# Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing

**Hein Meling** **Alberto Montresor** **Özalp Babaoğlu**
**Bjarne E. Helvik**

**Technical Report UBLCS-2002-12**

October 2002

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

# Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing

**Hein Meling** [3] **Alberto Montresor** [1] **Özalp Babaoğlu** [1]
**Bjarne E. Helvik** [3]

Technical Report UBLCS-2002-12

October 2002

**Abstract**

*We present the design and implementation of the Jgroup distributed object platform and its replication management framework ARM. Jgroup extends Java RMI through the group communication paradigm and has been designed specifically for application support in partitionable distributed systems. ARM is layered on top of Jgroup and provides extensible replica distribution schemes and application-specific recovery strategies. The combination Jgroup/ARM can reduce significantly the effort necessary for developing, deploying and managing dependable, partition-aware applications.*

3. Department of Telematics, Norwegian University of Science and Technology, O.S. Bragstadsplass 2A, N-7491 Trondheim (Norway), Email: {meling,bjarne}@item.ntnu.no
1. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy), Email: {montresor,babaoglu}@CS.UniBO.IT

## 1 Introduction

Our increasing reliance on network information systems in day-to-day activities requires that the services they provide remain *available* and the actions they perform be *correct*. A common technique for achieving these goals is to *replicate* critical system components whereby the functions they perform are repeated by multiple replicas. As long as replica failures are independent and their numbers can be bounded, the technique can be made to guarantee higher availability and correctness for the system than those of its components. Distributing replicas geographically is often effective for rendering failures independent.

Distributed object-based middleware platforms such as CORBA [23], Java RMI [25], Jini [1] and J2EE [26] hold the promise of simplifying network application complexity and development effort. Their ability to exploit commercial off-the-shelf components, cope with heterogeneity and permit access to legacy systems makes them particularly attractive for building application servers and three-tier e-business solutions. Yet, they remain unsuitable for implementing replication since the required "one-to-many" interaction model among objects has to be simulated through multiple one-to-one interactions [22]. This not only increases application complexity, but also degrades performance. The shortcoming has been recognized by numerous academic research projects [12, 6, 19], and also by the Object Management Group in its Fault Tolerant CORBA specification [24]. In the various proposals, distributed objects are replaced by their natural extension *distributed object groups* [11, 14]. Clients interact with an object group transparently through remote method invocations (RMI), as if it were a single, non-replicated remote object. Global consistency of the object group (visible through results of RMI) is typically guaranteed through a *group communication service* [9].

In this paper, we describe the design and implementation of *Jgroup*, a novel object group-based middleware platform, and ARM, the *Autonomous Replication Management* framework built on top of Jgroup. When used together, Jgroup and ARM provide a flexible platform for developing, deploying and managing dependable distributed applications based on replication.

Jgroup integrates the Java RMI and Jini distributed object models with object group technology and includes numerous innovative features that make it suitable for developing modern network applications. Jgroup promotes *environment awareness* by exposing network effects to applications that best know how to handle them. If they choose, operational objects continue to be active even when they are partitioned from other group members. This is in contrast to the *primary partition* approach that hides network effects as much as possible by limiting activity to a single *primary* partition while blocking objects in all other partitions. In Jgroup, applications become *partition-aware* through *views* that give consistent compositions of the object group within each partition. Application semantics dictate how objects should behave in a particular view. Jgroup also includes a *state merging service* as further support for partition-aware application development. Reconciling the replicated application state when partitions merge is typically one of the most difficult problems in developing applications to be deployed in partitionable systems. While a general solution is highly application dependent and not always possible, Jgroup simplifies this task by providing systematic support for certain stylized interactions that frequently occur in solutions. Jgroup is unique in providing a uniform object-oriented programming interface (based on RMI) to govern *all* object interactions including those within an object group as well as interactions with external objects. Other object group systems typically provide an object-oriented interface only for interactions between object groups and external objects, while intra group interactions are based on message passing. This heterogeneity not only complicates application development, but also makes it difficult to reason about the application as a whole using a single interaction paradigm.

Most object group systems, including Jgroup, do not include mechanisms for distributing replicas to hosts or for recovering from replica failures. Yet, these mechanisms are essential for satisfying application dependability requirements such as maintaining a fixed redundancy level. ARM is a replication management facility we have built on top of Jgroup to simplify implementation of application-specific replication policies [15]. A replication policy describe the quality of service mechanisms used by an application group, such as replication style and recovery strategy. ARM provides a simple interface through which a management client can install and remove object groups within the distributed system. After its installation, an object group becomes an "autonomous" entity requiring no user interaction, unless manual removal of the group is desired. In other words, ARM handles both replica distribution, according to an extensible distribution scheme, as well as replica recovery, based on a group-specific policy. This allows the creation of object groups with varying dependability requirements and recovery needs. ARM includes a correlation mechanism to collect and interpret failure notifications from the underlying group communication system. This information is used to trigger group-specific recovery actions in order to reestablish system dependability properties after failures. The properties of our framework as described above lead to what we call "autonomous replication management".

The rest of the paper is structured as follows. Section 2 states the system model and gives an overview of Jgroup and ARM. Section 3 presents the Jgroup distributed object model, and in Section 3.4 we briefly describe certain aspects of the Jgroup implementation, in particular those related to system configuration. Section 4 gives an architectural overview of the replication management framework and describes its core components. Some initial performance measurements are given in Section 5. Section 6 compares Jgroup and ARM with related work. Finally, Section 7 concludes the paper.

