figure 40: The Callback interface.

at each invocation). Interface IID, together with interface VMID can be used to access the information contained in an invocation identifier. Further details about these interfaces may be found in the Javadoc documentation of package jgroup.

The other methods contained in ExternalGMIService enables servers to be bound in a dependable registry service. The first version of method bind assumes that the dependable registry services is located in the same distributed system of the application servers, while the second version takes an explicit (remote) reference to a dependable registry service. Both methods take the name under which the application server must be bound as an argument.

6.8.2 The Internal GMI Service

InternalGMIService is the command interface of the internal GMI service. The only method defined in this interface enables servers to perform asynchronous invocations of internal methods. invoke takes three arguments. The first one is the method to be invoked; only methods belonging to an interface extending InternalGMIListener can be invoked. The second argument is an array of objects containing the parameters of the invocation. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subject to widening conversions as necessary. The third argument is a *callback* object which will receive the return value upon completion of the invocation. The callback object must be written by the application developer and must implement the Callback interface illustrated in Figure 40.

Callback defines two methods, result and exception. The former is invoked when the method has been executed correctly with a regular return value, while the second is invoked when an exception has been generated during the method execution. For both methods, the first argument is the identifier of the member that executed the method.

6.9 The Client and Server Side Proxies

In order to enable clients and servers to use a uniform method call interface, even for communication with a group of objects, Jgroup makes extensive use of so called proxies. You may think of a proxy object is a replacement for a real object. Using a proxy, requires that the proxy object be invoked instead of the real object, and in Jgroup this facility is used to invoke multiple "real" objects. In particular, Jgroup makes use of dynamic proxy generation. Also note that, the word "stub" is often used interchangably with the term proxy.

A server proxy for an object group is a server-side entity that contains a method which dispatches calls to the actual server implementation. Moreover, the server proxy is also a object group stub which is responsible for forwarding internal method invocations to the servers forming the group.

A client proxy is a object group stub which is responsible for forwarding external method invocations to the servers where the actual servers forming the object groups reside. A client's reference to an object group, therefore, is actually a reference to a local client proxy.

The server proxy implements only the internal GMI interfaces implemented by the server, while the client proxy implements only the external GMI interfaces. Interfaces not extending ExternalGMIListener and InternalGMIListener are not considered by the dynamic proxy generation facility of Jgroup. Because the client (server) proxy implement exactly the same set of external (internal) GMI interfaces as the object group itself, a client can use the Java language's built-in operators for casting and type checking.

6.10 The Dependable Registry Service

A dependable registry is a bootstrap naming service which is used by GMI servers to bind server to group names. Clients can then look up object groups and make external group method invocations.

The `dregistry` command starts a dependable registry instance on the local machine. The command produces no output and is typically run in the background.

The syntax for the `dregistry` command is the following

```
ant dregistry
```

In order to programmatically access the dependable registry service, the DependableRegistry and RegistryFactory classes in the jgroup.registry package may be used. These classes are similar to the Registry and LocateRegistry classes contained in the java.rmi.registry package included in the SDK. They can be used to get a dependable registry replica operating on a particular host. Interested readers may refer to the Jgroup API documentation for further information on these classes.

6.11 Greg: the Group-Enabled Lookup Service

As mentioned above, in Jgroup 1.1 it is possible to use Greg instead of the dependable registry to obtain proxies for object groups. Greg derives from Reggie, the Sun's reference implementation of the Jini lookup service, and extends it to manage the registration of group proxies.

Reggie enables registration of customized proxies for services. This feature could be used to register group proxies through any implementation of the lookup service. Group proxies, however, differ from standard proxies as their contents may be dynamic. A server registering in a lookup service must not overwrite existing information about previously registered group servers. Instead, it must add its information to an existing group proxy. Furthermore, when a server crashes or becomes partitioned, and fails to renew the lease obtained from the lookup service when registering, the information about it has to be removed from the group proxy, clearly without removing the entire proxy. These considerations lead us to develop an alternative implementation of the lookup service, in which group servers register their information in a group proxy by specifying a group name attribute.

In order to start greg, other services must be started. First of all, greg is an activatable service; this means that it requires an RMI daemon running on the same machine, which has to be started through the `rmid` command. Furthermore, an http daemon must be able to provide the jar file containing the code of remotely loaded classes. The `readme.txt` file contained in the Jgroup distribution explain how to start greg.

6.12 The Reliable Multicast Service

Internal group method invocations are not the only way for servers to communicate; when a simpler (and more efficient) communication mechanism is needed, Jgroup provide also a reliable multicast service (RMS) which can substitute the internal group method invocation service. Classes related to the RMS are contained in package jgroup.multicast. For those interested in using RMS, please refer to the RMS specification [16] and to the Jgroup API documentation.

6.13 The Partition Simulator

For more information on the partition simulator, please refer to the source code (relacs.simulator and relacs.mss packages). Further information will be provided in future version of this manual.

References

[1] 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.

[2] The jakarta site - ant. http://jakarta.apache.org/ant/.

[3] Ant User Manual. http://jakarta.apache.org/ant/manual/index.html.

[4] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison-Wesley, 1999.

[5] 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.

[6] 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.

[7] Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001.

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

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

[10] 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.

[11] Object Management Group. Fault Tolerant CORBA Using Entity Redundancy. OMG Request for Proposal orbos/98-04-01, Object Management Group, Framingham, MA, April 1998.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 2.3.* Object Management Group, Framingham, MA, June 1999.

[13] 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.

[14] Hein Meling and Bjarne E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, Italy, May 2001.

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

[16] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems.* PhD thesis, Department of Computer Science, University of Bologna, February 2000.

[17] Sun Microsystems, Mountain View, CA. *Java Remote Method Invocation Specification, Rev. 1.7*, December 1999.

[18] Sun Microsystems, Mountain View, CA. *Enterprise JavaBeans Specification, Version 2.0*, August 2001.

[19] 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.