Enhancing Jini with Group Communication

Alberto Montresor Renzo Davoli Özalp Babaoğlu

Technical Report UBLCS-2000-16

December 2000
(Revised January 2001)

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

Enhancing Jini with Group Communication

Alberto Montresor[1] **Renzo Davoli**[1] **Özalp Babaoğlu**[1]

Technical Report UBLCS-2000-16

December 2000
(Revised January 2001)

Abstract

Reliable group communication has proven to be an important technology for building fault-tolerant applications. Yet, many frameworks for distributed application development (e.g., Dcom, Jini and Enterprise JavaBeans) do not support it. The only notable exception to this situation is Corba that has been recently extended to include a replication service. We claim that lack of group communication support in other development frameworks constitutes a major obstacle for their wider diffusion among industry. In this paper we discuss issues related to integrating reliable group communication and Jini technologies.

1. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, Bologna 40127 (Italy). Tel: +39 051 2094871, Fax: +39 051 2094510. Email: {montresor,davoli,babaoglu}@CS.UniBO.IT

1 Introduction

Reliable group communication is an important technique for building fault-tolerant applications based on replication [12, 7, 23, 13, 1]. The abstractions and primitives provided by group communication services make it easier to achieve consistency of replicated state information despite failures.

Despite these advantages, industrial or commercial applications based on group communication are far and few. In our opinion, the reason for this lack of acceptance on the part of industry is that, until recently, important frameworks for distributed application development including OMG's Corba [11], Microsoft's DCOM [4], Sun's Jini [2] and Enterprise JavaBeans [18] did not provide any support for reliable group communication.

In all of the named frameworks, support for fault-tolerance has been limited to the *transactional paradigm*, in which operations performed on one or more objects are grouped together so as to guarantee certain properties on their execution. An example of such a property is *failure atomicity*: either all operations of the transaction are performed on all objects, or no operation is performed.

In the case of Corba, however, the situation is about to change. In 1998, the OMG recognized the need for a replication mechanism capable of achieving more stringent fault tolerance requirements than that possible through the *object transaction service* (OTS) [9] and issued a *request for proposals* for fault tolerance to be included in Corba 3. The OMG standardization process concluded with the definition of *FT-Corba* specified as a set of interfaces, policies and services that provide support for applications requiring high reliability. FT-Corba is based on the notion of object groups and insulates clients of such groups from the details of group management, reliable group communication, failure masking and recovery.

1.1 The Problem

In a recent paper [8], Frolund and Guerraoui criticize the fact that the new FT-Corba standard considers replications and transactions as separate aspects of fault tolerance. As a result, composition of FT-Corba and OTS does not result in any meaningful combination of their respective strengths. The problem lies in the fact that these services have been specified as coarse-grained abstractions, thus precluding the possibility of developing Corba-compliant replication and transaction services that are both replication-aware and transaction-aware. The authors advocate an approach where the fundamental building blocks of these services are standardized.

To illustrate the point, consider the following OTS example. Each transaction is conceptually driven by a *coordinator object*. Coordinators are based on a "crash-restart" recovery model where they maintain their recovery state in a disk-based log file. Transaction participants must remain blocked until their coordinator recovers. In the current OTS model, the recovery time of the coordinator may be long. Not only the coordinator process has to be restarted, it has also to read the log file and rebuild its state. In many situations, replication-based fail-over for coordinators would be preferable. Unfortunately, using FT-Corba to replicate coordinator objects is not straightforward, as there is no access to the coordinator definition through the standardized OTS API.

1.2 Contribution

Corba is one of the most important industrial standard for building distributed applications. Recently, however, other notable alternative distributed framework, such as Jini [2] and Enterprise JavaBeans [18], have been gaining a considerable share of the distributed application market. In order to promote the utilization of group communication in distributed applications, we believe that this paradigm should be integrated in these frameworks as well.

In this paper, we present Jgroup [20], an object group system that enhances the Jini model with reliable group communication. The aim of Jgroup is to enable construction of replicated Jini services capable of being federated in a standard Jini distributed system. As in Corba, this goal requires not only the presence of an object group service, but also a strict integration between the transactional model promoted by Jini and the group communication model. Unlike Corba, however, this integration is much simpler since the Jini transaction specification has been designed in order to precisely identify the different roles that objects may assume in transactions, such as transaction managers, participants and clients.

