The Jgroup Reliable Distributed Object Model

Alberto Montresor *

Abstract

This paper presents the design and the implementation of Jgroup, an extension of the Java distributed object model based on the group communication paradigm. Aim of Jgroup is to support the development of dependable applications in partitionable distributed systems. Jgroup consists of a partitionable group communication service that simplifies the cooperation among groups of replicated server objects, and a client-side mechanism to transparently invoke methods on object groups as if they were single, non-replicated entities.

Keywords: Distributed Systems, Dependability, Object Groups, Group Communication, Java RMI

1 Introduction

In the last decade, the distributed object technology has proven to be a successful paradigm in dealing with the increasing complexity of distributed systems. Examples of popular distributed object frameworks are CORBA [13] and Java Remote Method Invocation (RMI) [23, 18]. These middleware platforms enable distributed objects to interact using a client/server approach similar to the remote procedure call paradigm, but opportunely adapted for object-oriented systems. Client objects are allowed to access the services provided by server objects by issuing remote method invocations on them. Each server presents a well-defined interface that describes the set of methods that can be remotely invoked by clients. All the low-level details of remote invocations (e.g. marshaling and unmarshaling arguments and results) are handled by local surrogate objects, that present the same interface as their remote counterparts and act as proxies for them.

The distributed object models listed above focus their attention on improving portability, interoperability and reusability of distributed software components and applications. Unfortunately, none of them provide an adequate support for the development of reliable and high-available applications. This constitutes a major drawback for many modern industrial applications, for which requirements such as reliability and high-availability are gaining increasing importance. In the absence of any kind of systematic support, building applications capable to deal with partial failures such as process crashes and network partitionings is an error-prone and time-consuming task.

In order to overcome these difficulties, the *object group* paradigm [16] has been proposed. In the object group approach, the functionalities of a remote server are replicated among a

*Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, Bologna 40127 (Italy), Tel: +39 51 354871, Fax: +39 51 354510, Email: montresor@CS.UniBO.IT

group of objects. Clients interact with object groups in a transparent way, as if they were single, non-replicated entities. Replicated objects forming a group cooperate in order to provide a reliable and high-available service to their clients. This cooperation is established through the facilities offered by a group communication service (GCS) [7, 22, 1, 20, 3], that enables the creation of dynamic groups of objects that communicate through reliable multicast primitives. Objects forming a group are kept informed about the current membership of the group itself, that may vary on run-time due to accidental events such as failures and repairs, or to voluntary requests to join or leave the group. Examples of distributed object models based on the object group paradigm are Orbix+Isis [14], Electra [15, 14], Object Group Service (OGS) [11] and Filterfresh [5]. Apart from Filterfresh, that is a group-enhanced extension of the Java distributed object model, the others combine the benefits of CORBA with the reliability provided by a group communication service.

These group-oriented distributed object models are based on *primary-partition* GCSes [7]. The primary-partition approach is intended for systems with no network partitionings, or for systems that require that the membership of a group is allowed to change in at most one network partition, the so-called "primary partition". This is a serious limitation for modern large-scale distributed systems, characterized by highly partitionable communication networks. Partitionings tend to become more frequent and longer-lasting as the geographic extent of the system grows, or its connectivity weakens due to the presence of mobile units or wireless links. Applications based on a primary-partition approach cannot guarantee continued availability outside the primary partition; moreover, particular failure scenarios may cause the complete blocking of a primary-partition GCS [17], and consequently the complete blocking of applications based on them.

In order to overcome these problems and to provide a systematic support for the development of dependable applications in partitionable systems, we have designed Jgroup, an extension of Java RMI based on a *partitionable* GCS [3]. The partitionable approach to group communication provides replicated objects with the capability of carrying on the computation and being available in multiple concurrent partitions. In the Jgroup distributed object model, replicated server objects presenting the same interface are gathered into groups; these groups simulate the behavior of single, non-replicated remote objects by presenting the same interface and allowing clients to invoke their methods through the standard Java RMI semantics. Consistency among replicated servers forming a group can be guaranteed by implementing opportune consistency protocols based on the group communication primitives provided with Jgroup.

