Group-Enhanced Remote Method Invocations

Alberto Montresor **Renzo Davoli** **Özalp Babaoğlu**

Technical Report UBLCS-99-05

April 1999

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

The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`. Questions and comments should be addressed to `tr-admin@cs.unibo.it`.

Recent Titles from the UBLCS Technical Report Series

96-16 *Fault Tolerance through View Synchrony in Partitionable Asynchronous Distributed Systems*, A. Montresor, December 1996.

96-17 *A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time*, M. Bernardo, R. Gorrieri, December 1996 (Revised January 1997).

97-1 *Partitionable Group Membership: Specification and Algorithms*, Ö. Babaoğlu, R. Davoli, A. Montresor, January 1997.

97-2 *A Truly Concurrent View of Linda Interprocess Communication*, N. Busi, R. Gorrieri, G. Zavattaro, February 1997.

97-3 *Knowledge-Level Speech Acts*, M. Gaspari, March 1997.

97-4 *An Algebra of Actors*, M. Gaspari, G. Zavattaro, May 1997.

97-5 *On the Turing Equivalence of Linda Coordination Primitives*, N. Busi, R. Gorrieri, G. Zavattaro, May 1997 (Revised October 1998).

97-6 *A Process Algebraic View of Linda Coordination Primitives*, N. Busi, R. Gorrieri, G. Zavattaro, May 1997.

97-7 *Validating a Software Architecture with respect to an Architectural Style*, P. Ciancarini, W. Penzo, July 1997.

97-8 *System Support for Partition-Aware Network Applications*, Ö. Babaoglu, R. Davoli, A. Montresor, R. Segala, October 1997.

97-9 *Generalized Semi-Markovian Process Algebra*, M. Bravetti, M. Bernardo, R. Gorrieri, October 1997.

98-1 *Group Communication in Partitionable Systems: Specification and Algorithms*, Ö. Babaoğlu, R. Davoli, A. Montresor, April 1998.

98-2 *A Catalog of Architectural Styles for Mobility*, P. Ciancarini, C. Mascolo, April 1998.

98-3 *Comparing Three Semantics for Linda-like Languages*, N. Busi, R. Gorrieri, G. Zavattaro, May 1998.

98-4 *Design and Experimental Evaluation of an Adaptive Playout Delay Control Mechanism for Packetized Audio for use over the Internet*, M. Roccetti, V. Ghini, P. Salomoni, M.E. Bonfigli, G. Pau, May 1998 (Revised November 1998).

98-5 *Analysis of MetaRing: a Real-Time Protocol for Metropolitan Area Network*, M. Conti, L. Donatiello, M. Furini, May 1998.

98-6 *GSMPA: A Core Calculus With Generally Distributed Durations*, M. Bravetti, M. Bernardo, R. Gorrieri, June 1998.

98-7 *A Communication Architecture for Critical Distributed Multimedia Applications: Design, Implementation, and Evaluation*, F. Panzieri, M. Roccetti, June 1998.

98-8 *Formal Specification of Performance Measures for Process Algebra Models of Concurrent Systems*, M. Bernardo, June 1998.

98-9 *Formal Performance Modeling and Evaluation of an Adaptive Mechanism for Packetized Audio over the Internet*, M. Bernardo, R. Gorrieri, M. Roccetti, June 1998.

98-10 *Value Passing in Stochastically Timed Process Algebras: A Symbolic Approach based on Lookahead*, M. Bernardo, June 1998.

98-11 *Structuring Sub-Populations in Parallel Genetic Algorithms for MPP*, R. Gaioni, R. Davoli, June 1998.

98-12 *The Jgroup Reliable Distributed Object Model*, A. Montresor, December 1998 (Revised March 1999).

99-1 *Deciding and Axiomatizing ST Bisimulation for a Process Algebra with Recursion and Action Refinement*, M. Bravetti, R. Gorrieri, February 1999.

99-2 *A Theory of Efficiency for Markovian Processes*, M. Bernardo, W.R. Cleaveland, February 1999.

99-3 *A Reliable Registry for the Jgroup Distributed Object Model*, A. Montresor, March 1999.

99-4 *Comparing the QoS of Internet Audio Mechanisms via Formal Methods*, A. Aldini, M. Bernardo, R. Gorrieri, M. Roccetti, March 1999.

Group-Enhanced Remote Method Invocations

Alberto Montresor **Renzo Davoli** **Özalp Babaoğlu**

Technical Report UBLCS-99-05

April 1999

Abstract

We present a specification for Jgroup, an extension to the Java distributed object model based on group communication. Jgroup is particularly suited for developing dependable network applications that are to be deployed in environments subject to voluntary or involuntary network partitionings. Jgroup adapts view synchrony semantics, typically defined for message-based group communication systems, to remote method invocations on objects. Unlike existing object group systems that mix two different paradigms (message passing within groups and remote method invocations outside), Jgroup presents a single uniform programming interface based entirely on remote method invocations. Furthermore, Jgroup extends view synchrony semantics that govern interactions within the group to external clients of the group without requiring them to become full-fledged members. In this manner, the higher costs associated with providing strong guarantees on method invocations are limited to (typically small number of) group members while guarantees for clients can be light-weight and thus incur smaller costs.

