Recent Titles from the UBLCS Technical Report Series


A Reliable Registry for the Jgroup Distributed Object Model

Alberto Montresor

Technical Report UBLCS-99-03

March 1999

Abstract

The registry service is a fundamental component of Java RMI. A registry is a repository facility used by remote servers to advertise their availability to provide certain services, and by clients to retrieve remote servers by simple names. Unfortunately, the service provided by the Java RMI registry (as the rest of the Java RMI architecture) is not reliable: partitioned or crashed registries force clients to engage complex recovery protocols in order to locate alternative registry instances. In this paper we present the design and the implementation of the reliable registry service included in Jgroup, an extension of Java RMI based on the group communication paradigm. Aim of Jgroup is to support the development of dependable applications in partitionable distributed systems. The Jgroup registry is based on a set of distributed remote objects that cooperate using the group communication service provided with Jgroup, and may be invoked as they were a single entity using the Jgroup invocation mechanism.
1 Introduction

In the last decade, the distributed object technology has proven to be a successful paradigm in dealing with the increasing complexity of distributed systems. Examples of popular distributed object frameworks are Java Remote Method Invocation [14, 10] and Corba [6]. These middleware platforms enable distributed objects to interact using a client/server approach. Client objects are allowed to access the services provided by server objects by issuing remote method invocations on them. Each server presents a well-defined interface that describes the set of methods that can be remotely invoked by clients. The 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.

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

In order to overcome these difficulties, the object group paradigm has been proposed [9]. In this paradigm, the functionalities of a remote server are replicated among a distributed group of objects. Clients transparently interact with object groups as if they were single, non-replicated entities. Replicated remote objects forming a group cooperate in order to provide a dependable and high-available service to their clients. This cooperation is established through the facilities offered by a group communication service (GCS) [3, 13, 4, 1], that enables the creation of dynamic groups of objects that communicate through reliable multicast primitives. Objects forming a group are kept informed about the current membership of the group itself, that may vary due to accidental events such as failures and repairs, or to voluntary requests to join or leave the group. This information may be used by objects to reconfigure themselves in order to provide clients with a consistent behavior.

Several examples of group-enhanced distributed object models exist; for example, Orbix+Isis [7] and Electra [8, 7] integrate CORBA with group communication toolkits such as Isis [3] and Horus [13], while Filterfresh [2] is an extension of Java RMI. All these distributed object models are based on primary-partition group communication services [12]. The primary-partition approach is intended for systems with no network partitionings, or for applications that require that the computation is carried on in at most one network partition, the so-called “primary partition”. This is a serious limitation for modern large-scale distributed systems, characterized by highly partitionable communication networks. Applications based on the primary-partition approach cannot guarantee continued availability outside the primary partition; moreover, particular failure scenarios may completely block a primary-partition GCS, and cause the complete blocking of applications based on it.

In order to provide a systematic support for the development of dependable applications in partitionable systems, we have designed Jgroup, an extension of Java RMI based on a partitionable group communication service [1]. The partitionable approach to group communication enables the members of a group to carry on the computation and to be continuously available in multiple, concurrent partitions. Each partition works independently until it merges with other partitions, in which case a reconciliation protocol is established among the members of the merging partitions. Jgroup includes also a group-oriented invocation mechanism that enables clients to transparently invoke methods on a group of remote objects as if they were a single, non-replicated remote object.

The architecture of Jgroup is the subject of a companion work [11]. Aim of this paper is to present the design and the implementation of a dependable version of the registry service included in Jgroup. The registry service is a fundamental component of Java RMI [10]. Registries are repository facilities that maintain a mapping between simple names and remote objects, used by remote servers to advertise their availability to provide certain services, and by clients to re-
retrieve remote servers by simple names. Unfortunately, Java RMI registries cannot be used with Jgroup since they are not able to associate simple names to groups of remote objects. Moreover, they constitute single points of failure: partitioned or crashed registries force clients to engage complex recovery protocols in order to locate alternative registry servers. The registry service included in Jgroup is composed by a distributed collection of remote objects maintaining a database of bindings \((name, group of remote objects)\). The remote objects forming a Jgroup registry cooperate using the GCS provided with Jgroup, and may be invoked as a single Java RMI registry through the Jgroup invocation mechanism.