Goal of this work is to present the design and the implementation of Jgroup. The paper is organized as follows. Sections 2 and 3 recall some background on the group communication paradigm and Java RMI. Section 4 describes the Jgroup distributed object model, while Section 5 contains few notes about the Jgroup implementation. Section 6 compares this work with similar projects. Finally, conclusions are presented in Section 7.

2 The Group Communication Paradigm

The group communication paradigm [6] allows the provision of reliable and high-available applications through replication. *Groups* are the key abstraction of group communication. A group consists of a collection of *members* (i.e., processes or objects) that share a common

goal and actively maintain a replicated state.

During the last years, several experimental and commercial group communication services have appeared [7, 22, 1, 20, 3]. Although the services provided by these systems present several differences, the key mechanisms underlying their architectures are the same: a *group membership service* integrated with a *reliable multicast service*. Task of a group membership service is to keep members consistently informed about changes in the current membership of a group through the *installation* of *views*. The membership of a group may vary due to voluntary requests to join or leave a group, or to accidental events such as failures and repairs of both the computing system (member crashes and recoveries) and the communication system (network partitioning and mergings). Installed views consist of a collection of members and represent the perception of the group's membership that is shared by its members. In other words, there has to be agreement among the members on the composition of a view before it can be installed. Task of a reliable multicast service is to enable the members of a group to communicate by multicasting messages. Message deliveries are integrated with view installations as follows: two members that install the same pair of views in the same order deliver the same set of messages between the installations of these views. This delivery semantics, called *view synchrony*, enables members to reason about the state of other members using only local information such as the current view composition and the set of delivered messages.

As noted in the introduction, two classes of GCS have emerged: *primary-partition* [7] and *partitionable* [22, 1, 20, 3]. A primary-partition GCS attempts to maintain a single agreed view of the current membership of a group. Members excluded from this view are not allowed to participate in the distributed computation. In contrast, a partitionable GCS allows multiple agreed views to co-exist in the system, each of them representing one of the partitions in which the network is subdivided. Members of a view are allowed to carry on the distributed computation separately from the members not included in the view. Primary-partition group communication services are suitable for non-partitionable systems, or for applications that need to maintain a unique state across the system. Partitionable systems are intended for applications that are able to take advantage of their knowledge about partitionings in order to make progress in multiple, concurrent partitions. Applications with these characteristics are called *partition-aware* [4]. Examples can be found in areas such as computer-supported cooperative work (CSCW), mobile systems, weak-consistency data sharing.

3 The Java Distributed Object Model

Java RMI is a distributed object model that maintains the semantics of the Java object model, making distributed objects easy to implement and to use. *Remote objects* are characterized by the fact that their methods can be invoked from other Java virtual machines, potentially on different hosts. Given a remote object class, the set of its methods that can be remotely invoked is defined by one or more *remote interfaces*. Clients of a remote object never interact with the actual implementation class of this object, but only with a local surrogate object that presents the same set of remote interfaces.

The Java RMI architecture is illustrated in Figure 1 and consists of three layers: the *stub/skeleton layer*, the *remote reference layer* and the *transport layer*. The stub/skeleton layer is the interface between applications and the rest of the RMI system; moreover, it is responsible for