1 Introduction

Distributed object technology has proven to be a successful paradigm for dealing with increased complexity of modern network applications. Notable examples of distributed object frameworks include CORBA [9] and Java Remote Method Invocation (RMI) [14]. These middleware platforms enable client/server interactions among distributed objects: server objects encapsulate an internal state and make it accessible through a well-defined interface; client objects are allowed to access services provided by server objects by issuing remote method invocations on them. Low-level details of remote invocations are handled by local surrogate objects that present the same interface as their remote counterparts and act as proxies for them.

Most of the existing object models focus their attention on distribution, interoperability and reusability of software components. Currently, little or no support is provided for dependability. This constitutes a major drawback for many modern applications, for which reliability and high-availability are essential requirements. The *object group* paradigm has been proposed in an effort to fill this void [12]. In this paradigm, functions of a distributed service are replicated among a group of server objects. Clients interact transparently with object groups as if they were single, non-replicated entities. Server objects forming a group cooperate in order to provide a more dependable version of the service to their clients. Cooperation among server objects is achieved through a *group communication service* [6, 13, 17, 18, 1, 2], that enables the creation of dynamic object groups and supports communication among objects through reliable multicasts. Objects forming a group have access to *views* representing the current group membership that may vary due to accidental events such as failures, or due to voluntary requests by objects to join or leave the group.

Existing object group systems [11, 4, 8] necessitate use of different communication paradigms for programming service providers and service consumers. This is because communication within a group of server objects is based on (reliable) message exchanges, while communication between external clients and an object group is based on remote method invocations. Furthermore, existing object group systems limit group semantics to operations among the servers. Client interactions with the server object group remain external to the group service and thus no guarantees beyond a traditional remote *single* invocation are given. In cases where it is necessary, the burden of diffusing the single invocation to multiple server objects falls on the programmer. Another common characteristic of existing object group systems is that they are based on *primary-partition* group communication services [6, 13]. Such services are appropriate either for systems where the network partitionings are unlikely (such as a LAN) or for applications that can limit their availability to a single partition. These are serious restrictions for modern network applications to be deployed in environments subject to frequent voluntary or involuntary partitionings and that need to remain available not just in one, but in as many partitions as possible.

In this paper we describe Jgroup, a group-enhanced extension of Java RMI, that we have developed in an effort to address the above issues [16]. Jgroup is novel in several respects. First, it presents to the programmer a uniform interface based entirely on remote method invocations. In other words, all object interactions, whether they are internal to server groups or external to clients, occur through remote method invocations. This not only simplifies the task of programming dependable applications, it also presents an opportunity to give a specification for the system adapted from *view synchrony* semantics typically defined for message-based group communication systems [2]. In Jgroup, servers are full members of a group in the sense that method invocations among them satisfy all properties of view synchrony. Clients, on the other hand, can be considered *light-weight* members of a group in that method invocations by clients on server object groups satisfy a subset of view synchrony properties. Recognition of differing membership roles allows higher costs of full view synchrony to be limited to server objects that are typically far fewer in number than client objects. Finally, Jgroup differs from existing object group systems since it is based on a *partitionable* group communication service [2]. Unlike the primary-partition case, a partitionable group communication service admits multiple views of the same group to exist concurrently. Intuitively, each view corresponds to those group members that exist in the same network partition, and thus can communicate. A partitionable group communication ser-

vice is the basis for building *partition-aware* applications that continue to make progress and remain available in multiple concurrent partitions [3]. Exactly which services remain available in which partitions depend on application semantics and the degree of consistency that is desired.

The rest of the paper is organized as follows. Before presenting an overview of Jgroup, Section 2 recalls some background notions on group communication and on remote method invocations. Section 3 defines the system model in which Jgroup is cast. Section 4 describes some of the main features of Jgroup remote method invocations, while Sections 5 and 6 contain the Jgroup specification. Finally, in Section 8 we compare our work to similar projects and draw conclusions.

2 Background

The Jgroup distributed object model is an integration of two consolidated technologies for the development of distributed applications: *group communication* to support dependability requirements, and *remote method invocation* (or equivalently *object orientation*) to support distribution, reusability and interoperability requirements. In this section, we briefly introduce the fundamental notions behind these technologies and present the main features of Jgroup.

2.1 Group Communication

Group communication technology [5] enables construction of dependable applications through replication. Over the last few years, many experimental group communication systems (GCS) have been proposed [6, 13, 17, 18, 1, 2]. While they may differ in the exact set of services they provide, they share a common architecture: a *group membership service* (GMS) is integrated with a *reliable message multicast service*. The task of GMS is to keep members consistently informed about changes in the current group membership through the installation of *views*. Group membership may vary either due to voluntary requests to join or leave a group, or due to involuntary events such as failures and repairs of computing resources (process crashes and recoveries) or the communication system (network partitionings and mergings). Installed views are sets of members and correspond to the perception of the group's current 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. The task of a reliable multicast service is to enable communication among group members through multicast messages. Message deliveries are integrated with view installations as follows: two members that install the same pair of consecutive views deliver the same set of messages between their installations. This delivery semantics, called *view synchrony*, enables group members to reason globally about the state of other members using only local information based on current view composition and the set of delivered messages.

As noted in the introduction, two distinct classes of GCS have emerged: *primary-partition* [6, 13] and *partitionable* [17, 18, 1, 2]. A primary partition GCS attempts to maintain a single view of the current group membership. Members outside of this view are logically blocked even if they remain perfectly functional. In contrast, a partitionable GCS allows multiple views of the group to co-exist in the system, with each view corresponding to one of the partitions into which the network is divided. Members of each view can potentially continue computing independently from those in other views, subject to consistency constraints placed by the application semantics.