2 The Jgroup Distributed Object Model

In the Java RMI model, a remote object is characterized by the fact that some of its methods can be invoked from other Java virtual machines, potentially on different hosts. The set of methods of a remote object that can be remotely invoked is defined by one or more remote interfaces. Clients of a remote object never interact with the actual implementation class of this object; instead, they invoke methods on a local surrogate object called stub, that implements the same set of remote interfaces and acts as a proxy of the remote object. Method invocations, together with their arguments, are marshaled by the stub and sent to the remote object. On the server side, a skeleton object unmarshals the method invocations and dispatches them to the corresponding methods of the remote object.

The Java RMI paradigm is not suitable for the development of dependable applications. On the client side, Java RMI includes only unicast invocation mechanisms, thus enabling only the execution of simple client-server interactions. Partitioned or crashed servers cause the raising of remote exceptions, informing the clients that the service they requested cannot be accessed. Thus, clients need to know multiple instances of a remote service and to be capable to select a reachable and operational instance among them. On the server side, Java RMI does not include any support for easing the cooperation of remote object groups, thus forcing developers to implement complex consistency protocols. Aim of Jgroup is to provide a systematic support for overcoming these difficulties in partitionable systems.

Jgroup is based on two fundamental abstractions: remote object groups and replicated remote objects. From the server’s point of view, a remote object group consists of a dynamic collection of replicated remote objects (sometimes called replicas for brevity) that implement the same set of remote interfaces and coordinate their executions in order to appear, whenever possible, as a non-replicated remote object. Replicas forming a remote object group cooperate using a GCS, whose task is to simplify the development of the consistency protocols needed to offer a reliable and high-available service. The composition of a remote object group is dynamic; remote objects may join and leave a group at run-time.

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

A detailed analysis of the Jgroup group communication service is out of the scope of this work. The specification of our GCS, together with an high-level algorithm implementing it, and a general overview about the Jgroup architecture are contained in previous papers \([1, 11]\). The next section contains only a brief presentation of the group communication paradigm and the main characteristics of the our GCS, while Section 2.2 describes the group-oriented invocation mechanism of Jgroup.
2.1 The partitionable group communication service

The object group paradigm enables the creation of dynamic groups of replicated remote objects that collaborate towards some common goal using the facilities offered by a GCS. The membership of a group is dynamic due to both voluntary requests such as objects joining or leaving the group, and accidental events such as replica crashes and recoveries or network partitionings and mergings. A GCS can be subdivided in two logical units: a group membership service and a reliable multicast service. Task of the former is to keep replicas consistently informed about changes in the current membership of the group through the installation of views. Installed views consist of a collection of replicas and represent the perception of the group’s membership that is shared by the replicas composing the view itself. In other words, there has to be agreement among the replicas on the composition of a view before it can be installed. Differently from other object group systems, the GCS provided with Jgroup is partitionable: the group membership service allows the installation of multiple, concurrent views, each of them representing one network partition. Task of the reliable multicast service, on the other hand, is to enable replicas forming a group to communicate by multicasting messages. Message deliveries are integrated with view installations as follows: two replicas that install the same pair of views in the same order deliver the same set of messages in the period occurring between the installations of these views. This delivery semantics, called view synchrony, enables replicas to reason about the state of other replicas using only local information such as the current view composition and the set of messages delivered in the previous view.

Replicas must be able to react opportunely to new failure scenarios as depicted by views, for example by modifying the quality of the service they provide or trying to recover from previous failures. The fact that our partitionable group membership service allows the existence of multiple, concurrent views enables members to provide continuously available services in every partition of the system, but requires particular attention from application developers. Replicas contained in a partition may carry on the computation independently from the replicas not contained in it; but when the partition merges with another one, a reconciliation protocol is needed. This reconciliation protocol depends on the application semantics and must be carefully designed in order to minimize the number of messages exchanged. Section 3 contains an example of reconciliation protocol.