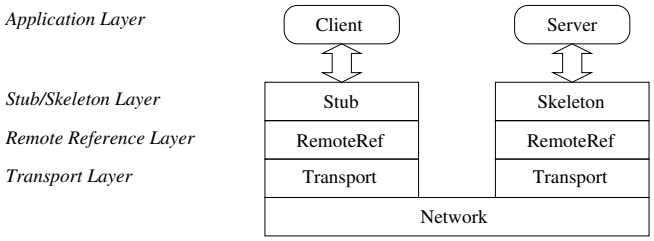

Figure 1: The Java RMI Architecture

marshaling and unmarshaling the invocation parameters and the return values. Clients invoke methods of a remote object through a *stub*, which acts as a proxy for the remote object. The stub implements the same remote interfaces as the remote object, and forwards each invocation request to the remote object through the remote reference layer. On the server side, a *skeleton* object dispatches the requests coming from the remote reference layer to the corresponding methods of the remote object. The *remote reference layer* is responsible for the semantics of the invocation. The current version of Java RMI includes only two unicast (point-to-point) invocation mechanisms, one relative to servers always running on some machine, and one relative to servers that are activated only when one of their methods is invoked. Finally, the *transport layer* is responsible for all the low-level details such as connection management and invocation request transmission.

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

4 The Jgroup Distributed Object Model

The Jgroup distributed object model (Figure 2) is based on two fundamental abstractions: *remote object groups* and *replicated remote objects*. From the server's point of view, a remote object group consists of a collection of replicated remote objects (sometimes called *replica* for brevity) that implement the same set of remote interfaces and coordinate their executions in order to appear as a non-replicated remote object. Replicas forming a remote object group cooperate using a partitionable GCS, whose task is to simplify the development of the consistency protocols needed to offer a reliable and high-available service.

Clients have no access to single replicated remote objects and interact only with remote object groups. From the client's point of view, remote object groups are not distinguishable from standard remote objects. Each group implements one or more remote interfaces, whose methods can be invoked using the RMI mechanism: clients obtain a local stub that presents the same set of interfaces and acts as a surrogate of the remote object group. Every method invocation on the local stub will correspond to a remote method invocation on one or more of the replicas forming the group, depending on the particular invocation semantics adopted.

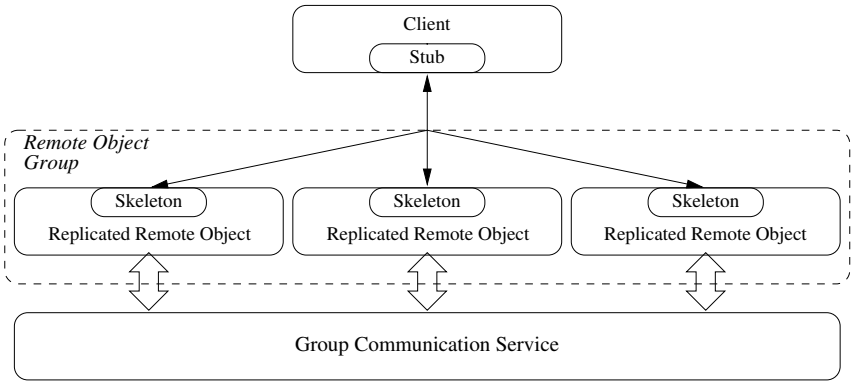

Figure 2: The Jgroup Architecture

The rest of this section is dedicated to two fundamental components of the Jgroup architecture: on the server side, our partitionable GCS; on the client side, the transparent, fault-tolerant invocation mechanism. The section is concluded by a description of the *dependable registry*, a reliable and high-available application for registering replicated remote objects and retrieving stubs for remote object groups.

4.1 The Group Communication Service

As introduced in Section 2, the partitionable GCS included in Jgroup can be subdivided in two components: a partitionable group membership service and a reliable multicast service. In another paper, we provide a formal specification for both of them [3]; here, we discuss briefly their main characteristics and how they can be used to implement replicated remote objects.

In order to cooperate with its peers, a replicated remote object has to become *member* of a remote object group g through a *join(g)* request. After having joined a group g, replicated remote objects may leave it through *leave(g)* requests. The partitionable group membership service notifies members of changes in the membership of a group g through a *vchg*(g, v) event, that corresponds to the installation of a view v. Each view is given a unique identifier and consists of a collection of replicated remote objects. Members must be able to react opportunely to new failure scenarios as depicted from views, for example by modifying the quality of the service they provide or trying to recover from previous failures. The fact that our partitionable group membership service allows the existence of multiple concurrent views must be taken into particular consideration. As an example, consider the implementation of a shared blackboard for supporting cooperative work sessions among groups of users. The installation of a new view due to a new partitioning may require the display of a warning message informing the users that the state of the blackboard will evolve inconsistently in distinct partitions. On the other hand, the installation of a new view due to the merging of two partitions may cause the execution of a state reconciliation protocol on the contents of the blackboard.

