Middleware for Dependable Network Services in Partitionable Distributed Systems*

Alberto Montresor ‡ Renzo Davoli ‡ Özalp Babaoğlu ‡

Abstract

We describe the design and implementation of *Jgroup*: a middleware system that integrates group technology with distributed objects and is based on Java RMI. Jgroup supports a programming paradigm called *object groups* and enables development of dependable network services based on replication. Among the novel features of Jgroup is a uniform object-oriented interface for programming both services and their clients. The fact that Jgroup exposes network effects, including partitions, to applications makes it particularly suitable for developing highly-available services in partitionable distributed systems.

1 Introduction

Distributed services are being called on to perform increasingly critical functions in our everyday lives such that errors or delays in their results may compromise human safety or result in economic loss. Thus, distributed services have to be *dependable*: they need to perform actions or furnish results that are *correct* and they need to remain *available* despite the inherent unreliability of their environment. Replication is the principal technique for rendering services dependable. By performing the same service at several replicas distributed over the system, we can guarantee both its correctness and availability as long as failures are independent and their number can be bounded.

Existing middleware platforms for distributed application development are ill suited for managing replication since they lack primitives for reliable "one-to-many" interactions between objects [14]. This, however, turns out to be the natural interaction style both for obtaining a (replicated) service on the part of clients, and for maintaining the consistency of the replicated service state on the part of servers. In its absence, existing middleware platforms require simulation of one-to-many interactions through multiple one-to-one interaction primitives, adding to the already complex and error-prone task of distributed service development. This shortcoming has been acknowledged by the OMG and is the subject of a recent "Request for Proposals" for fault-tolerant CORBA [14].

In this paper we present the Jgroup middleware system and argue why it is an effective basis for developing dependable distributed services. Jgroup integrates group technology and distributed objects based on Java RMI [20]. Group technology adopted by Jgroup follows the *object group* paradigm [11, 18] where functions of a distributed service are replicated among a collection of server objects that make up the logical object group for the service. Client objects interact with an object group implementing some distributed service through an *external group method invocation* (EGMI) facility. 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

*Partial support for this work was provided by Sun Microsystems, Inc. through a Collaborative Research Grant and the Italian Ministry of University, Research and Technology.

‡Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, Bologna 40127 (Italy), Email: {montresor,davoli,babaoglu}@CS.UniBO.IT

by Jgroup for both EGMI and IGMI in the presence of 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. While a general solution to the problem is highly application dependent and not always possible, Jgroup simplifies this task by providing support for stylized interactions (e.g., exchange of state information among different partitions) that occur frequently in solutions.

2 The Jgroup Distributed Object Model

The context of our work is a distributed system composed of client and server objects interconnected through a communication network. The system is *asynchronous* in the sense that neither the computational speeds of objects nor communication delays can be bounded. Furthermore, the system is unreliable and failures may cause objects and communication channels to *crash* whereby they simply stop functioning. Once failures are repaired, they may return to being *operational* after an appropriate recovery action. Finally, the system is *partitionable* in that certain communication failure scenarios may disrupt communication between multiple sets of objects forming *partitions*. Objects within a given partition can communicate among themselves, but cannot communicate with objects outside the partition. When communication between partitions is re-established, we say that they *merge*. The above properties are necessary for faithfully modeling practical distributed systems such as the Internet. Developing dependable applications to be deployed in these systems is a complex and error-prone task due to the uncertainty resulting from asynchrony and failures. The desire to render services partition-aware to increase their availability adds significantly to this difficulty. The Jgroup middleware system has been designed to simplify partition-aware application development by abstracting complex system events such as failures, recoveries, partitions, merges and asynchrony into simpler, high-level abstractions with well-defined semantics.

Jgroup promotes dependable application development through replication implemented as *object groups* [11, 18]. Distributed services that are to be made dependable are replicated among a collection of server objects that implement the same set of remote interfaces and form a group in order to coordinate their activities and appear to clients as a single server. Client objects access a distributed service by interacting

with the group identified through the name of the service. Jgroup handles all details such that clients need not be aware that the service is being provided by a group rather than a single server object. In particular, clients are unaware of the number, location or identity of individual servers in the group. Communication between clients and groups takes the form of *group method invocations*, that result in methods being executed by one or more servers forming the group, depending on the invocation semantics. For clients, group method invocations are indistinguishable from standard RMI interactions: clients obtain a representative object called a *stub* that acts as a proxy for a group, and perform invocations on it. Stubs handle all low-level details of group method invocations, such as locating the servers composing the group, establishing communication with them and returning the result to the invoker.