2.2 Remote Method Invocations

In the Java distributed object model [14], a *remote object* is one whose methods can be invoked from *client objects* running on different hosts. A remote object is described through a *remote interface* that is a collection of methods. *Remote method invocation* refers to the action of invoking one of the methods in the remote interface[1]. Clients of an object never interact directly with it, but only through a local surrogate object called a *stub*, that presents the same interface and acts as a proxy for the object. Objects are located by their stubs through *remote references*. Method

1. Given that we consider only remote versions of objects, interfaces and method invocations, we omit the term "remote" in the following.

invocations, together with their arguments, are marshaled by the stub and sent to the object. On the remote object side, a *skeleton* object unmarshals method invocations and dispatches them to the corresponding methods of the object. Once the execution of the invoked method terminates, the return value is marshaled by the skeleton and sent back to the stub, which returns it to the client.

2.3 The Jgroup Distributed Object Model

Current distributed object models such as Java RMI and Corba [14, 9] are not suitable for the development of dependable applications. On the server side, they lack systematic support for cooperation among multiple objects replicating a given service, thus forcing developers to implement their own mechanisms. On the client side, Corba and Java RMI provide only unicast method invocation semantics, thus supporting only simple client-server interactions. Partitioned or crashed servers cause the raising of exceptions, informing clients that the service they requested cannot be accessed. In order to access a replicated service, clients need to know each of the replicas and be able to select one that is both reachable and operational among them.

Jgroup solves these problems by extending the Java RMI model with group communication technology. In Jgroup, services are provided by server object groups and services are consumed by clients. Server objects within a group coordinate their activity so as to appear, whenever possible, as a single object. Coordination among servers is achieved through a GCS, while interactions between clients and servers are based on remote method invocations. Jgroup supports the *open group model* [10] where only server objects are effective members of the group while clients are external to the group. In other words, clients do not participate in group membership protocols and thus do not maintain views. This is essential for efficiency and scalability of the system since the number of server objects is typically small but there may be large numbers of short-lived clients.

Clients access services provided by an object group by invoking methods on the entire group through an *external reliable multi-invocation* (ERMI) service. From a client's point of view, object groups are indistinguishable from standard objects: clients obtain a stub for the group and perform remote method invocations through it as usual. The ERMI service attempts to invoke the method on each reachable server object in the group. In order to preserve transparency, remote method invocations performed by clients return a single result corresponding to one of the server invocations it provoked.

The main difference between existing object group systems and Jgroup is the fact that the remote method invocation paradigm is extended to communication among group members. Within a group, server objects invoke methods on the entire group through an *internal reliable multi-invocation* (IRMI) service. Unlike external invocations by clients, multiple results corresponding to multiple invocations are collected together and returned to the invoker object. View synchrony semantics introduced in Section 2.1 is extended to both internal and external invocations: two servers that install the same pair of views execute the same set of invocations in the period occurring between their installation. Invocation-based view synchrony enormously simplifies the development of distributed applications, as a server within a group can reason globally about the current state of other servers based only on its current view and on the set of invocations it performed.

The Jgroup system is implemented entirely in Java [16]. In order to become a member of a group and use the facilities provided by Jgroup, a Java object must implement the *Member* interface, whose methods are event handlers used by Jgroup to notify the object of group communication events such as view installations. The object also implements one or more remote interfaces, whose methods are specific to the offered services and can be invoked by clients. In each Java virtual machine, group communication semantics is provided by a *Jgroup daemon* that communicates with other Jgroup daemons present in the distributed system and establishes the basic membership and communication protocols on behalf of member objects.

The Jgroup architecture is completed by the *dependable registry*, an object repository service used by object groups to advertise their availability to provide certain services, and by clients to retrieve stubs for these groups [15]. Stubs for object groups returned by the dependable registry

are standard Java stubs in the sense that they are generated by the Java RMI compiler [14]. The dependable registry is a fundamental building block of Jgroup; at the same time, it constitutes a realistic dependable application written using Jgroup itself (see Section 7 for details).

Jgroup includes services beyond those described in this work. For lack of space, we mention them only briefly. The internal and external reliable multi-invocation services can be augmented to provide FIFO, causal and atomic ordering among invocations. This additional knowledge about the order in which invocations are completed by servers may further simplify development of certain applications.

3 System Model

Informally, the distributed system model for which Jgroup has been targeted can be characterized as asynchronous and partitionable. More formally, $\mathcal{O}$ denotes the finite set of *objects* that may communicate through a network. Objects are identified through unique names that maintain throughout their life. The system is asynchronous in the sense that computation and communication delays cannot be bounded. Practical distributed systems often have to be considered as being asynchronous due to transient failures, unknown scheduling strategies and variable loads on the computing and communication resources. To simplify the presentation, however, we make reference to a discrete global clock whose ticks coincide with the natural numbers in some unbounded range $\mathcal{T}$. This simplification is not in conflict with the asynchrony assumption since objects cannot access this clock.