## 2 System Model and Architectural Overview

The context of this work is a distributed system composed of client and server objects interconnected through a network. The system is *asynchronous* in the sense that neither the computational speeds of objects nor communication delays are assumed to be bounded. Furthermore, the system is unreliable and failures may cause objects and communication channels to *crash* whereby they simply stop functioning. Once failures are repaired, they may return to being *operational* after an appropriate recovery action. Finally, the system is *partitionable* in that certain communication failure scenarios may disrupt communication between multiple sets of objects forming *partitions*. Objects within a given partition can communicate among themselves, but cannot communicate with objects outside the partition. When communication between partitions is reestablished, we say that they *merge*.

Developing dependable applications to be deployed in these systems is a complex and error-prone task due to the uncertainty resulting from asynchrony and failures. The desire to render services partition-aware to increase their availability adds significantly to this difficulty. Jgroup and ARM have been designed to simplify the development of partition-aware, dependable application by abstracting complex system events such as failures, recoveries, partitions, merges and asynchrony into simpler, high-level abstractions with well-defined semantics.

Jgroup promotes dependable application development through replication, based on the *object group* paradigm [11, 14]. In this paradigm, distributed applications are replicated among a collection of server objects (*replicas* for brevity) that form a group in order to coordinate their activities and appear to clients as a single server. Jgroup provides a collection of facilities aimed at simplifying the coordination among replicas. These facilities include a *group membership service*, a *reliable communication service* and a *state merging service* [9, 17]. Communication between clients and groups takes the form of *group method invocations* (GMI) [17], that result in methods being executed by servers forming the group. To clients, GMI interactions are indistinguishable from standard remote method invocations (RMI): clients obtain a representative object called a *stub* that acts as a *group proxy* and manages group invocations on behalf of clients. Group proxies handle all low-level details of GMI, such as locating the servers composing the group, establishing communication with them and returning the result to the invoker.

ARM extends Jgroup with a replication management facility whose task is to simplify the deployment of dependable applications by permitting the specification of application-specific quality of service requirements such as replication style and recovery strategy. ARM provides a simple interface through which a management client can install and remove object groups within the distributed system without having to specify explicitly the hosts on which the replicas should

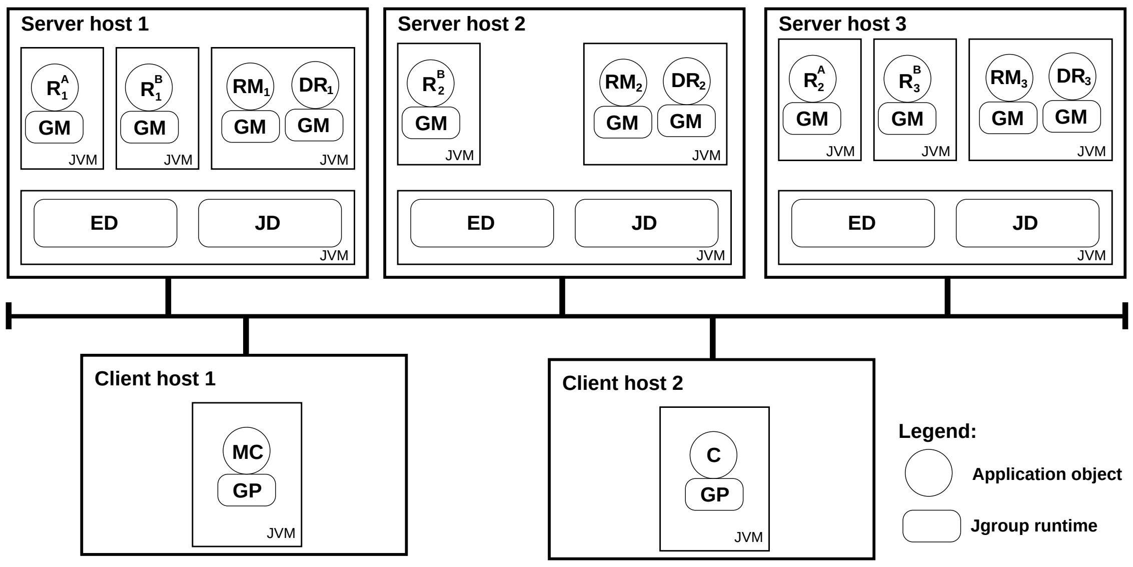

**Figure 1. Architectural Overview of Jgroup/ARM.**

be placed. Each application is associated a set of quality of service parameters, including the redundancy level, which ARM uses in conjunction with runtime monitoring of the distributed system to determine the replica placement of an object group. To facilitate its tasks, ARM uses a replication manager that localizes faults, performs failure analysis and reconfigures the system on-demand and according to application-specific requirements. This involves collecting failure information as provided by Jgroup, analyzing them in an effort to distinguish between different fault scenarios that may be responsible for them, and finally determining the appropriate reconfiguration actions to recover from them. Reconfiguration entails creation of additional replicas to substitute crashed or partitioned ones, or to removing excess replicas that may have been created due to incorrect failure suspicions or when partitions merge after repairs.