Jgroup extends the object group paradigm to partitionable systems through three major components: a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS).

2.1 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. Note that this may cause proposals for new view compositions to shrink during the installation phase in case agreement fails among the initial constituents. 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.

2.2 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

[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.

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 server, 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*. The next step in server object development is the creation of appropriate proxy classes for invoking external and internal group methods. Proxies for external group method invocations are called *stubs* as in standard Java RMI, while internal group method invocations are delegated to the group managers that handle all communication between server objects through group method invocations. Stubs implement the same external interface for which they act as a proxy, while group managers implement a *multi-response interface* derived from the internal remote interface. Multi-response interfaces differ from the corresponding internal remote interface due to the fact that internal invocations may return an array of results, rather than single values. Stubs, multi-response interfaces and group managers are all generated by the Jgroup tool `gmic` (group method invocation compiler), that extends the standard `rmic` compiler of Java RMI [20] to handle group method invocations.

In order to perform an internal group method invocation, servers must invoke the method on the appropriate group manager. Clients that need to interact with a group, on the other hand, must request a stub from a *dependable registry service* [21]. A dependable registry service allows servers to register themselves under a group name represented as a character string. Clients look up desired services by name in the registry and obtain their stub. The dependable registry service is an integral part of Jgroup and is implemented as a replicated service using Jgroup itself.

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 [3]. 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.

2.2.1 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. If the return type of the method being invoked is void, no return type 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 [7]. 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. For example, a distributed shared whiteboard would require that IGMI completions at participating servers be totally-ordered. 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 same view.

2.2.2 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 invocation semantics. 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, `gmic` analyzes the *throws* clause using reflection and produces the appropriate code.

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.

2.3 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 group membership service, 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 server state has to be constructed. This new state should reconcile, to the extent possible, any divergence that may have taken place during partitioned operation.

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 [2]. 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 [4, 21].

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: (i) 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; (ii) 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

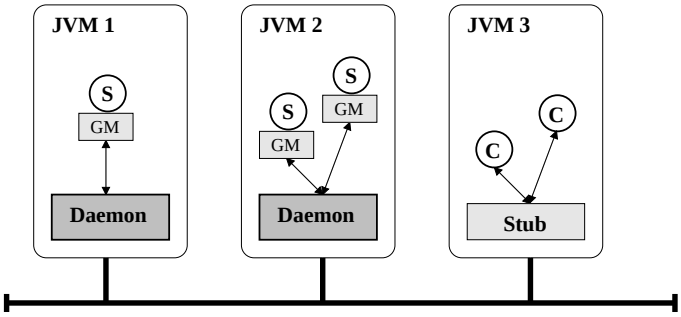

Figure 1: The overall Jgroup architecture. A service that is replicated three-fold is distributed over the first two Java Virtual Machines (JVM). Servers participate in the distributed computation through their respective group managers and the per-site Jgroup daemon. Clients request services and receive results through the stub object for the service.

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 complete specification of SMS is contained in another work [22]. Here we very briefly outline its basic properties. 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).

3 The Jgroup Implementation

In this section we give an overview of the Jgroup architecture and the principal algorithms used to implement it. Details can be found in another work [22]. The Jgroup architecture is illustrated in Figure 1. As discussed in the previous Section, object group facilities are provided to client and server objects through stubs and group managers, respectively. Stubs act as proxies for clients performing external group method invocations on the group, while group managers are used by servers to perform internal group method invocations. Group managers are also responsible for notifying servers of group events such as view changes and method invocations issued by other client or server objects. Each stub and group manager is associated with exactly one group; stubs may serve several clients concurrently, while each group manager is associated with exactly one server object (see Figure 1).

On the client side, stubs have total responsibility for handling external invocations. On the server side,