The rest of the paper is organized as follows. In Section 2, we provide a short introduction to Jini. In Section 3, we discuss how we are integrating the reliable group communication model of Jgroup and the transactional model of Jini. Finally, conclusions are drawn in Section 4, together with directions for future work.

2 Jini

The most important concept within the Jini architecture is that of a *service* — a software component providing some facility, such as a computation or storage service. Syntactically, services appear as objects written in Java, with an interface defining the operations that can be requested on them.

The aim of Jini is to federate services and their clients into a single, dynamic distributed system. The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time according to demand, need, or the changing requirements of the distributed applications. Beyond services, components of the Jini architecture may be divided in two other categories: *infrastructure* and *programming model*.

2.1 The Jini Infrastructure

The infrastructure is the set of components that enables building a federated Jini system, and defines the minimal Jini core. The infrastructure is composed of the *Java RMI protocol* [17], which enables objects to communicate through remote method invocations, and the *lookup service* [2], which provides a central registry for services.

Java RMI is based on the concept of *proxy*, which is the local representative of the remote entity involved in the invocation. Client-side proxies are called *stubs*, while server-side proxies are called *skeletons*. A stub implements the same remote interfaces as the remote object it represents, and forward method invocations received from clients to the appropriate skeleton. Skeletons wait for remote method invocations, and dispatch them to their remote objects. Together, stubs and skeletons take care of all low-level details of communication between clients and servers.

Services are found and resolved through the lookup service, which enables the registration of proxies for them. More precisely, a lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement the service. In addition, descriptive *attributes* associated with a service allow more fine-grained selection of services based on properties understandable to people.

2.2 The Jini Programming Model

The Jini specification defines the programming model as a set of interfaces that enables "the construction of reliable services", including those that are part of the infrastructure (as the lookup service) and those that join into the federation. The programming model is based on three distinct paradigms for distributed computing: *events*, *leases* and *transactions*.

The event notification interfaces enable event-based communication between Jini services. An object may allow other objects to register interest in events in the object and receive a notification of the occurence of such an event. This enables distributed event-based programs to be written with a variety of reliability and scalability guarantees.

The lease interface extends the Java programming model by adding time to the notion of holding a reference to a resource, enabling references to be reclaimed safely in the face of partitions. As an example, registrations in the lookup service are leased: a service must periodically renew its registration, otherwise its proxy is removed when its lease expires.

The transaction interfaces introduce a lightweight, object-oriented protocol enabling Jini services to coordinate state changes. The Jini transaction protocol differs from existing transaction interfaces (e.g., those defined in Corba) in that it does not assume that transactions follow a particular transaction semantics. Jini takes a more object-oriented view, leaving the correct implementation of transaction semantics up to the designer of the object involved in the transaction. The Jini transaction specification has identified the basic components of a transaction, such as transaction *clients*, *managers*, and *participants*. Transaction clients start a transaction by contacting a transaction manager through a proxy. The proxy is obtained by querying the lookup service for a service implementing the manager interface. The transaction manager responds with a *semantic transaction object*, which will represent the transaction in subsequent communications and contains information such as an identifier for the transaction and the proxy for the transaction manager. At this point, clients interact with participants by communicating the semantic object and the operation requested. The participants use the semantic object to communicate their commit/abort vote to the transaction manager. The transaction manager oversees the consistent execution of the operations and guarantees that all participants will eventually know if they should commit the operations or abort them. All interactions between clients, managers and participants are based on the Java RMI protocol.

Despite the claims made in the Jini specification, the Jini architecture does not provide an adequate support for development of reliable and high-available distributed applications. In particular, neither the architecture nor the programming model provide support for implementing a service as a group of replicated objects.