The execution of a distributed algorithm results in each object performing a sequence of *events* chosen from a set $\mathcal{E}$. In addition to internal events not relevant for this work, $\mathcal{E}$ includes method invocation and group communication events. The *global history* of an execution is a function σ from $\mathcal{O} \times \mathcal{T}$ to $\mathcal{E}$. If object α executes an event $e \in \mathcal{E}$ at time t, then $\sigma(\alpha, t) = e$. Given some interval $\mathcal{I}$ of $\mathcal{T}$, we write $e \in \sigma(\alpha, \mathcal{I})$ if α executes event e sometime during interval $\mathcal{I}$ of global history σ (i.e, $\exists t \in \mathcal{I} : \sigma(\alpha, t) = e$).

Objects may fail by *crashing* whereby they stop executing events. For simplicity, we do not consider object recovery after a crash. The evolution of object failures during an execution is captured through the *crash pattern* function C from $\mathcal{T}$ to $2^{\mathcal{O}}$ where $C(t)$ denotes the set of objects that have crashed by time t. Since crashed objects do not recover, we have $C(t) \subseteq C(t+1)$. With $\mathit{Correct}(C) = \{\alpha \mid \forall t : \alpha \notin C(t)\}$ we denote those objects that never crash, and thus, are correct in C.

The communication system is based on typical unreliable datagram transport services such as UDP. In the absence of failures, the network is connected and each object can communicate with every other object. Communication failures may disable communication between objects. Unlike object crashes, communication failures may be temporary due to subsequent repairs. The evolution of communication failures and repairs during an execution is captured through the *unreachability pattern* function U from $\mathcal{O} \times \mathcal{T}$ to $2^{\mathcal{O}}$ where $U(\alpha, t)$ denotes the set of objects with which α cannot communicate at time t. If $\beta \in U(\alpha, t)$, we say that object β is *unreachable* from α at time t, and write $\alpha \not\rightsquigarrow_t \beta$ as a shorthand; otherwise we say that object β is *reachable* from α at time t, and write $\alpha \rightsquigarrow_t \beta$. The dynamic nature of communication failures is reflected by the fact that $U(\alpha, t)$ and $U(\alpha, t+1)$ may differ arbitrarily.

4 Remote Method Invocation Model

Each object α is characterized by an interface $int(\alpha)$, consisting of the collection of methods that can be invoked on α. Every method is uniquely identified through a name m. For sake of simplicity, in this work we do not consider the problems related to formal parameters and actual arguments for method invocations, and focus our attention on the issues related to fault-tolerance and distribution.

A server object α *performs* an internal invocation of method m on the object group to which α belongs if α executes event $invoke(m, k)$. The *invocation identifier* k serves to distinguish multiple

invocations of the same method by the same object. A client object *performs* an external invocation of method m on object group g if it executes event $invoke(g, m, k)$. Note that external invocations need to mention the object group explicitly while it is implicit for internal invocations.

Invocation k of method m returns value r through the execution of event $return(m, k, r)$ by the invoker. How the return value is related to the values returned by servers executing the methods is specified in Sections 5 and 6. Here, we simply assume that before executing event $return(m, k, r)$, an object must have performed the corresponding invocation, and that each object executes event $return$ for a given invocation at most once. Formally,

$$(i) \quad \sigma(\alpha, t) = return(m, k, r) \Rightarrow invoke(m, k) \in \sigma(\alpha, [0, t[) \vee invoke(g, m, k) \in \sigma(\alpha, [0, t[)$$
$$(ii) \quad \sigma(\alpha, t) = return(m, k, r) \Rightarrow return(m', k, r') \notin \sigma(\alpha, \mathcal{T} - \{t\})$$

Invocations cause the execution of the invoked method at certain number of objects, depending on the failure scenario. Execution of method m at object α corresponding to invocation k results in a sequence of events determined by the internal state of α and by the algorithm associated with m. This sequence starts with $begin(m, k)$ and, in the absence of failures, terminates with $end(m, k, r)$, where r is the return value chosen from a set $\mathcal{R}$. Obviously, we assume that before executing event $end(m, k, r)$, object α must have executed event $begin(m, k)$, and each object executes event end for a given invocation at most once. Formally,

$$(i) \quad \forall \alpha \in \textit{Correct}(C) : \sigma(\alpha, t) = begin(m, k) \Rightarrow \exists r : end(m, k, r) \in \sigma(\alpha,]t, \infty[)$$
$$(ii) \quad \sigma(\alpha, t) = end(m, k, r) \Rightarrow begin(m, k) \in \sigma(\alpha, [0, t[) \wedge end(m', k, r') \notin \sigma(\alpha, \mathcal{T} - \{t\})$$

We say that object α *completes* invocation k of method m at time t if α executes event $end(m, k, r)$ at time t.

5 The Partitionable Group Communication Service

This section specifies the partitionable GCS included in Jgroup. This specification is based on previous work on partitionable group communication [17, 18, 1], and in particular extends the specification of Relacs [2]. Nevertheless, it differs from classical group communication since it is based on remote method invocations rather than on multicast messages. For the sake of brevity, we refer the reader to previous papers for a thorough discussion of the motivations behind each of the properties included in the specification [2, 3].