Figure 1 gives an overview of the principal components of Jgroup and ARM. The figure illustrates two applications that have been deployed: application $A$, implemented by replicas $R_1^A$, $R_2^A$ and application $B$, implemented by replicas $R_1^B$, $R_2^B$, $R_3^B$. The replicas have been distributed over three distinct hosts. Two other hosts are available for clients (C). As described above, communication between clients and groups is mediated by group proxies (GP), that transmit group method invocations to the group managers associated with replicas forming the group.

The main component of Jgroup is the *Jgroup daemon* (JD), that implements basic group communication services such as failure detection, group membership and reliable communication. Each machine hosting at least one replica must run a Jgroup daemon. Replicas belonging to different applications normally share the same daemon, but it is also possible to run several daemons on the same machine when distinct applications have completely different membership and communication requirements.

Each replica is associated a *group manager* (GM), whose task is to act as an interface between Jgroup daemons and the replica. Group managers are based on a layered architecture that is composed dynamically. Each layer exploits the basic group communication facilities implemented by Jgroup daemons and provides applications with complex group communication services. Description of services that can be composed in group managers can be found in Section 3.

Finally, in order to enable clients locate server groups, Jgroup include the *dependable registry* (DR), a replicated lookup service that allows dynamic groups of replicas to register themselves under the same name. Information about the replicas composing the group are collected in a group proxy, that can be retrieved by clients. This enables clients to seamlessly communicate with the whole group as a single entity, by performing invocations through the group proxy. Note that the dependable registry itself is implemented as a replicated object group using Jgroup services; therefore the DR replicas in Figure 1 are associated group managers just like application replicas.

The ARM framework is built on top of Jgroup and consists of three core components. The *management client* (MC) provides system administrators with a GUI-based management interface, enabling them to install and remove dependable applications in the system and to specify the replication style and recovery strategy to be used. The *replication manager* (RM) is the main component of ARM. Its tasks include replica distribution, failure logging and correlation, and interactions with the management client. The replication manager interacts with *execution daemons* (ED), whose task is to create and remove application replicas on demand. An execution daemon must be running on all hosts in the system that could possibly host replicas. Although replicated on each machine, execution daemons do not form a group, and the replication manager communicates with them using plain RMI.

On each host with application replicas or the associated Jgroup runtime, multiple Java Virtual Machines (JVM) are executed. This is motivated by the desire to enhance the failure independence of different Jgroup/ARM components. In particular, each application replica is executed in a different JVM so as to prevent misbehaving replicas from disrupting the Jgroup runtime or other applications located on the same host.

## 3 The Jgroup Distributed Object Model

Jgroup extends the object group paradigm to partitionable systems through three core components: a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS). An important aspect of Jgroup is the fact that properties guaranteed by each of its components have formal specifications, admitting formal reasoning about the correctness of applications based on Jgroup [17]. Due to space constraints, in this paper we give only short, informal descriptions.

### 3.1 The Partition-aware Group Membership Service

Groups are collections of server objects that cooperate in providing distributed services. For increased flexibility, the group composition is allowed to vary dynamically as new servers are added and existing ones removed. Servers desiring to contribute to a distributed service become a *member* of the group by *joining* it. Later on, a member may decide to terminate its contribution by *leaving* the group. At any time, the *membership* of a group includes those servers that are operational and have joined but have not yet left the group. Asynchrony of the system and possibility of failures may cause each member to have a different perception of the group's current membership. The task of a PGMS is to track voluntary variations in the membership, as well as involuntary variations due to failures and repairs of servers and communication links. All variations in the membership are reported to members through the *installation* of *views*. Installed views consist of a membership list along with a unique view identifier, and correspond to the group's current composition as perceived by members included in the view.

A useful PGMS specification has to take into account several issues (See [3] for a detailed discussion of the problem). First, the service must track changes in the group membership accurately and in a timely manner such that installed views indeed convey recent information about the group's composition within each partition. Next, we require that a view be installed only after agreement is reached on its composition among the servers included in the view. Finally, PGMS must guarantee that two views installed by two different servers be installed in the same order. These last two properties are necessary for servers to be able to reason globally about the replicated state based solely on local information, thus simplifying significantly their implementation. Note that the PGMS we have defined for Jgroup admits coexistence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications.

## 3.2 The Group Method Invocation Service

Jgroup differs from existing object group systems due to its uniform communication interface based entirely on GMI. Clients and servers interact with groups by remotely invoking methods on them. In this manner, benefits of object-orientation such as abstraction, encapsulation and inheritance are extended to internal communication among servers. Although they share the same intercommunication paradigm, we distinguish between *internal GMI* (IGMI) performed by servers and *external GMI* (EGMI) performed by clients. There are several reasons for this distinction:

- *Visibility:* Methods to be used for implementing a replicated service should not be visible to clients. Clients should be able to access only the "public" interface defining the service, while methods invoked by servers should be considered "private" to the implementation.
- *Transparency:* Jgroup strives to provide an invocation mechanism for clients that is completely transparent with respect to standard RMI. This means that clients are not required to be aware that they are invoking a method on a group of servers rather than a single one. Servers that implement the replicated service, on the other hand, may have different requirements for group invocations, such as obtaining a result from each server in the current view.
- *Efficiency:* Having identical specifications for external and internal GMI would have required that clients become members of the group, resulting in poor system scalability. In Jgroup, external GMI have semantics that are slightly weaker than those for internal GMI. Recognition of this difference, results in a much more scalable system by limiting the higher costs of full group membership to servers, which are typically far fewer in number than clients [5].