Jgroup replicas have access to three primitives: join\( (g) \) and leave\( (g) \) are used to respectively join and leave a group, while mcast\( (g, m) \) is used to multicast a message \( m \) to the replicas contained in the last view installed by the caller for the group \( g \). In the Jgroup API, these primitives are provided by a Group Manager object. Replicas must be able to react to two different events: vchg\( (g, v) \), representing the installation of view \( v \) for the group \( g \), and dlvr\( (g, m) \), representing the delivery of the message \( m \) for group \( g \). Each view \( v \) is given a unique identifier \( v.id \) and consists of a collection of replica identifiers \( v.comp \). In the Jgroup API, replicas must implement the vchange and deliver methods contained in the Member interface. These methods are invoked, when needed, by the group manager.

2.2 Remote Method Invocation Semantics

The Jgroup invocation semantics guarantees that a method invocation performed by a client on a group of replicas will be executed by invoking the same method on one of the replicas, provided that the partition of the client contains at least one operational replica. The invocation mechanism attempts to sequentially invoke the replicas forming a group, until a reachable replica is found (in which case the invocation successfully terminates), or none of the replicas can be contacted (in which case an exception is raised). This invocation mechanism is completely transparent to clients, that simply invoke the methods on their local stub. Note that due to the Two Generals Problem [5], we cannot guarantee that a method is invoked on at most one of the replicas. Consider the following scenario in which the same method is executed on two replicas: the first replica contacted receives the invocation request, executes the method, but is unable to deliver the return value due to a sudden partitioning. Then, the invocation mechanism of the client contacts another replica that correctly executes the method. Duplicated method invocations must be
carefully handled when developing replicated applications (e.g., consider an update method on a replicated database).

3 The Jgroup Registry

As described in the introduction, the Java RMI architecture includes a remote server called registry, used as a repository facility for registering remote objects and retrieving stubs for them. Each registry object acts independently and maintains a different set of bindings \((name, \text{remote object})\), thus constituting a single point of failure. Furthermore, a registry object running on a certain host can be used to register only remote objects running on the same host; thus, in order to obtain a stub for a certain service, a client need to be aware of its location.

The registry implementation included with Jgroup has several advantages over the standard Java RMI registry. First, it enables a dynamic set of replicated remote objects to register themselves under the same name. This set forms a remote object group that can be retrieved as a single entity. The Jgroup registry objects running in a distributed system cooperate in order to maintain a replicated database of bindings and offer a reliable and high-available repository facility. Clients no longer need to be aware of the location of a service: they can simply access the group of registry replicas (whose stub is obtained through a bootstrap mechanism) and obtain a stub for every service registered in the distributed system. A distributed system can be designed by including a certain number of registry replicas running on different hosts and possibly on distinct portions of the communication network. Clients access these replicas through standard RMI interactions as if they were a single registry, and are guaranteed that their invocation will successfully terminate, provided that at least one operational replica is running in their partition.

During a partitioning, the Jgroup registry presents a partitioned behavior reflecting the current failure scenario. The registration of a remote object inside a partition will not affect the registry replicas not contained in that partition, while a retrieval operation will not success if the service we are looking for has been registered in another partition. Nevertheless, the replicas contained in a partition consistently maintain the same set of bindings and act as a single entity; moreover, the disappearance of the partitioning causes the execution of a reconciliation protocol in order to re-establish a consistent set of bindings among the replicas that belonged to different partitions. It is important to note that this behavior is perfectly reasonable for a partitionable distributed system, since clients asking for remote services are interested only in servers running in their current partitions.