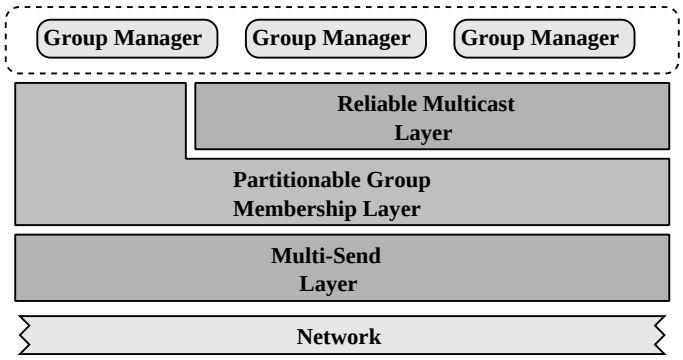

Figure 2: The architecture of a Jgroup Daemon

group managers implement only a subset of the object group services described in the previous section. Basic group membership and multicast communication facilities are implemented in a separate module called the *Jgroup daemon*. In each Java Virtual Machine (JVM) hosting Jgroup servers, a single instance of the Jgroup daemon is executed. There are several reasons for not putting the entire group service in the group managers but factoring out certain functions to the per-site Jgroup daemons. First, the number of messages exchanged to establish group communication is reduced; low-level services such as failure detection are implemented only once per-JVM and not replicated at every server object. Furthermore, this model enables the distinction between server objects local to a given JVM and those that are remote. Servers local to a given JVM share the same destiny with respect to failures: the crash of a JVM causes the crash of all servers hosted on it, and if a JVM becomes partitioned from the rest of the system, all its local servers stop communicating with remote servers. Thus, two distinct membership lists are maintained: one regarding local servers, and the other regarding remote daemons (and their associated servers). Voluntary variations in the membership due to join and leave operations are translated into single messages modifying the list of servers hosted at a daemon, without requiring complex agreement protocols.

Implementing the PGMS directly on top of a point-to-point unreliable, unsequenced datagram transport service provided by the network would be very difficult. For this reason, Jgroup daemons have a layered structure (Figure 2) with each layer providing higher-level abstractions as follows:

- *Multi-Send Layer (MSL):* The task of MSL is to hide the complexities of the underlying network by transforming the unreliable, point-to-point network communication primitives to their best-effort, one-to-many counterparts. Informally, MSL tries to deliver messages sent through it to all daemons in some destination list. Moreover, it keeps daemons informed about with which servers communication is possible and which are suspected to be partitioned or crashed.

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

- *Reliable Multicast Layer (RML):* The task of RML is to provide a message-based reliable multicast service. RML is integrated with the partitionable group membership layer and implements the message-based version of the view synchrony properties as discussed in Section 2.2.1.

Jgroup daemons are the basic building blocks of the Jgroup architecture; they provide the fundamental portions of the group communication service, such as failure detection, group membership and reliable message multicasts. All other facilities specified in the previous Section are provided directly by group managers. Like Jgroup daemons, group managers have a layered architecture. A group manager is composed specifically to satisfy the needs of a given server by including the necessary components (Figure 3). Group

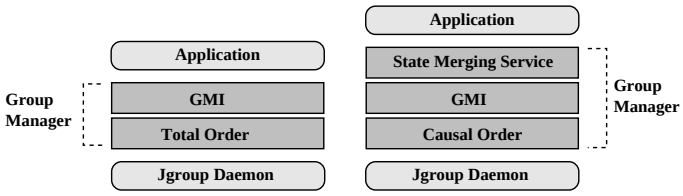

Figure 3: Two group managers with different configurations.

manager components that are currently available in Jgroup include group method invocation service, state merging service and several different ordering properties. Additional group manager components can be easily added to Jgroup in order to provide new facilities to developers.

We now describe in more detail each of the layers making up the Jgroup architecture. The Multi-Send Layer is a transport protocol based on UDP that also does message routing and failure detection [9]. Explicit routing at this layer is necessary to get around the lack of transitivity at the UDP layer. This phenomenon is not uncommon in large-scale distributed systems including the Internet [19]. The failure detector is necessary to guarantee the termination of the group membership protocol despite the asynchronous nature of the system, and is based on information obtained from the routing algorithm. Each change in the perceived communication state of the system (corresponding to server crashes, recoveries, partition, merges) is notified by MSL to the group membership service.

The group membership and reliable multicast layers included in Jgroup have been extensively discussed in a previous work [3]. When MSL notifies a variation in the communication state, a coordinator-based agreement protocol is initiated. Each daemon sends its estimate of the new group composition to a coordinator daemon chosen deterministically from within that estimate. When the coordinator observes an agreement on the group composition among all daemons included in an estimate, the estimate is translated to a view and is sent to all daemons interested in it. Each daemon delivers the view to the servers that are local to it and are included in the view itself. Termination of the algorithm is guaranteed by the fact that during the agreement phase, estimates sent by daemons are monotonically decreasing sets. In the limit, agreement will be guaranteed on singleton sets and each server will install a new view containing itself alone.