In our specification, the correspondence between the failure scenario and the views installed by the group membership service is guaranteed by the View Completeness and View Accuracy properties [3]. View Completeness forces crashed or partitioned members to be excluded from

installed views, while View Accuracy forces the inclusion of operational members with which it is possible to communicate. Another important property is View Agreement, that requires that before a view can be installed, members composing it must have reached an agreement on its composition. Finally, the View Order property states that members installing the same pair of views must install them in the same order.

The reliable multicast service enables replicated remote objects to multicast a message m to the members of a group g through a *mcast*(g, m) request. The delivery of a message m relative to the group g is notified to members through a *dlvr*(g, m) event. In the shared blackboard example, the reliable multicast service can be used to report every change in the contents of the blackboard to all the users participating in a cooperative work session. As introduced in Section 2, the view synchrony semantics guarantees that two blackboards surviving together from a view to another one have delivered the same set of messages between the two installations, and thus display the same contents.

In our specification, the view synchrony semantics is enclosed in two properties, called Message Agreement and Uniqueness. The Message Integrity property places simple integrity requirements to prevent a message from being delivered multiple times or without having been multicast. Finally, Message Liveness specifies under which conditions a multicast message is delivered.

4.2 Remote Method Invocation Semantics

Method invocations on remote object groups may be executed following two different semantics. The *reliable unicast* invocation semantics guarantees that a method invocation performed by a client on a group of replicas will be executed by invoking the same method on at least one of the replicas, unless the partition of the client does not contain any operational replica. On the other hand, the *reliable multicast* invocation semantics guarantees that a method invocation performed by a client on a group of replicas will be executed by invoking the same method on every replica contained in the client's partition, and that invocations on replicas are synchronized with view installations in the same way as multicast messages.

The reliable unicast semantics is suitable for methods whose execution can be performed on a single replica, without involving the entire group (e.g., read-only methods for interrogating a replicated database, such as the `lookup()` method of a distributed registry). On the other hand, the reliable multicast semantics is suitable for methods whose execution must affect every replica of a group (e.g., methods that alter the state of a replicated database, such as the `bind()` method of a distributed registry). It is important to note that clients may be unaware of the invocation semantics, while the replica implementation depends on it: for example, using a reliable unicast semantics implies that methods altering the state of a database must multicast the update to every replica in the group.

The invocation of the same method on multiple replicas poses interesting problems regarding the handling of multiple return values from non-void methods. We have chosen to return the value obtained from the first concurrent invocation to conclude. The main advantage of this approach is transparency, since clients obtain a single value as if they were invoking a method of a non-replicated object. Note that applications must be implemented in order to guarantee that values returned from the replicas forming a group are idempotent. For this reason, an alternative approach may be to return an array containing each return value obtained from

the replicas. Unfortunately, this approach is not transparent with respect to clients, since remote interfaces and stubs must be opportunely modified in order to return array of objects.

The current version of Jgroup includes only the reliable unicast semantics: when a method of a remote object group is invoked, the replicas forming the group are sequentially invoked, until a reachable replica is found (in which case the result is returned to the client), or none of the replicas can be contacted (in which case an exception is raised). This invocation mechanism is completely transparent to the client. Note that due to the *Two Generals Problem* [12], we cannot guarantee that a method is invoked on at most one of the replicas. Consider the following scenario in which the same method is executed on two replicas: the first contacted replica receives the invocation request, executes the selected method, but is unable to deliver the return value due to a sudden partitioning. At this point, the invocation mechanism on the client contacts another replica that correctly executes the method.