When developing dependable distributed services, internal methods are collected to form the *internal remote interface* of the server object, while external methods are collected to form its *external remote interface*. Proxy objects capable of handling GMI are generated dynamically (at runtime) based on the remote interfaces of the server object. A proxy object implement the same interface as the group for which they act as a proxy, and enable clients and servers to communicate with the entire group of servers using local invocations on the proxy object.

In order to perform an internal GMI, servers must obtain an appropriate group proxy from the Jgroup runtime running in the local JVM. Clients that need to interact with a group, on the other hand, must request a stub from a *registry service*, whose task is to enable servers to register themselves under a group name. Clients can then look up desired services by name in the registry and obtain their stub. Jgroup support two registry services. The first, called *dependable registry* [16], is derived from the standard registry included in Java RMI, while the second is based on the Jini lookup service [1]. The dependable registry service is an integral part of Jgroup and is replicated using Jgroup itself.

In the following sections, we discuss how internal and external GMI in Jgroup work, and how internal invocations substitute message multicasting as the basic communication paradigm. In particular, we describe the reliability guarantees that GMI provide. They are derived from similar properties that have been defined for message deliveries in message-based group communication systems [3].

### 3.2.1 Internal Group Method Invocations

Unlike traditional Java remote method invocations, IGMI return an array of results rather than a single value. IGMI come in two flavors: *synchronous* and *asynchronous*. In synchronous IGMI, the invoker remains blocked until an array containing the results from each server that completed the invocation can be assembled and returned to it. There are many situations in which such blocking may be too costly, as it can unblock only when the last server to complete the invocation has produced its result. Furthermore, it requires programmers to consider issues such as deadlock that may be caused by circular invocations. In asynchronous IGMI, the invoker does not block but specifies a *callback* object that will be notified when return values are ready from servers completing the invocation.

Completion of IGMI by the servers forming a group satisfies a variant of "view synchrony" that has proven to be an important property for reasoning about reliability in message-based systems [8]. Informally, view synchrony requires two servers that install the same pair of consecutive views to complete the same set of IGMI during the first view of the pair. In other words, before a new view can be installed, all servers belonging to both the current and the new view have to agree on the set of IGMI they have completed in the current view. This enables a server to reason about the state of other servers in the group using only local information such the history of installed views and the set of completed IGMI. Clearly, application semantics may require that servers need to agree, not only on the set of completed IGMI, but also the order in which they were completed. Jgroup leaves different ordering semantics for IGMI completions to be implemented at layers on top of the basic GMI service.

We now outline some of the main properties that IGMI satisfy. First, they are *live*: an IGMI is guaranteed to terminate either with a reply array (containing at least the return value computed by the invoker itself), or with one of the application-defined exception contained in the *throws* clause of the method being invoked. Furthermore, if an operational server $S$ completes some IGMI in a view, all servers included in that view will also complete the same invocation, or $S$ will install a new view. Since installed views represent the current failure scenario as perceived by servers, this property guarantees that an IGMI will be completed by every other server that is in the same partition as the invoker. IGMI also satisfy *integrity* requirements whereby each IGMI is completed by each server at most once, and only if some server has previously performed it. Finally, Jgroup guarantees that each IGMI be completed in at most one view. In other words, if different servers complete the same IGMI, they cannot complete it in different views. In this manner, all result values that are contained in the reply array are guaranteed to have been computed during the same view.

### 3.2.2 External Group Method Invocations

As with IGMI, EGMI also come in two flavors: *anycast* and *multicast*. An anycast EGMI performed by a client on a group will be completed by at least one server of the group, unless there are no operational servers in the client's partition. Anycast invocations are suitable for implementing methods that do not modify the replicated server state, as in query requests to interrogate a database. A multicast EGMI performed by a client on a group will be completed by every server of the group that is in the same partition as the client. Multicast invocations are suitable for implementing methods that may update the replicated server state.

Each method exposed to clients through the external remote interface have distinct invocation semantics. The default semantics for an external method is anycast, and methods with multicast semantics have to be tagged by including a specific exception in their *throws* clause.

Our implementation of Jgroup guarantees that EGMI are *live*: if at least one server remains operational and in the same partition as the invoking client, EGMI will eventually complete with a reply value being returned to the client. Furthermore, an EGMI is completed by each server *at-most-once*, and only if some client has previously performed it. These properties hold for both anycast and multicast versions of EGMI. In the case of multicast EGMI, Jgroup also guarantees view synchrony as defined in the previous section.

Internal and external GMI differ in one important aspect. Whereas an IGMI, if it completes, is guaranteed to complete in the same view at all servers, an EGMI may complete in several different concurrent views. This is possible, for example, when a server completes the EGMI but becomes partitioned from the client before delivering the result. Failing to receive a response for the EGMI, the client's stub has to contact other servers that may be available, and this may cause the same EGMI to be completed by different servers in several concurrent views. The only solution to this problem would be to have the client join the group before issuing the EGMI. In this manner, the client would participate in the view agreement protocol and could delay the installation of a new view in order to guarantee the completion of a method in a particular view. Clearly, such a solution may become too costly as group sizes would no longer be determined by the number of servers (degree of replication of the service), but by the number of clients, which could be very large.

One of the goals of Jgroup has been the complete transparency of server replication to clients. This requires that from a client's perspective, EGMI should be indistinguishable from standard Java RMI. This has ruled out consideration of alternative definitions for EGMI including multi-value results or asynchronous invocations.