The GMIS implements IGMI and EGMI facilities on top of the daemon, in cooperation with remote stubs residing at client sites. Servers interact with group managers alone, while clients interact only with stubs. Stubs are obtained by interrogating the dependable registry service. Object groups are located by a stub through *group references* that contain a standard Java RMI remote reference for each server. Group references are only approximations to the group's actual membership as known by the registry at the time of the interrogation. This is done to avoid updating a potentially large number of stubs that may exist in the system every time a variation in a group's membership occurs. On the negative side, group references may become stale, particularly in systems with highly-dynamic, short-lived server groups. Consequently, when an external group method invocation fails because all servers known to the stub have since left the group, the stub contacts the registry again in order to obtain fresher information about the group membership.

In the case of external invocations with anycast semantics, stubs select one of the servers composing the group and try to transmit the invocation to the corresponding group manager through a standard RMI interaction. The contacted group manager dispatches the method at the server and sends back the return value to the stub. If the selected server cannot be reached (due to a crash or a partition), the RMI system throws a remote exception. In this case, the stub selects a new group manager and tries to contact it. This process continues until either a server completes the method and a return value is received, or the list of group managers is exhausted. In the latter case, GMIS throws a remote exception to the client, in order to notify that the requested service cannot be accessed.

At the client stub, external invocations with multicast semantics proceed just as those with anycast semantics. The group manager receiving the invocation multicasts it to all members in its current view. A single return value (usually the one returned by the group manager initially contacted) is returned to the

client stub. Note that a direct multicasting of the invocation to all servers cannot be used since the actual composition of a group may be different from that of the group reference maintained by a stub. In any case, an additional communication step among servers is necessary in order to transmit the invocation to servers not included in the group reference and to guarantee view synchrony.

Stubs are also responsible for the creation of invocation identifiers. Invocation identifiers are used by group managers to detect and discard duplicate executions of the same method on the same server, and may be used by application servers to identify invocations. Each invocation 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 at each invocation). Invocation identifiers are transmitted together with the method name and the invocation arguments.

Unlike client stubs, group managers are located at server sites and are continuously notified about view changes by Jgroup daemons. In this manner, internal group method invocations issued by servers may be multicast directly to the group managers in the current view of the invoker using the reliable multicast service of Jgroup. When a group manager delivers an invocation, it invokes the corresponding method on the server and sends back the return value to the invoker. Return values are collected by the invoking group manager, which returns them to the associated server as an array of replies.

The state merging service (SMS) is implemented on top of the Jgroup daemon and GMIS. Whenever a new view is installed, SMS verifies that each member included in it is up-to-date with respect to the others. If not, a new state reconciliation protocol is started. SMS drives each reconciliation protocol by performing the following operations. First, it elects a set of coordinators; then, SMS calls back to coordinators to obtain the application-dependent information needed to update the other members. Finally, this information is diffused through an internal group method invocation of the "merging" method, whose task is to merge data coming from other servers into the local state. Note that the coordinator election step is based on the local information about the up-to-date relation maintained by Jgroup daemons, and does not require any exchange of messages. Generally, the state reconciliation protocols driven by SMS are very simple: when two partitions merge, two coordinators are elected and each of them updates the servers contained in the other partition. SMS, however, can tolerate more complex scenarios. If, for example, a coordinator fails before being able to provide the necessary information, a new view is installed and a new state reconciliation protocol is started.

Jgroup is implemented entirely in Java. Unlike other systems, Jgroup does not rely on any additional facilities that may be written in languages other than Java [5]. This assures that Jgroup is fully portable to every architecture for which a JVM is available.

4 Related Work

The problem of integrating group communication [7] and distributed object technologies such as CORBA [15] and Java RMI [20] has been the subject of intense investigation [17, 25, 13, 6, 23, 5]. Most of the research effort has been dedicated to CORBA and resulting object group systems may be classified into three categories [13]. The *integration approach* consists of modifying and enhancing an object request broker (ORB) using existing group communication services. CORBA invocations are passed to the group communication service that multicasts them to replicated servers. This approach has been pursued by the Electra system [17]. In the *interception approach*, low-level messages containing CORBA invocations and responses are intercepted on client and server sides and mapped to group communication services. This approach does not require any modification to the ORB, but relies on OS-specific mechanisms for request interception. Eternal [25] is based on the interception approach. Finally, the *service approach* provides group communication as a separate CORBA service. The ORB is unaware of groups, and the service can be used in any CORBA-compliant implementation. The service approach has been adopted by object group systems such as OGS [13], Doors [10] and Newtop [23]. The service approach has also influenced many of the proposals responding to the OMG "Request of Proposals" on fault-tolerant CORBA [14].