4.3 The Dependable Registry

As described in Section 3, the Java RMI architecture requires a repository facility for registering and retrieving remote objects. Unfortunately, the registry implementation included in the current JDK is not suitable for remote groups. First of all, it does not provide a reliable and high-available service: registry instances are not replicated and maintain different set of bindings, thus constituting single points of failures. Moreover, remote objects running on a certain host can be registered only in a registry executed on the same host; this precludes the possibility of registering groups of replicated remote object concurrently running on several different hosts.

For these reasons, the Jgroup distributed object model includes a dependable registry service, implemented as a collection of replicated remote objects forming a remote object group. Each replica maintains a set of bindings $\langle$*name*, *remote object group*$\rangle$. The *bind* primitive is used to add a replica to the remote object group registered under a certain name; in other words, the dependable registry enables a set of replicas to bind themselves under the same name in order to form a remote object group. The *lookup* primitive is used to retrieve the stub of a remote object group by name.

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

During a partitioning, a dependable registry presents a partitioned behavior reflecting the current failure scenario. A *bind* primitive executed inside a partition will not affect the replicas not contained in that partition, while a *lookup* primitive will not be able to retrieve bindings registered outside the current partition after the beginning of the partitioning. Nevertheless, replicas contained in a partition consistently maintain the same set of bindings and act as a single entity; moreover, the disappearance of the partitioning causes the execution of a reconciliation protocol in order to re-establish a consistent set of bindings among the replicas

that belonged to different partitions. It is important to note that this behavior is perfectly suitable for a partitionable distributed system, since clients asking for remote services are interested only in servers running in their partitions.

5 Jgroup Implementation

Jgroup is completely written in Java and consists of three different components: a partitionable GCS, the invocation mechanism and the dependable registry.

5.1 The Group Communication Service

The partitionable GCS included in Jgroup is based on the Relacs algorithm [3]. For sake of brevity, in this section we provide only an overview about the main characteristics of the Java implementation of Relacs; a detailed description of the algorithm and a proof of its correctness can be found in the relative paper. Before starting, it is important to note that the Relacs implementation can be exploited separately from the Jgroup remote method invocation model. In other words, applications based on Relacs will be reliable and high-available, but will not be able to be invoked as remote object groups.

Our Java implementation of Relacs is based on a daemon/client model where a set of *Relacs daemons* (implemented in the `RelacsDaemon` class) provides the Relacs partitionable group communication service to a set of remote objects. For each Java virtual machine, there could be at most one Relacs daemon, whose functionalities are shared by each remote object running in the same JVM. The use of this model, as opposed to having the group communication service provided by every object joining a group, minimizes the number of messages exchanged to establish the group communication service.

Remote objects that want to participate in a distributed computation using Relacs must implement the `Member` interface. Methods defined in `Member` are callbacks that will be invoked to notify the member object about group-related events such as view installations (method `vchg(int g, View v)`) and message deliveries (method `dlvr(int g, Msg m)`).

Member objects do not interact directly with the Relacs daemons running in their JVM; instead, the interactions between a member and the corresponding daemon are handled by a *group manager*. Group managers are defined in the `GroupManager` interface and are obtained through the `GMFactory` class. Every member object is associated to one and only one group manager. Task of a group manager is to provide its associated member with the group communication functionalities needed to interact with groups. Each group manager runs its own thread of control that is responsible for the invocation of the callback methods on its member. The member may invoke the group manager in order to join and leave a group, or to multicast a message to a group. The `join(int g)` method of a group manager is used to join the associated member object to the group identified by g; `leave(int g)` is used to remove the member from the group g; finally, `mcast(int g, Msg m)` is used to multicast a message to the members of g.