## 3.3 The State Merging Service

While partition-awareness is necessary for rendering services more available in partitionable environments, it can also be a source of significant complexity for application development. This is simply a consequence of the intrinsic availability-consistency tradeoff for distributed applications and is independent of any of the design choices we have made for Jgroup.

Being based on a PGMS, Jgroup admits partition-aware applications that have to cope with multiple concurrent views. Application semantics dictate which of its services remain available where during partitionings. When failures are repaired and multiple partitions merge, a new server state has to be constructed. This new state should reconcile, to the extent possible, any divergence that may have taken place during partitioned operation.

Generically, state reconciliation tries to construct a new state that reflects the effects of all non-conflicting concurrent updates and detect if there have been any conflicting concurrent updates to the state. While it is impossible to automate completely state reconciliation for arbitrary applications, a lot can be accomplished at the system level for simplifying the task [2]. Jgroup includes a state merging service (SMS) that provides support for building application-specific reconciliation protocols based on stylized interactions. The basic paradigm is that of full information exchange — when multiple partitions merge into a new one, a coordinator is elected among the servers in each of the merging partitions; each coordinator acts on behalf of its partition and diffuses state information necessary to update those servers that were not in its partition. When a server receives such information from a coordinator, it applies it to its local copy of the state. This one-round distribution scheme has proven to be extremely useful when developing partition-aware applications [4, 16].

SMS drives the state reconciliation protocol by calling back to servers for "getting" and "merging" information about their state. It also handles coordinator election and information diffusion. To be able to use SMS for building reconciliation protocols, servers of partition-aware applications must satisfy the following requirements: (i) each server must be able to act as a coordinator; in other words, every server has to maintain the entire replicated state and be able to provide state information when requested by SMS; (ii) a server must be able to apply any incoming updates to its local state. These assumptions restrict the applicability of SMS. For example, applications with high-consistency requirements may not be able to apply conflicting updates to the same record. Note, however, that this is intrinsic to partition-awareness, and is not a limitation of SMS.

The complete specification of SMS is given in [17]. Here we very briefly outline its basic properties. The main requirement satisfied by SMS is *liveness*: if there is a time after which two servers install only views including each other, then eventually each of them will become up-to-date with respect to the other, either directly or indirectly through different servers that may be elected coordinators and provide information on behalf of one of the two servers. Another important property is *agreement*: servers that install the same pair of views in the same order are guaranteed to receive the same state information through invocations of their "merging" methods in the period occurring between the installations of the two views. This property is similar to view synchrony, and like view synchrony may be used to maintain information about the updates applied by other servers. Finally, SMS satisfies *integrity*: it will not initiate a state reconciliation protocol without reason, e.g., if all servers are already up-to-date.

## 3.4 Jgroup Implementation

To conclude our description of Jgroup, we give a brief overview of its implementation. Further details can be found in another work [17]. As discussed in Section 2, object group facilities are provided to clients and servers through stubs and group managers, respectively. Stubs act as proxies for clients performing external GMI on the group, while group managers are used by

```
<Configuration>
  <Transport payload="1024" maxTTL="10" TTLWarning="5"/>
  <DistributedSystem>
    <Domain name="cs.unibo.it">
      <Host name="cartoonia"/>
      <Host name="obelix"/>
    </Domain>
    <Domain name="item.ntnu.no" address="226.1.2.3" port="6156">
      <Host name="kilkenny" port="20000"/>
      <Host name="samson" port="20000"/>
    </Domain>
  </DistributedSystem>
</Configuration>
```

**Figure 2. Example configuration file for the distributed system**

servers to perform internal GMI. Group managers are also responsible for notifying servers of group events such as view changes and method invocations issued by other clients or replicas. Each stub and group manager is associated with exactly one group; stubs may serve several clients concurrently, while each group manager is associated with exactly one replica.

While stubs have total responsibility for handling external invocations on the client side, group managers implement only a subset of the object group services described in the previous sections. Basic group membership and multicast communication facilities are implemented in Jgroup daemons. There are several reasons for not putting the entire group service in the group managers but factoring out certain functions to the per-site Jgroup daemons. First, the number of messages exchanged to establish group communication is reduced and low-level services such as failure detection are implemented only once per-site. Furthermore, this model enables the distinction between server objects local to a given host and those that are remote. Servers local to a given host share the same destiny with respect to failures. Thus, two distinct membership lists are maintained: one regarding local servers, and the other regarding remote daemons (and their associated servers).

The facilities described in the previous sections and not included directly in Jgroup daemons are provided by group managers. Group managers have a layered architecture, and are composed specifically to satisfy the needs of the associated replica. Group manager layers that are currently available in Jgroup include group method invocation service, state merging service and several different ordering properties. Additional group manager components can be easily added to Jgroup in order to provide new facilities to developers. Application replicas may easily describe their requirements about group manager layers by specifying a *layer stack ordering* string. An example of such a string is shown in Figure 4, that contains all the configuration parameters needed by ARM to describe an application and its replication policies. When a group manager is built, the stack order is checked for structural correctness.

To enable communication among group members, Jgroup daemons need to know the set of hosts on which their peer group members may reside. To accomplish this, domains and hosts on which Jgroup daemons will run are specified through XML configuration files, as shown in Figure 2. Thus, when a Jgroup server attempts to join a specific group, it will probe the other hosts specified in this file in order to find members of its group. Clients, however, are not restricted to the hosts listed in the configuration file, but only need to know the location of the dependable registry in order to obtain a group proxy for accessing the server group [16].