5.1 The Group Membership Service

Servers with the same interface can be collected together in order to form a *group*. Every group is uniquely identified by a group name g and is associated the interface $int(g)$ common to its members denoting the set of methods that can be invoked on g. Membership of a group is dynamic: a server α may join a group g, provided that $int(\alpha) = int(g)$; later on, a server may decide to leave a group it has joined. A server α joining or leaving a group g results in events $join(g)$ and $leave(g)$, respectively. The membership of g at time t is denoted by the set $memb(g, t)$ and comprises those servers that have joined the group but that have not crashed or left the group by time t. Formally,

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

For sake of brevity, in this work we assume that a server can be a member of at most one group at a time; this simplifies the specification, as interactions among views installed for different groups need not to be considered.

The *partitionable* GMS included in Jgroup keeps servers in a group g informed about the group's current membership through the installation of *views*. The installation of a view v for group g at server α corresponds to the execution of event $vchg(g, v)$. Due to asynchrony and the

possibility of partitions, a view of group g can only be an approximation of the current membership. We assume that view installations cannot interrupt execution of a method: given an event $begin(m,k)$ executed by α at time t_1 and an event $end(m,k,r)$ executed by α at time t_2, no $vchg(g,v)$ events can be executed by α between t_1 and t_2. Without this assumption, it would be difficult to program methods whose actions were dependent on the composition of the current view.

Views are labeled in order to be globally unique. Given a view v, we write $\overline{v}$ to denote its composition as a set of process names. The *current view* of process α at time t is v, denoted $\mathit{view}(\alpha,t) = v$, if v is the last view to have been installed at α before time t. Events are said to occur *in the view* that is current. In the same way, invocations are said to be completed *in the view* that is current, given the fact that method executions cannot be interrupted by view installations. View w is called *immediate successor of* v *at* α, denoted $v \prec_\alpha w$, if α installs w in view v. View w is called *immediate successor* of v, denoted $v \prec w$, if there exists some process α such that $v \prec_\alpha w$. The *successor* relation $\prec^*$ denotes the transitive closure of $\prec$. Two views that are not related through $\prec^*$ are called *concurrent*.

The GMS specification has to take into account several issues. The service must track server crashes, network partitionings and mergings, as well as changes in membership due to $join$ and $leave$ operations. We also require that servers install only views to which they belong, and only after having agreed upon their composition with other servers in the view. Two views installed by two different servers must be installed in the same order. Being partitionable, our GCS admits the co-existence of concurrent views. Unlike a primary-partition GCS, our system is *live*: it never stops to track changes in group membership, even under failures scenarios where no primary partition exists. Without lack of generality, the specification below is stated for a single group. The GMS specification is summarized through the following properties:

GM1 (View Accuracy) *If there is a time after which two correct servers are permanently and mutually reachable, then eventually all views installed by one will contain the other. Formally,*

$$\exists t_0, \forall t \geq t_0 : \alpha \rightsquigarrow_t \beta \wedge \alpha, \beta \in \mathit{Correct}(C) \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : \beta \in \overline{\mathit{view}(\alpha,t)}.$$

GM2 (View Completeness) *If there is a time after which two correct servers are permanently and mutually unreachable, then eventually all views installed by one will exclude the other. Formally,*

$$\exists t_0, \forall t \geq t_0, \forall \beta \in \Theta, \forall \alpha \notin \Theta : \alpha \not\rightsquigarrow_t \beta \;\Rightarrow\; \exists t_1, \forall t \geq t_1, \forall \gamma \in \mathit{Correct}(C) - \Theta : \overline{\mathit{view}(\gamma,t)} \cap \Theta = \emptyset.$$

GM3 (Group Consistency) *(i) If a server permanently leaves the group, then eventually it will be excluded from all views for the group installed by other servers. (ii) A server that has never joined the group cannot be included in any view installed for the group. Formally,*

$$(i) \quad \exists t_0, \forall t \geq t_0 : \alpha \notin memb(g,t) \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : \beta \notin memb(g,t) \vee \alpha \notin \mathit{view}(\beta,t);$$
$$(ii) \quad \sigma(\alpha,t_0) = vchg(g,v) \;\Rightarrow\; \exists t < t_0 : \alpha \in memb(g,t).$$

GM4 (View Agreement) *(i) A group member is eventually notified with a view installation if one of the servers included in its current view v never installs v or installs some other view as an immediate successor to v. (ii) If server α installs view v and its immediate successor w, both containing β, then α installs w only after β has installed v. Formally,*

$$(i) \quad \alpha \in \mathit{Correct}(C) \wedge \mathit{view}(\alpha,t_0) = v \wedge (\exists \beta \in \overline{v}, \exists t_1, \forall t > t_1 : \mathit{view}(\beta,t) \neq v) \;\Rightarrow\; \exists w : v \prec_\alpha w;$$
$$(ii) \quad \sigma(\alpha,t_0) = vchg(g,v) \wedge v \prec_\alpha w \wedge \beta \in \overline{v} \;\Rightarrow\; vchg(g,v) \in \sigma(\beta,[0,t_0[).$$

GM5 (Merging Rule) *Two views merging into a common view must have disjoint compositions. Formally,*

$$(v \prec u) \wedge (w \prec u) \wedge (v \neq w) \;\Rightarrow\; \overline{v} \cap \overline{w} = \emptyset.$$

GM6 (View Order) *If two servers install the same two views, then they install them in the same order. Formally,*

$$v \prec^* w \;\Rightarrow\; w \not\prec^* v.$$

GM7 (View Integrity) *A view v cannot be installed by servers not belonging to v. Formally,*

$$\sigma(\alpha, t) = vchg(g, v) \;\Rightarrow\; \alpha \in \overline{v}.$$

5.2 The Internal Reliable Multi-Invocation Service

Properties of GMS included in Jgroup are similar to those contained in other partitionable GMS such as Horus [18] and Transis [1]. The communication service of our specification, however, presents some novelties: communication among servers is performed by invoking methods on the entire group instead of multicasting messages. This change is motivated by the desire to unify the communication model within server groups, and between clients and server groups. The resulting specification is simple, as view synchrony applies to invocations performed by both clients and servers.