Relacs daemons are the heart of the group communication protocol and consist of three components: a transport service, a partitionable group membership service and a reliable multicast service. The transport service is based on UDP and includes a routing algorithm

fully integrated with a failure detector [9]. The routing algorithm is needed to overcome the lack of transitivity in the communication network that can be sometimes experienced in large-scale distributed systems like the Internet [17]. The failure detector is needed to guarantee the termination of the group communication protocol and is based on the information obtained from the routing algorithm. The group membership service and the reliable multicast service exploit the functionalities offered by the transport service in order to implement the specification outlined in Section 4.1. Every variation in the failure scenario, like a new partitioning or a merging, causes the underlying failure detector to modify its list of suspected daemons. The new suspect list is notified to the group membership service, which starts executing an agreement protocol among the survived daemons. The agreement protocol is coordinator based: for every group interested by the variation in the failure scenario, each daemon sends its estimate of the group composition to a coordinator deterministically chosen from that estimate. When the coordinator observes an agreement among the daemons composing an estimate, the estimate is translated in a view and sent to every daemon interested in that view. The termination of the algorithm is guaranteed by the fact that during an agreement protocol, the estimates sent by a daemon are monotonically decreasing. When a daemon receives a new view from the coordinator, it notifies the group managers interested in the view, which inform their associated members by invoking the opportune callback method.

5.2 The `ReplicatedRemoteObject` Class

The integration between our group communication model and the Java RMI system is based on the `ReplicatedRemoteObject` class, the superclass of each replicated remote object. This class plays two different roles: it is a remote object, and thus extends a set of remote interfaces; it is a replicated object, and thus extends the `Member` interface. The twofold nature of a replicated remote object is reflected on the operations performed by the class constructor. As a remote object, it announces its existence to the local remote reference and transport layers of the RMI system through the `export()` method of the `UnicastRemoteObject` class. As a member object, the class constructor instantiates a group manager that will provide the replicated remote object with the group communication primitives described in Section 5.1.

5.3 The `GroupRef` Class

As described in Section 3, a method invocation on a remote object is executed by invoking the same method on a local stub that acts as a surrogate of the remote object. Each stub contains a *remote reference* that identifies the remote object and is responsible for the invocation protocol. When a stub method is invoked, the stub calls the `invoke()` method of its remote reference, passing an identifier for the method and the associated invocation arguments to it. Thus, the remote reference owned by the stub is the core of the invocation mechanism.

In order to implement the reliable unicast invocation semantics for remote object groups, we have implemented the `GroupRef` class. Group references differ from unicast remote references because they contain a remote reference for each of the replicas forming a remote object group. When the `invoke()` method of a group reference is called, one of the replicas contained in the group reference is selected, and the corresponding `invoke()` method is called. If the invocation on the selected replica terminates, the result value is returned to the client application. On the other hand, if the invocation on the selected replica raises an exception due to a crash or

a partitioning, another remote reference is selected and the corresponding `invoke()` method is called. This process continues until one of the invocations on the replicas successfully completes, or the list of remote references terminates. In this case, an exception is raised to the application layer.

5.4 The Dependable Registry

The group reference description contained in the previous section does not explain how group references are created. This basic task is performed by the dependable registry provided with Jgroup, which builds a group reference by collecting the remote references of a set of replicated remote objects registered under the same name. In this section, we provide a brief overview about the dependable registry implementation. The complete algorithm can be found in a companion work [19].

The `DependableRegistry` interface defined in the Jgroup API contains two kinds of methods: retrieval methods such as `list()` and `lookup()`, and update methods such as `bind()`, `rebind()` and `remove()`. Since each registry replica maintains its copy of the set of bindings, the retrieval methods can be executed locally: the replicated registry instance that receives the method invocation inspects its local set of bindings and returns an appropriate result value. The behavior of update methods is different, since they involve the update of all replicas forming the dependable registry group. When one of the replicas receives an update method invocation, it multicasts an update message to the replicas in its current view. Replicas that deliver an update message modify their local copy of the set of bindings. Given the possibility that the same remote method invocation is executed on more than one replica (see Section 4.2), each invocation request is tagged with an unique identifier. This avoids incorrect scenarios in which a duplicated and delayed bind operation is executed after a subsequent remove operation.