Unlike CORBA, the specification of Java RMI enables programmers to implement their own remote

references, thus extending the standard behavior of Java RMI. We have exploited this feature by developing the concept of group reference, whose task is to manage interactions between clients and remote object groups. The resulting system provides transparent access to object groups and completely satisfies the specification of Java RMI. Other Java-based middleware systems include Filterfresh [6] and JavaGroups [5]. Filterfresh shares the same goals as Jgroup: integration of the object group paradigm with the Java distributed object model. Filterfresh is rather limited, as it provides neither external remote method invocations with multicast semantics nor internal remote method invocations among servers. JavaGroups is a message-based group communication toolkit written in Java providing reliable multicast communication. In its current version, JavaGroups uses group communication protocols provided by the Ensemble system [16]. The remote method invocation facility of JavaGroups is not transparent, as it is based on the exchange of objects that encode method invocation descriptions.

What distinguishes Jgroup from existing object group systems is its focus on supporting highly-available applications to be deployed in partitionable environments. Most of the existing object group systems [13, 10, 6] are based on the primary-partition approach and thus cannot be used to develop applications capable of continuing to provide services in multiple partitions. Very few object group systems abandon the primary-partition model [23, 25] but do not provide adequate support for partition-aware application development.

5 Conclusions

We have presented the design and implementation of Jgroup, an extension of the Java distributed object model based on the group communication paradigm. Unlike other group-oriented extensions to various existing distributed object models, Jgroup has the primary goal of supporting reliable and high-available application development in partitionable systems. This addresses an important requirement for modern applications that are to be deployed in networks where partitions can be frequent and long lasting. In designing Jgroup, we have taken great care in defining properties for group membership, remote group method invocation and state merging service so as to enable and simplify partition-aware application development.

Jgroup enables meeting of reliability and availability requirements of applications through replication: services are implemented by groups of remote objects that use Jgroup facilities to cooperate in order to maintain the consistency of their state in the absence of partitionings. When a partitioning occurs, the state of partitioned servers is allowed to diverge; later, when the partitioning disappears, the state merging protocol included in Jgroup enables remote objects to re-establish a consistent state. Remote object groups simulate the behavior of standard, non-replicated remote objects by implementing the same set of remote interfaces. Jgroup allows clients to access replicated services transparently using standard Java RMI mechanisms. Jgroup is the first object group system with a uniform communication model based entirely on remote group method invocations. This allows us to exploit the characteristics and advantages of object-orientation for the application as a whole. Systems that base server-to-server interactions on message passing require programmers to cope with two different paradigms in reasoning about the correctness of applications.

As explained in Section 3, the principal engineering issue in implementing Jgroup was the transparent integration of group communication in an existing client-server communication paradigm as Java RMI. Jgroup is the subject of a collaborative research project between Sun Microsystems and the University of Bologna and our experiences in building Jgroup will be valuable for the evolution of the Java RMI API.

Acknowledgments We are grateful to Bill Joy for his comments on an early draft of this work.

References

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

[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. *IEEE Transactions on Software Engineering*, 2000. To appear.

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

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

[6] A. Baratloo, P. E. Chung, Y. Huang and al. Filterfresh: Hot Replication of Java RMI Server Objects. In *Proc. of the 4th Conf. on Object-Oriented Technologies and Systems*, Santa Fe, New Mexico, April 1998.

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

[8] K.P. Birman and T.A. Joseph. Exploiting Virtual Synchrony in Distibuted Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, 1987.

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

[10] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih. Doors: Providing Fault-Tolerance for CORBA Applications. In *Proc. of the IFIP International Conference on Distributed System Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

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

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

[13] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA Objects. In S. Krakowiak and S. Shrivastava, editors, *Advanced Distributed Computing: From Algorithms to Systems*, Lecture Notes in Computer Science, chapter 11, pages 254–276. Springer-Verlag, 2000.

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

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

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

[17] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of the 1st Conf. on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995.

[18] S. Maffeis. The Object Group Design Pattern. In *Proc. of the 2nd Conf. on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996.

[19] 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.

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

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

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

[23] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and Implementation of a CORBA Fault-Tolerant Object Group Service. In *Proc. of the 2nd IFIP Int. Conf. on Distributed Applications and Interoperable Systems*, pages 361–374, Helsinki, Finland, June 1999.

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

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

[26] 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.