After having joined group g, server α may perform an internal invocation of method m on group g. The IRMI service attempts to force the execution of method m at each server object belonging to the current view, including the invoker itself, which will eventually complete the invocation (unless it crashes). In the following, we use M_α^v to denote the set of method invocations that have been completed by object α in view v. Formally,

$$M_\alpha^v = \{k \mid \sigma(\alpha, t) = end(m, k, r) \wedge \mathit{view}(\alpha, t) = v\}$$

If the invoker is correct, internal invocation k of method m terminates with event $return(m, k, R)$, where R is a *reply set* containing a collection of pairs (α, r) denoting that server α has returned value r for the invocation through the execution of an event $end(m, k, r)$. If the current view remains unchanged, the invoker server is guaranteed that the reply set contains a value for each of the servers contained in the view. Otherwise, when installing new view w, a server is guaranteed that the reply set obtained through $return$ event executed in the previous view v contains a return value for each of the servers surviving from v to w (i.e., those servers belonging to the intersection of v and w). In other words, a server performing an internal invocation will obtain a reply set containing a return value from each reachable server completing the invocation, and will install a new view if some of the servers in its current view are unable to return a value due to failures. Finally, we guarantee that internal invocations are completed in at most one view, thus avoiding situations where an invoker obtains return values from servers belonging to different views and maintaining potentially inconsistent states. In the following, we use M_α^v to denote the set of method invocations that have been completed by object α in view v. These requirements are summarized through the following properties:

IRMI1 (Self-Completion) *A correct server will eventually complete internal invocations performed by itself. Formally,*

$$\sigma(\alpha, t) = invoke(m, k) \wedge \alpha \in \mathit{Correct}(C) \;\Rightarrow\; end(m, k) \in \sigma(\alpha,]t, \infty[).$$

IRMI2 (Termination) *A correct server performing an internal invocation will eventually obtain a reply set for that invocation. Formally,*

$$\sigma(\alpha, t) = invoke(m, k) \wedge \alpha \in \mathit{Correct}(C) \;\Rightarrow\; \exists R : return(m, k, R) \in \sigma(\alpha,]t, \infty[) \wedge R \subseteq \mathcal{O} \times \mathcal{R}$$

IRMI3 (Non-Triviality) *If a correct server α obtains a reply set R for an internal invocation performed in view v, then R contains the pair (β, r) for each server β included in v, or α will eventually install a new view after v. Formally,*

$$\sigma(\alpha, t) = return(m, k, R) \wedge \alpha \in \mathit{Correct}(C) \wedge \mathit{view}(\alpha, t) = v \wedge \beta \in \overline{v} \Rightarrow$$
$$(\exists r : (\beta, r) \in R) \vee (\exists w : v \prec_\alpha w).$$

IRMI4 (Validity) *(i) If the pair (α, r) is included in the reply set R obtained by a server β, then server α must have previously returned r through an event $end(m, k, r)$. (ii) Given two views v, w such that w is an immediate successor of v, all reply sets R obtained through event $return(m, k, R)$ executed in view v are such that there is a pair (α, r) for each server α that survives from v to w. Formally,*

$$(i) \quad \sigma(\beta, t) = return(m, k, R) \wedge (\alpha, r) \in R \Rightarrow \ end(m, k, r) \in \sigma(\alpha, [0, t[).$$
$$(ii) \quad v \prec_\alpha w \wedge \beta \in \overline{v} \cap \overline{w} \wedge \sigma(\alpha, t) = return(m, k, R) \wedge \mathit{view}(\alpha, t) = v \Rightarrow \exists r : (\beta, r) \in R$$

IRMI5 (Uniqueness) *An internal invocation is completed in at most one view. Formally,*

$$(\sigma(\alpha, t) = invoke(m, k)) \wedge k \in M^v_\beta) \wedge (k \in M^w_\gamma) \Rightarrow v = w.$$

These properties describe how internal invocations are related to invocation completions. As stated earlier, internal invocations cannot be guaranteed to complete at each server included in the current view of the invoker. Nevertheless, IRMI guarantees reliable communication properties analogous to view synchrony. First of all, a correct server completing an invocation is guaranteed that each server in the current view will complete the same invocation, unless it crashes or becomes partitioned. Servers completing an invocation are eventually notified of servers not completing it through the installation of a new view. Furthermore, two servers surviving from a view to its immediate successor complete the same set of invocations in the first view. These requirements are summarized by the following properties:

IRMI6 (Invocation Liveness) *If a correct server α completes an invocation in a view v, then each server included in v will eventually complete the same invocation, or α will eventually install a new view after v. Formally,*

$$\sigma(\alpha, t) = end(m, k, r) \wedge \alpha \in Correct(C) \wedge \mathit{view}(\alpha, t) = v \wedge \beta \in \overline{v} \Rightarrow$$
$$(\exists r' : end(m, k, r) \in \sigma(\beta, \mathcal{T})) \vee (\exists w : v \prec_\alpha w)$$

IRMI7 (Invocation Agreement) *Given two views v, w such that w is an immediate successor of v, all servers belonging to both views complete the same set of invocations in view v. Formally,*

$$v \prec_\alpha w \wedge \beta \in \overline{v} \cap \overline{w} \Rightarrow M^v_\alpha = M^v_\beta.$$

These properties, corresponding to liveness and safety of view synchrony, are extremely useful when developing distributed applications, since servers may reason about the state of other servers in the group using their local knowledge about installed views and completed invocations. Note that the above properties are stated in terms of "invocations" and not "internal invocations". In fact, these properties hold for the ERMI service described in the next section as well. The same remark applies to the final property of our specification, which places a simple integrity condition for the IRMI service.