At this point, we can conclude the description of our algorithm by analyzing three different scenarios:

- When the system is stable (i.e. when no new failures or repairs occur), each network partition contains a set of replicas that will eventually install the same view and deliver the same set of multicast messages. This implies that each update operation invoked inside that partition will be eventually performed by every replica in that partition.
- In case some of the replicas become partitioned or not operational, the GCS installs a new view containing the surviving replicas. Due to the view synchrony semantics, all replicas in the new view have delivered the same set of update messages during the previous view. If the replicas of the new view maintained an identical set of bindings at the beginning of the previous view, they maintain an identical set of bindings also at the beginning of the new view. This implies that no additional operation is needed in case of crashes or partitionings.
- When two or more partitions merge into a common one, replicas belonging to distinct partitions may have inconsistent set of bindings. For this reason, a state reconciliation protocol is needed. This protocol [19] is based on the election of a coordinator for each of the merging partitions, whose task is to multicast a restore message containing a

compact representation of the update operations performed during the previous partitioning. Each coordinator acts on behalf of all the replicas contained in its previous view; this reduces the number of multicast messages exchanged to restore a global state. When a replica delivers a restore message, it updates its set of bindings.

6 Discussion and Related Work

In the last few years, the problem of integrating the group communication paradigm with distributed object technologies such as CORBA [13] and Java RMI [23, 18] has been the subject of intense investigation [15, 10, 14, 5].

Electra [15, 14] and Orbix+Isis [14] are two CORBA object request brokers (ORB) that support the implementation of reliable distributed applications based on group communication. The former is a commercial product that integrates Orbix, a CORBA-compliant development environment, with the group communication facilities of ISIS [7]. The latter is an academic ORB whose implementation is based on group communication platforms such as ISIS [7] or Horus [22]. Both toolkits allow programmers to treat collections of CORBA objects as if they were a single entity, and clients invoke operations on object groups without needing to know the membership of the group.

Electra and Orbix+Isis are based on the *integration* approach [10], which consists in modifying and extending an ORB with group communication mechanisms. The integration approach is appealing for its transparency, since clients do not need to know if the service they require is provided by a single object or a group; nevertheless, the resulting ORBs are not CORBA-compliant. An alternative methodology is the service approach [10], which consists in providing group communication as a service on top of the ORB. Clients hold references to OGS services, whose task is to provide primitives to communicate with groups of objects. An example of object group toolkit based on the service approach is the Object Group Service [11]. Although the service approach is CORBA-compliant, clients cannot transparently access a group of objects as if they were a single entity, but they must use the primitives offered by the OGS.

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

iBus [**?**] is a commercial product written in Java and aimed at supporting intranet applications such as content delivery systems, groupware and fault-tolerant client/server systems. The iBus architecture does not integrate the group communication paradigm with the standard Java RMI architecture; instead, iBus is based on the concept of multicast channels mapped on IP multicast groups. Clients can subscribe to multicast channels and can push and pull messages over a subscribed channel. Unfortunately, the current version of the protocol does not provide view synchrony; in other words, there is no guarantee that pull operations (i.e. message deliveries) are synchronized with view changes.

Filterfresh [5] and Jgroup share the same goal, i.e. the integration of the group communication paradigm with the Java distributed object model. Due to the constraints inherent

to Java RMI, the approaches they follow are similar: both offer a reliable invocation mechanism for remote object groups composed by a collection of remote objects that cooperate through a GCS, and a distributed implementation of the RMI registry. Nevertheless, there are also important differences. Filterfresh group references contain a reference for only one of the remote objects forming a group. If this remote object is partitioned from the invoking client, the `invoke()` method of the group reference asks the reliable registry for the reference of another remote object belonging to the group. This means that each invocation failure requires an additional RMI interaction with the reliable registry. This slows down the invocation mechanism in highly partitionable distributed systems such as large-scale and wireless networks. In contrast, Jgroup group references contain a reference for each of the replicas forming a group and are able to independently select an operational replica in their partition. Another difference is in the replication degree of the distributed registry. In Filterfresh, each host containing a RMI client must execute a registry replica. This poses serious scalability problems, for example for the high costs of `bind()` operations. Instead, application developers can choose the appropriate replication degree for the Jgroup dependable registry, for example executing a registry instance only on hosts containing server objects.