A Jgroup domain coincides with a DNS domain, and may be associated a multicast address and port number. This permits Jgroup to exploit multicast facilities of the underlying network that may be available within that domain. Hosts have names and are associated port numbers to which servers will connect for Jgroup related services. In addition to domain and host specifications, a number of parameters related to the Jgroup transport layer can also be configured for performance tuning. All parameters are optional, and sensible default values are used for those left unspecified.

Group managers implement IGMI and EGMI facilities in cooperation with remote stubs residing at client sites. Object groups are located by a stub through *group references* that contain a standard Java RMI remote reference for each server. Group references are only approximations to the group's actual membership as known by the registry at the time of the interrogation. This is done to avoid updating a potentially large number of stubs that may exist in the system every time a variation in a group's membership occurs. On the negative side, group references may become stale, particularly in systems with highly-dynamic, short-lived server groups. Consequently, when an external group method invocation fails because all servers known to the stub have either left the group or are unreachable, the stub has to re-contact the registry in order to obtain fresher information about the group membership.

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

At the client stub, external invocations with multicast semantics proceed just as those with anycast semantics. The group manager receiving the invocation multicasts it to all members in its current view. A single return value (usually the one returned by the group manager initially contacted) is returned to the client stub. Note that a direct multicasting of the invocation to all servers cannot be used since the actual composition of a group may be different from that of the group reference maintained by a stub. In any case, an additional communication step among servers is necessary in order to transmit the invocation to servers not included in the group reference and to guarantee view synchrony.

## 4 Autonomous Replication Management

The ARM framework has been designed on top of Jgroup to facilitate automatic installation and distribution of object replicas and recovery from failures. These are fundamental tasks for dependable application deployment and management. The goal of the recovery mechanism in ARM is to localize faults, perform failure analysis and reconfigure the system according to application-specific requirements. Jgroup deals with low-level fault detection, providing ARM with view change events, and reconfigures the object group to reflect the correct membership at all times. Recovery, on the other hand, is a concerted effort requiring contributions from ARM, Jgroup and the application itself in restoring the lost state due to failures.

Figure 3 gives an overview of the components in ARM, and illustrates how they interact (through their core access interfaces). In the following, we walk through two example scenarios; the creation of an object group, and recovery from replica failure by relocating away from a crashed host.

To install a dependable application, the *Management Client* (MC) invokes the createGroup method (À) on the *Replication Manager* (RM). Upon receiving such a request, the RM will add the group ( ` ) to its *Group Table* (GT), and obtain from the *Distribution Scheme* (DS) a set of hosts ( ´ ) on which the object group should reside. Although not shown in the figure, before assigning a replica to a given host, the distribution scheme will query the *Daemon Manager* (DM) to check the availability of the potential hosts. Given a set of hosts, the RM obtains a reference ( ˆ ) to the *Execution Daemon* (ED) running on the respective hosts from the DM, and invokes the createReplica ( ˜ ) method on each host. The dashed arrow in the figure is meant to indicate that ED ensures the creation of a local JVM process to hold the *replica* (R).