IRMI8 (Invocation Integrity) *Each server completes a invocation at most once and only if some object (client or server) actually performed it earlier. Formally,*

$$\sigma(\alpha,t) = begin(m,k) \;\Rightarrow\; (begin(m,k) \notin \sigma(\alpha, \mathcal{T} - \{t\})) \wedge$$
$$(\exists \beta : invoke(m,k) \in \sigma(\beta, [0,t[) \vee invoke(g,m,k) \in \sigma(\beta, [0,t[)).$$

The multi-invocation / multi-reply invocation service expressed by Properties IRMI1–IRMI8 has many advantages over multicast message-based group communication. The development of complex partition-aware consistency protocols is simplified since marshaling and unmarshaling of messages is handled automatically by the stubs when passing arguments. As each internal invocation involves a request / reply protocol among multiple servers, programming protocols such as state transfer for obtaining the current state of the computation by a new server are greatly simplified. Finally, the unified communication model allows a more compact specification with respect to object group systems that mix message multicast and remote method invocation.

6 The External Reliable Multi-Invocation Service

Clients access services offered by an object group through external reliable multi-invocations of group methods. In this section we give a specification for the ERMI service used by clients and how it is integrated with the IRMI service used by servers.

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

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

ERMI1 (Termination) *External invocations performed by correct clients always terminate. Formally,*

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

ERMI2 (Non-Triviality) *If there is a time after which server β remains permanently reachable from a correct client α and β is permanently a member of group g, then eventually α obtains return values different from $\perp$ for each external invocation it performs on g. Formally,*

$$\exists t_0, \forall t \geq t_0 : \alpha \rightsquigarrow_t \beta \wedge \beta \in memb(g,t) \;\Rightarrow\; \exists t_1, \forall t > t_1 : (\sigma(\alpha,t) = return(m,k,r) \;\Rightarrow\; r \neq \perp).$$

ERMI3 (Validity) *A return value r different from $\perp$ obtained through an event $return(m,k,r)$ for an external invocation on group g must have been previously returned by a server in g through an event $end(m,k,r)$. Formally,*

$$\sigma(\alpha,t_0) = return(m,k,r) \wedge invoke(g,m,k) \in \sigma(\alpha,[0,t_0[) \Rightarrow$$
$$\exists\beta, \exists t_1 < t_0 : end(m,k,r) \in \sigma(\beta,t_1) \wedge \beta \in memb(g,t_1).$$

As was noted in the previous section, Properties IRMI6–IRMI8 need to be added to the above in order to complete the specification of the ERMI service in Jgroup. The fact that Properties IRMI6–IRMI8 hold for both internal and external invocations allows the global reasoning possible for server objects to extend to clients as well: two servers that install the same pair of consecutive views complete the same set of invocations (internal and external) in the period occurring between the installations of these views.

7 Active Replication with Jgroup

In order to conclude the description of Jgroup, we discuss how it can be used to implement distributed applications based on *active replication* in partitionable systems. In the active replication paradigm, the replicated state is updated by each server of the group in response to every operation, and servers respond to identical invocations with identical results. In a partitionable system, we relax these requirements by allowing states of servers in different partitions to diverge until the partitions merge. Consequently, two servers may respond differently to external invocations performed by clients during periods while partitioned.

The *dependable registry* (or *registry* for brevity) included in Jgroup[15] may be considered an example of this programming style. A registry is a repository facility used by long-lived server groups to advertise their availability to provide certain services, and by clients to retrieve stubs for these groups. Among others, the interface of the registry contains two fundamental methods: $lookup(id)$ used to build and return a stub for a group identified by id, and $bind(id,\alpha)$ used by server α to inform the registry that it has joined group id. The registry is implemented through a group of registry servers that actively maintain a replicated database of bindings between servers and group names. Clients access services offered by server groups transparently through the stubs that they obtain from the registry. Stubs contain descriptions of the group membership as known to the registry at the time of the $lookup$ request and handle the actual interactions with the server objects. The registry itself is just another service available to clients with the only exception that the stub for it is obtained through a bootstrap mechanism. Group stubs are only approximations for the group's membership at any time. This is to avoid the high cost of updating a potentially large number of stubs that may exist in the system when servers join or leave the group. Obviously, this choice may cause stubs to become stale, particularly in the case of highly-dynamic, short-lived server groups. For this reason, when an external invocation on a group fails because all servers known to the stub have since left the group, the stub contacts the registry in order to be updated on the membership of the group.