But the main difference between Jgroup and Filterfresh is that Jgroup is based on a partitionable GCS, while Filterfresh is based on a primary-partition system. The primary-partition approach is not satisfactory for the design of high-available applications in partitionable environments, since primary-partition systems require the existence of a totally-connected majority of correct members. In a highly partitionable environment, this requirement may be rarely satisfied. The absence of a primary partition leads to the total blocking of the system; for example, Filterfresh does not guarantee that the process of constructing a new view will eventually terminate [5]. In Jgroup, the agreement protocol on new views will eventually terminate under any failure scenario [3]. More importantly, even when a primary partition exists, members not belonging to it cannot collaborate until communication with the primary partitions is restored, thus precluding continued availability in concurrent partitions. The Jgroup dependable registry is able to provide continued availability in every partition containing at least one operational registry instance.

7 Conclusions

CORBA [13] and Java RMI [23, 18] do not specify any abstraction for supporting the development of dependable applications in spite of process failures and network partitionings. Group communication, on the other hand, has proven to be an adequate paradigm for building reliable and high-available distributed systems. The integration of these complementary technologies in order to obtain a programming framework capable to support the design and the implementation of object-oriented, dependable distributed systems is an open research area [15, 14, 5, 11].

In this paper, we present the design and the implementation of Jgroup, an extension of the Java distributed object model based on the group communication paradigm. Differently from other group-oriented extensions of existing distributed object models, Jgroup is expressly aimed at supporting the development of reliable and high-available distributed applications in partitionable environments. Jgroup enables the creation of groups of remote objects that cooperate towards some common goal using a partitionable GCS. Remote object groups sim-

ulate the behavior of standard remote objects by implementing a set of remote interfaces and by enabling clients to remotely invoke the methods defined in these interfaces through the standard Java RMI mechanism.

Currently, we are working on the implementation of the reliable multicast invocation semantics defined in Section 4.2 and on the possibility of specifying which methods of a remote interface should be invoked with a reliable unicast invocation semantics (e.g., methods that could be efficiently carried out by a single object) or with a reliable multicast invocation semantics (e.g., methods that involve the entire group of replicas). We are also considering how to improve the performance of the low-level communication protocols of Jgroup through IP multicast. Future work will include the design and the implementation of state reconciliation primitives for supporting the reconstruction of a shared state after the disappearing of a partitioning [2].

References

[1] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable Group Membership Services for Novel Applications. In M. Mavronicolas, M. Merritt, and N .Shavit, editors, *Proc. of theDIMACS Workshop on Networks in Distributed Computing*, pages 23–42. American Mathematical Society, 1998.

[2] 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.

[3] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. Technical Report UBLCS-98-1, Dept. of Computer Science, University of Bologna, April 1998.

[4] Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Applications. In *Proc. of the 18th Int. Conference on Distributed Computing Systems*, pages 184–191, Amsterdam, May 1998.

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

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

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

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

[9] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the15th Symposium on Reliable Distributed Systems*, pages 150–159, Niagara-On-The-Lake, Canada, October 1996.

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

[11] James Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 2.2.* OMG Inc., Framingham, Mass., March 1998.

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

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

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

[16] S. Maffeis. iBus - The Java Intranet Software Bus. Technical report, Olsen and Associates, April 1997.

[17] 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.

[18] Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.42.* Sun Microsystems, Inc., Mountain View, California, October 1997.

[19] A. Montresor. A Dependable Registry Service for the Jgroup Distributed Object Model. Technical Report UBLCS-99-02, Dept. of Computer Science, University of Bologna, 1999.

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

[21] 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.

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

Biography

Alberto Montresor received the M.S. degree in Computer Science from the University of Bologna in 1995. He is currently a Ph.D. student at the University of Bologna. His current research interests include distributed computing, fault tolerance and distributed object frameworks.