3 Integrating Jgroup and Jini

The Jgroup middleware system [19, 20], currently under development at the University of Bologna, is an object group communication service completely written in the Java language. Jgroup is being integrated with Jini in order to provide a more suitable environment for the development of reliable Jini services. The aim of this section is to describe the main characteristics of Jgroup and its integration with Jini, and discuss some of the open issues.

Jgroup promotes dependable application development through replication based on the *object group* paradigm [5, 15]. In this paradigm, distributed services that are to be made dependable are replicated among a dynamic 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 client objects as a single service.

Coordination among group members is obtained through the facilities provided by the partitionable group membership service, the reliable group communication service and the state merging service included in Jgroup [19].

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 an object group rather than a single server object. In particular, clients are unaware of the number, location or identity of individual server objects in the group.

The integration between Jini and the object group communication model provided by Jgroup has to be performed at each level of the Jini architecture, thus involving both infrastructure and programming model.

3.1 Extending the Jini Infrastructure: Java RMI

In order to extend the Jini infrastructure for supporting the reliable group communication model, the first step to be taken is the extension of the Java RMI protocol. In Jgroup, communication between clients and groups takes the form of *reliable group method invocations*, that result in methods being executed by one or more servers forming the group. Jgroup provides two different invocation semantics:

- The *anycast* invocation semantics guarantees that a method invocation performed by a client on a group of servers will be executed by invoking the method on at least one of the servers, unless the client is completely partitioned from the object group. Anycast invocations are suitable for implementing methods that do not modify the replicated state, such as in query requests to interrogate a database.

- The *multicast* invocation semantics guarantees that a method invocation performed by a client on a group of servers will be executed by invoking the same method on every reachable server. Multicast invocations are suitable for implementing methods that may update the replicated state of an application.

For clients, group method invocations are indistinguishable from standard RMI interactions: clients obtain a *group proxy* that is indistinguishable from a standard stub used for accessing non-replicated services.

Group proxies handle all low-level details of reliable group method invocations for clients, such as locating the servers composing the group, establishing communication with them and returning the result to the invoker. Proxy objects on the server side guarantee the reliability of the multicast method invocations, thus forwarding invocations to other servers when needed.

Unfortunately, the current Java RMI API does not allow the protocol to be easily extended. On the client side, RMI stubs contain *remote references* that maintain information needed to locate their remote counterparts and implement the communication protocol on behalf of clients. Java RMI enables the use of customized remote references; nevertheless, there is no analogous customizable entity on the server side. This lack makes remote references completely useless for group communication, as they allow modification of the communication protocol only on one side.

In order to avoid this limitation, we have had to modify the standard `rmic` compiler that is responsible for generation of stub and skeleton classes. We have added an additional communication layer on both sides that deals with all details of group invocations, such as reliability, eliminations of duplicates and result selection.

The additional communication layer is implemented as follows. Object groups are located by a group proxy 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. 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 group proxy have since left the group, the group proxy contacts the lookup service again in order to obtain fresher information about the group membership.

In the case of invocations with anycast semantics, the group proxy selects one of the servers composing the group and tries to transmit the invocation to the corresponding server proxy through a standard RMI interaction. The contacted server proxy dispatches the method at the server and sends back the return value to the group proxy. 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 group proxy 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 remote objects is exhausted. In the latter case, the group proxy throws a remote exception to the client, in order to notify that the requested service cannot be accessed.

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

3.2 Extending the Jini Infrastructure: the Lookup Service

The Jini lookup service has required modifications for dealing with group proxies. The reference implementation of the lookup service enables registration of customized proxies for services.

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

3.3 Extending the Jini Programming Model

In order to integrate group communication with the transactional model, we are following a two-step approach. In the first step, we have implemented a replicated transaction manager using the group communication facilities of Jgroup. For clients and participants, the presence of a replicated transaction manager is completely transparent: clients obtain a group proxy for the transaction manager by querying the lookup service for a service implementing the manager interface, as in the non-replicated case. As noted before, group proxies are totally indistinguishable from standard RMI proxies (both of them simply implement the remote interfaces they represent). Clients start a transaction by invoking a method on the group proxy, and obtain a semantic transaction object. Participants are notified by clients about transactions, together with information about the transaction manager enclosed in the semantic object. Following the Jini specification, participants interact with the transaction manager through the semantic object. The semantic object (actually, the group proxy for the manager included in it) hides from participants the fact that the transaction manager is replicated. Having a replicated transaction manager guarantees higher availability of the transaction service than that possible through the "crash-recovery" model offered by the reference implementation of the Jini transaction specification.