The Jgroup registry interface comprises two kind of methods: retrieval (read-only) methods such as lookup, and update (write) methods such as bind and remove. The \texttt{Remote lookup(String name)} returns the stub for the remote object group identified by a certain name. The \texttt{BID bind(String name, Remote remote)} method is used to add a replicated remote object to the remote object group registered under a certain name. It returns a binding identifier \(\texttt{BID}\) used to globally identify the registration of this replicated remote object. \(\texttt{BID}\)s are generated by the local registry stubs, and are composed by the identifier of the client issuing the bind operation plus an invocation timestamp (we assume that local clocks, although not real-time, never output the same value twice). \(\texttt{BID}\)s are used to uniquely identify every bind operation, in order to solve the problems originated by the possibility of duplicated invocations of the same method on different replicas. Moreover, \(\texttt{BID}\)s are used in the \texttt{void remove(BID bid)} method to remove a replicated remote object from a group; the replica to be removed is identified by the binding identifier obtained when it has been registered.

An high-level description of the Jgroup registry algorithm is illustrated in Figure 1. Apart from the initialization, in which the replica joins the registry group identified by \(\texttt{RG}\), the registry code can be subdivided in a client section, containing the methods that can be remotely invoked by clients, and a server section, containing the methods associated to group communication events such as view installations and message deliveries. For simplicity, we assume that all methods are \textit{synchronized}: they cannot be executed concurrently.

Every registry replica \(p\) maintains a set of variables as follows. Given a replica \(q\), variable
Initialization:

1. **foreach** \( q \in \Pi \) do
2. \( \text{bound}[q] = \emptyset \)
3. \( \text{removed}[q] = \emptyset \)
4. **od**

5. **function** lookup(name)
6. return \( \{ (\text{name}, r, \text{bid}) \in \text{bound}[] \cup \text{newb} \land \text{bid} \notin \text{removed}[], \text{newb} \} \)
7. **function** bind(name, remote, bid)
8. mcast(RG, \{BIND, name, remote, bid\})
9. return bid
10. **procedure** remove(bid)
11. mcast(RG, \{REMOVE, bid\})
12. **dlvr**(g, \{BIND, name, remote, bid\})
13. **od**
14. **dlvr**(g, \{REMOVE, bid\})
15. **newb** = **newr** = \( \emptyset \)
16. lastcomp = \{me\}
17. lset = mset = \{me\}
18. join(RG)
19. ** procedure** remove (bid):
20. mcast(RG, \{REMOVE, bid\})
21.
22. **dlvr**(g, \{BIND, name, remote, bid\})
23. **newb** = **newr** = \( \{ (\text{name}, \text{remote}, \text{bid}) \} \)
24. **dlvr**(g, \{REMOVE, bid\})
25. **newr** = **newr** \cup \{bid\}
26.
27. **forall** \( q \in \text{v.comp} \cap \text{lastcomp} \) do
28. **if** mset \( \neq \text{lastcomp} \) and \( \text{Min}(\text{mset}) = \text{me} \) then
29. mcast(RG, \{UPDATE, lset, bound[me] \cup \text{newb} - \text{mset \cap lastcomp} \}
30. mset = mset \cap lastcomp
31. **od**
32. **if** me \( \in \text{owners} \) then mset = lastcomp

**Figure 1. The Jgroup registry algorithm**

\( \text{bound}[q] \) is a set containing the bindings that \( p \) knows that \( q \) has added to its database, while variable \( \text{removed}[q] \) is a set containing the BID identifiers that \( p \) knows that \( q \) has removed. Obviously, partitionings may cause these sets to differ from the actual sets of bindings and removals applied by \( q \). These variables are used to minimize the amount of information exchanged during a reconciliation protocol. The sets \( \text{bound}[p] \) and \( \text{removed}[p] \) maintained by replica \( p \) (in the algorithm indicated as \( \text{me} \)) have a different meaning. They represent the sets of insertions and removals actually applied by \( p \), with the exclusion of those performed during the current view, respectively contained in variables \( \text{newb} \) and \( \text{newr} \). Each item of \( \text{bound} \) and \( \text{newb} \) is a triple formed by the name of the service, the remote object and a BID identifier. It is important to note that in the absence of partitionings, bindings removed by a replica \( q \) are eventually removed from the set \( \text{bound}[q] \) of every other replica; however, the set \( \text{removed}[q] \) can grow without limits. For this reason, in the actual implementation of the algorithm bindings are \( \text{leased} \): periodically, every replica removes from the \( \text{removed} \) sets the binding ids for which the corresponding binding leases have expired.