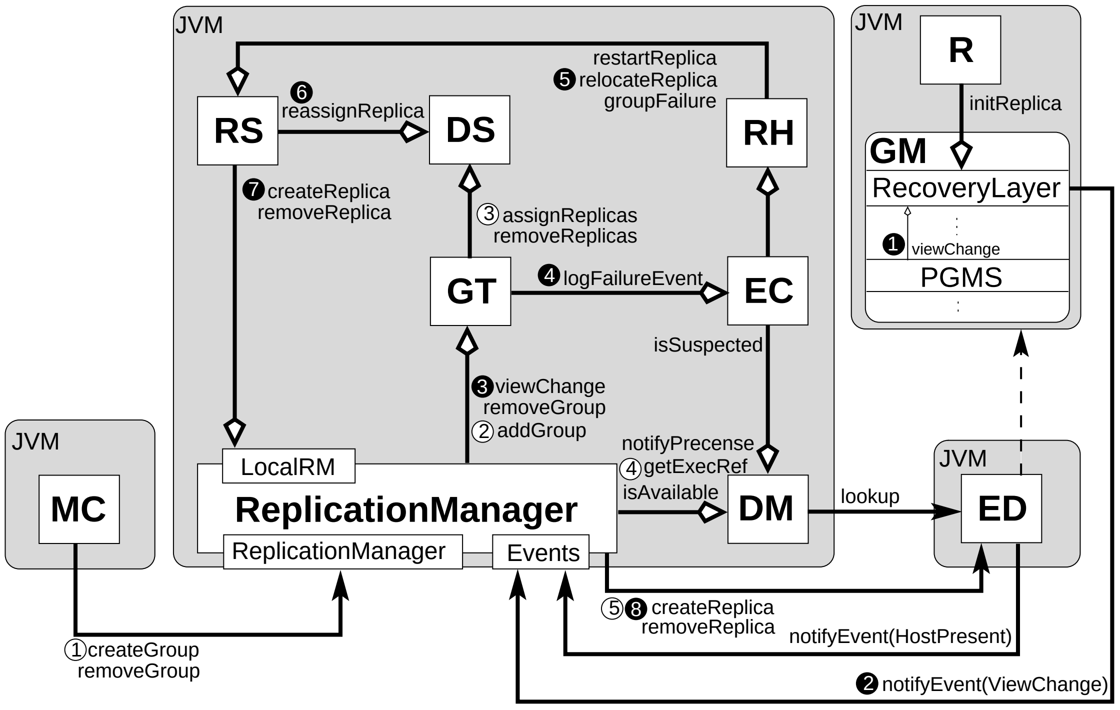

**Figure 3. Overview of interactions between the various components in the ARM framework. Each JVM is executed on a separate host, except for the JVMs containing the Execution Daemon (ED) and the application replica (R).**

```
<Applications>
  <Application name="ReplicationManager" group="1">
    <Class name="jgroup.arm.rm.ReplicaManagerImpl">
      <Argument value=""/>
    </Class>
    <LayerStack order="Dispatcher:Mcast:PGMS:SMS:EGMI:IGMI:Recovery"/>
    <Replication  style="jgroup.arm.styles.Passive" updateDegree="80"/>
    <Recovery  strategy="jgroup.arm.recovery.KeepMinimalInPartition">
      <Redundancy initial="3" minimal="2"/>
    </Recovery>
  </Application>
</Applications>
```

**Figure 4. Example application configuration**

Assume that an object group has been installed in the distributed system, with replicas running on a set of distinct hosts. If at a later time, one of the replicas (or a host) fails, the RecoveryLayer of the Group Manager (GM) associated with each of the replicas will detect this through the installation of a new view (˚ ). The RecoveryLayer will forward this event (¸ ) to the RM, that in turn will notify the GT (Ì ). Events will be logged (˝ ) with the generic *Event Correlator* (EC), and forwarded to the *Recovery Handler* (RH). In this example, RH invokes a *Recovery Strategy* (RS) that decides to create a new replica replacing the one that crashed (˛ ). To determine the host where to create the replica, RS queries DS (ˇ ). Finally, the last two steps (— and Ñ) actually create the new replica on the selected host.

We emphasize that RS, DS and RH are easily extensible and replaceable components, as long as their interfaces are maintained. In the next subsections, the components outlined above are described in more detail.

## 4.1 Application Configuration

Deploying applications using ARM requires specifying a number of application-specific configuration parameters, as illustrated in Figure 4. Specifying application properties through XML files gives us the flexibility to change them without recompilation. In future implementations, it is foreseen that some of these parameters can also be modified at runtime.

The application name and group identifier are used for registering the application with the dependable registry service [16]. The layer stack ordering is given as a colon separated list of service names that are used to construct the appropriate group managers for replicas. The next parameter specifies the desired replication style by naming the class implementing it. Certain well-known styles such as *active replication* or *passive replication* exist as pre-defined ARM classes. For certain replication styles (e.g., passive replication) it is possible to indicate the frequency with which replicas should be brought up to date (`updateDegree`). Finally, each application specifies a recovery strategy to be used in case of failures (`Recovery strategy`). Certain recovery strategies may require specifying initial (`Redundancy initial`) and minimal (`Redundancy minimal`) redundancy requirements for the application. We treat recovery in detail in Section 4.10.

## 4.2 The Replication Manager

The Replication Manager (RM) is the core component of the ARM framework, and provides interfaces for installing, removing application groups, distributing replicas uniformly over the system, and analyzing failure information and recovering from them.

As shown in Figure 1, the RM itself is replicated for fault tolerance, and co-located with replicas of the distributed registry DR. This excludes the possibility that partitions separate RM and DR replicas, which would prevent the system from making progress since RM depends on DR. To ensure consistent behavior of the replicated RM, a semi-active replication [?] scheme is used,

```
    interface LocalRM {
  boolean createReplica(AppInfo app, Host host);
  boolean removeReplica(AppInfo app, Host host);
  DistributionScheme getDistributionScheme();
}
```

Figure 5: Local Replication Manager Interface

```
    interface Events extends ExternalGMIListener {
  void notifyEvent(Event event)
    throws McastRemoteException;
}
```

Figure 6: Event interface for the Replication Manager

```
    interface ReplicationManager extends ExternalGMIListener {
  int createGroup(AppInfo app)
    throws McastRemoteException, GroupExistsException;
  void removeGroup(AppInfo app)
    throws McastRemoteException, UnknownGroupException;
}
```

Figure 7: Management Client access interface for the Replication Manager

```
    interface DistributionScheme {
  HostSet assignReplicas(AppInfo app)
    throws RedundancyException;
  HostSet removeReplicas(AppInfo app)
    throws UnknownGroupException;
  Host reassignReplica(AppInfo app, Host host)
    throws UnknownGroupException, RedundancyException;
}
```

Figure 8: Distribution Scheme Interface

together with a state merging protocol to reconcile divergent states.

As mentioned in Section 2, RM provides an external interface through which object groups can be installed. RM exploits this mechanism to bootstrap itself, making use of the replica distribution scheme to obtain the locations for the RM replicas and then the execution daemon on these hosts to create a local replica. In addition, RM uses the event correlator and recovery strategy to provide self-recovery. This kind of self-recovery requires only minor special handling in the recovery layer to avoid propagation of view change events to RM itself.