The algorithm associated with the $lookup(id)$ method simply builds and returns a stub for the group identified by id, based on the set of servers registered under the name id in the local database. Invocations of the $bind(id,\alpha)$ method are completed by simply adding α to the set of servers associated with id in the local database. The registry also allows voluntary removals of servers from groups. Involuntary removal of servers from groups due to crashes is achieved through *leases*: a server that does not renew its binding periodically is removed from the database.

View synchrony guarantees that registry servers surviving from one view to the next will have performed the same set of $bind$ invocations between the installations of these views. This implies that registry servers that maintain the same database of bindings at the beginning of the first view, maintain the same database of bindings at the beginning of the next view. On the other hand, registry servers belonging to different partitions may maintain different databases. When

these partitions merge, a reconciliation protocol is needed. This reconciliation protocol works as follows: for each merging view, one of the registry servers assumes the role of coordinator for the view and invokes a method $newstate(B)$, where B is the set of $bind$ invocations that may be unknown to registry servers in other views. The IRMI service guarantees that each of the registry servers participating in the reconciliation protocol will complete the method and update its database, or a new view is installed (possibly starting a new reconciliation protocol, if the previous one has not been completed).

In the presence of partitionings, the registry presents a partitioned behavior reflecting the failure scenario. $bind$ methods executed inside a partition will not affect registry servers not contained in that partition, while $lookup$ methods will not be able to retrieve groups that have been registered in other partitions. Nevertheless, registry servers in a partition eventually performs the same updates to the database and act as a single entity; moreover, consistency among diverged replicas is restored by the reconciliation protocol when partitions merge. This behavior is perfectly suitable for a partitionable distributed system, since clients are interested only in those server groups that are running in their partitions and thus are able to serve their requests.

8 Related Work and Conclusions

In the last few years, the problem of integrating group communication with distributed object technologies such as CORBA [9] and Java RMI [14] has been the subject of intense investigation [7, 11, 4]. Electra and Orbix+Isis [11] are two CORBA object request brokers (ORBs) whose implementation is based on group communication. Both toolkits allow programmers to treat a group of objects as if they were a single entity, and clients invoke methods on object groups. Internal communication among members is based on multicast messages satisfying the virtual synchrony semantics; remote method invocations are translated in multicast messages and delivered to members. Virtual synchrony guarantees that all significant events such as multicast messages and view installations are delivered to members in a consistent order.

Electra and Orbix+Isis are based on the *integration* approach, which consists in modifying an ORB using group communication. The integration approach is appealing for its transparency, since clients access object groups as if they were single objects; unfortunately, the resulting ORBs are not CORBA-compliant. An alternative methodology is the *service* approach used in the Object Group Service (OGS) [8], which consists in providing group communication as a service on top of an ORB. Clients hold references to OGS services, whose task is to provide primitives to communicate with a groups of objects. The resulting system is not transparent, but is completely CORBA-compliant.

As Jgroup, Filterfresh [4] extends Java RMI with group communication. Filterfresh, however, provides only a *unicast* RMI semantics: clients hold stubs for objects groups; method invocation on these stubs are forwarded to only one of the member of the group. In other word, view synchrony is not extended to method invocations; to circumvent this limitation, an object completing a method invocation should multicast it to the other group members. Note that Jgroup [16] provides this unicast RMI semantics as well, in order to support method invocations whose execution does not need to be performed by the members of a group.

Karamanolis et al. [10] discuss at length the issues related to client-access protocols for replicated services. Their work describes how to integrate group communication with remote procedure call. Several protocols are considered, ranging from the *closed* group model, in which clients must be members of a group in order to be continuously notified about the membership changes, to different classes of *open* group systems, in which clients are kept external to the group and procedure calls are forwarded to members following the virtual synchrony semantics.

Despite their differences, all systems discussed so far share a common problem: they are not satisfactory for the design of high-available applications in partitionable environments, since they are based on primary-partition GCSes. The primary partition approach requires the existence of a totally-connected majority of correct members, and this requirement may be rarely satisfied in a highly partitionable environment. The absence of a primary partition may lead to the total

blocking of a GCS and of the applications based on it. More importantly, even when a primary partition exists, members not belonging to it cannot collaborate until communication with the primary partition is restored, thus precluding continued availability in concurrent partitions. Another problem common to the papers cited above is the presence of different communication models for clients and servers, which complicate the development of applications.

Jgroup solves these problems by providing the first group-enhanced distributed object toolkit based on a partitionable GCS. This feature makes Jgroup the right support for the development of high-available applications in partitionable distributed systems. Currently, the Jgroup prototype is being used for the development of demonstrative applications such as partition-aware automatic teller machines and high-available tuple spaces. Further extension to Jgroup are under development; among them, we are implementing a reconciliation protocol that supports reconstruction of a shared state after merging of partitions.

References

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

[2] 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.

[3] 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, May 1998.

[4] A. Baratloo, P. Emerald Chung, Y. Huang, S. Rangarajan, and S. Yajnik. 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.

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

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

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

[8] 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.

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

[10] C. Karamanolis and J. Magee. Client-Access Protocols for Replicated Services. *IEEE Transactions on Software Engineering*, 25(1), January 1999.

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

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

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

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

[15] 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, April 1999.

[16] A. Montresor. The Jgroup Reliable Distributed Object Model. In *Proc. of the 2nd IFIP Int. Working Conf. on Distributed Applications and Systems*, Helsinki, Finland, June 1999.

[17] 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.

[18] 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.