In this first step, we have considered only the role of transaction manager as a candidate for replication. In the second step, we are completing the integration between group communication and transactions by enabling each of the roles identified in the Jini specification to be performed by a group of replicated objects. For example, an object group acting as a client may contact a replicated transaction manager, in order to request a set of operations to be performed by a collection of services. Some of these services could be non-replicated objects, while other could be object groups. As before, our aim is to provide complete transparency for non-replicated entities. This means that non-replicated participants should receive a single request from replicated clients, while non-replicated clients should not be required to be aware that they are requesting an operation to be performed by an object group. Once again, the elimination of duplicated requests and the enforcing of the reliability guarantees is performed by proxies.

4 Concluding Remarks and Future Work

Integrating reliable group communication with existing distributed development frameworks such as Corba [11], DCOM [4], Jini [2] and Enterprise JavaBeans [18] is necessary to promote a broader diffusion of this technology within industry. In this paper, we have described how an object group toolkit could be integrated with Jini. The transactional interfaces of Jini are well-suited for integration with group communication, as they precisely identify the fine-grained roles that may be performed by objects in a transaction, without specifying a monolithic service as in Corba.

Other examples of integration between distributed object group frameworks and group communication toolkits include Electra [14], Newtop [21], OGS [6] and Eternal [22], which extend Corba; Filterfresh [3] and Javagroups [21], which extend Java RMI; and Comera [24], which extends DCOM. Among these, perhaps the most important system is Eternal, which started as an academic project and now is a commercial product. The Eternal team contributed to the definition of the FT-Corba specification [10] and produced the first implementation of it. As noted above, none of these systems consider the problem of integrating group communication with the transactional model provided by the frameworks they extend.

Jgroup is the subject of a collaborative research project between Sun Microsystems and the University of Bologna. The first result of this collaboration is the definition of a new Java RMI API that will enable users to plug in custom behaviors in the Java RMI protocol [16]. In particular, the new RMI API enables developers to customize both the client and the server side of the interaction, making it suitable for implementing the reliable group method invocation semantics of Jgroup.

In their paper [8], Frolund and Guerraoui argue that the combination of the replication and transactional services included in Corba is not adequate for the development of reliable applications based on pure three-tier architectures for which Corba is considered appropriate. Enterprise JavaBeans [18] is an alternative framework which explicitly promotes the development of three-tier applications. We plan to exploit the experience gained in putting together Jgroup and Jini in order to integrate group communication in the Enterprise Javabeans model.

References

[1] T. Anker, G. Chockler, D.Dolev, and I. Keidar. Fault-Tolerant Video-on-Demand Services. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

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

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

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

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

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

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

[8] S. Frolund and R. Guerraoui. Corba Fault-Tolerance: why it does not add up. In *Proceedings of the1999 IEEE Workshop on Future Trends in Distributed Computing (FTDCS)*, Capetown, December 1999.

[9] Object Management Group. CORBA Services – Transaction Service. Technical report, Object Management Group, Framingham, Ma, November 1997.

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

[11] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 2.3*. OMG Inc., Framingham, Mass., June 1999.

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

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

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

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

[16] Sun Microsystems. JSR 78 - RMI Custom Remote Reference. Url: `http://java.sun.com/about Java/communityprocess/jsr/jsr_078_rmicrr.html`.

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

[18] Sun Microsystems. *Enterprise JavaBeans Specification, Version 1.1*. Sun Microsystems, Inc., Mountain View, California, December 1999.

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

[20] A. Montresor, R. Davoli, and Ö. Babaoğlu. Middleware for Dependable Network Services in Partitionable Distributed Systems. In *Proceedings of the First PODC Workshop on Middleware*, Portland, Oregon, July 2000.

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

[22] 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.

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

[24] Y. M. Wang and W. Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.