Variables \( \text{lastcomp}, \text{mset} \) and \( \text{lset} \) are used to determine what information needs to be exchanged and who is responsible for the exchange during the reconciliation protocol started after the installation of a new view. Variable \( \text{lastcomp} \) maintains the composition of the current view and is updated at every view installation. The algorithm maintains the following invariant at replica \( p \) with respect to \( \text{mset} \) and \( \text{lset} \): at the beginning of every view, each replica in \( \text{mset} \) has applied every update performed by \( p \), while \( p \) has applied every update performed by the replicas in \( \text{lset} \) (in both cases excluding the updates applied during the last view). Note that replicas in \( \text{mset} \cap \text{lset} \) have applied the same set of updates as \( p \). If a replica \( q \) of the current view is not in \( \text{mset} \), then \( q \) could be lagging behind with respect to replicas in \( \text{mset} \).

Since each registry replica maintains its copy of the update set, the lookup method can be executed locally: the replica that receives the method invocation inspects its local database and returns the set of remote objects that have been registered under a certain name and have not been removed. The behavior of update methods is different, since they involve the update of each replica forming the registry group. When a replica receives a bind or remove method
invocation, it multicasts a BIND or REMOVE message to the replicas in its current view. Replicas that deliver these messages modify their local variables newb or newr. When the system is stable (i.e. when no new failures and repairs occur), each network partition contains a set of replicas that will eventually install the same view and deliver the same messages. This implies that each update operation invoked inside that partition will be eventually performed by every replica in that partition.

The core of the algorithm is the code associated with the installation of new views reflecting changes in the failure scenario. When installing a new view, \( p \) adds the updates contained in newb and newr to bound[q] and removed[q], for each replica \( q \) in the intersection between the new view and the previous one. In other words, \( p \) memorizes the fact that every replica surviving from the previous view to the new one has delivered the same set of messages and thus has applied the same set of updates; this is guaranteed by the view synchrony semantics of the reliable communication service. The next step consists in verifying whether a reconciliation protocol is needed or not. For this reason, variables lastcomp, lset and mset are updated. The replicas not included in the new view are removed from lset and mset, to indicate that \( p \) cannot know the evolution of their database until communication is restored and a reconciliation is performed. If a replica \( q \) in the current view does not belong to mset, \( q \) may be lagging behind and a reconciliation protocol should be started. Only one replica in mset, selected through the Min function, is required to multicast the updates. Note that the reconciliation message does not contain any update operation that \( p \) knows has been already applied by all replicas. When this message is delivered, the update operations are applied to the local copy of the database; moreover, the sets lset and mset are modified to memorize that every update applied by the replicas in owners has been applied also by \( p \), and that every replica in the current view has applied every update applied by \( p \).

4 Conclusions

The Jgroup registry has several advantages with respect to the standard Java RMI registry. First, it enables multiple replicas of the same service to form a group by registering themselves under the same name. Clients do not need to be aware of the location of the service they want to access; their queries are addressed to the group of registry replicas running in the distributed system, located through a bootstrap mechanism. Finally, the service offered by the Jgroup registry is reliable.

The Jgroup registry is not the first repository service for remote object groups; as an example, consider the Filterfresh registry [2] or the Electra name server [7]. Nevertheless, the Jgroup registry is the first repository service offering a partitionable behavior. A replicated repository service whose consistency protocols are based on a primary-partition GCS suffers from two main problems: first, the service it provides is not high-available, since replicas in partitions different from the primary one cannot apply update operations on their database. Furthermore, the disappearance of the primary partition can completely block the service in the entire system.

The Jgroup registry, on the other hand, provides a high-available service, since the update operations are allowed in every partition containing at least one operational registry replica. An update will not be visible outside the partition where it has been requested until the communication with other partition is restored, in which case a reconciliation protocol among the inconsistent replicas of the database is performed. This behavior is perfectly suitable for the requirements of dependable applications in partitionable distributed systems, where a client running in a partition want to have continuous access to the registry and the services contained in its partition.

References