### 4.3 Replication Management Interfaces

The replication manager implements two separate external interfaces and a local interface. In the following we describe their usage. The local RM interface, shown in Figure 5 can only be used from within the same JVM as the RM replica itself. In particular, these methods can be used from replaceable RM components such as a recovery strategy implementation (see Section 4.10 for details) to create replicas on the given host. The first external interface, shown in Figure 7, is used by the management client to install and remove application groups within the distributed system, as discussed in Section 4.4.

The second external interface, Events, is used by the execution daemon and recovery layer to notify RM of events needed to make correct decisions, for example concerning recovery. The event notification mechanism currently supports the following event sub-types: ViewChangeEvent and HostPresenceEvent. Their interpretation will become clear in Sections 4.6 and 4.8. Delivery of these events does not require any total ordering guarantees among the RM replicas, making their implementation very simple.

As mentioned in Section 4.2, RM uses a semi-active like replication scheme for rendering it fault-tolerant [**?**]. In this scheme, all RM replicas receive method invocations, but only the *leader* replica actually performs actions that produce output. That is, only a single RM replica will actually create group object replicas on behalf of applications, while the state of *follower* RM replicas are kept synchronized with the leader RM replica. Assuming that only a single management client is connected to the RM, there is no need to use ordering notifications between the leader and followers.

Leader election can be achieved without additional communication, simply by using the total ordering of members defined in the current view. In our implementation, we choose the RM

```
    interface ExecService extends Remote {
  void createReplica(ClassData classData)
      throws RemoteException;
  void removeReplica(ClassData classData)
      throws RemoteException;
}
```

Figure 9: The Execution Service Interface

```
    public class DaemonManager {
  static boolean isAvailable(Host host);
  static boolean isSuspected(Host suspected);
  static ExecService getExecRef(Host host);
  static void notifyPresence(Host host);
}
```

Figure 10: Daemon Manager Class

replica in the last view position (-1) as the leader. If the current leader fails, the RM group will install a new view excluding the current leader, and in effect a follower replica will become the new leader of the group.

### 4.4 Management Client

The management client enables a system administrator to install or remove an application on demand, using the ReplicationManager interface shown in Figure 7. The current implementation of ARM includes a graphical management client tool. There is no reason why the same function could not be performed through a command-line tool or even another application. The AppInfo object is the runtime representation of the application information specified in Figure 4, containing information such as class name, arguments and redundancy requirements for the application to be installed.

Since the management client uses EGMI to install/remove applications, we may in rare circumstances experience multiple invocations of these methods, causing the RM to instantiate the application multiple times. This may occur in situations where RM replicas exist in multiple concurrent views (see Section 3.2.2) and the EGMI client proxy invokes the method on several RM replicas because no reply was received from the first. This is simply a manifestation of the fact that "exactly-once" operation semantics is impossible to guarantee in the presence of failures [20]. The recovery layer, described in Section 4.9, provides measures to deal with this problem.

### 4.5 Execution Daemon

To permit RM create replicas remotely, each host specified in the distributed system configuration (see Figure 2) must deploy an Execution Daemon (ED) implementing the ExecService interface as shown in Figure 9. The ClassData object provided to the ED when creating or removing a replica contains the class name and command-line arguments. These are passed to the main method of the class in question. The ED keeps track of all replicas running on the local host, permitting the RM to remove a given replica. This also makes the create and remove operations idempotent.

The ED implementation publishes its reference using the standard Java RMI [25] registry service, running on a well-known port. The Daemon Manager (DM), described further in Section 4.6, will connect to this registry on each of the hosts in the distributed system, to obtain the ED reference for each host. Having obtained this set of references, the RM can invoke methods to create and remove replicas on all hosts that are running an ED.

Note that ED is dependent neither on Jgroup service (e.g., the dependable registry [16]) nor the replication manager. It can thus be used to create instances of DR and RM replicas during the bootstrap phase. During initialization, however, if ED detects the presence of RM in the distributed system, it will notify RM of itself, enabling new hosts to become part of the distributed system.

Currently, ED will create application replicas in separate JVMs. This has the advantage of isolating misbehaved application replicas so that their damage remains local. An alternative approach is to start several replicas within the same JVM, having the obvious drawback that a single faulty replica could potentially bring down all replicas in that JVM. The scalability of this approach is of course more flexible than the former. Note that neither approach should create replicas in the same JVM as the ED, since a replica failure may halt the JVM, causing ED to

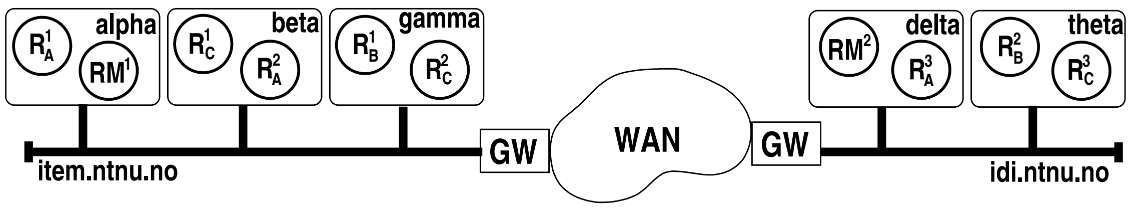

**Figure 11. Example system configuration with replicas $R_A(3)$, $R_B(2)$, $R_C(3)$ and $RM(2)$, where the number in parenthesis denotes the number of replicas. $R_A^i$ denotes the $i^{th}$ replica of type $A$.**

become unavailable.

## 4.6 Daemon Manager

The Daemon Manager (DM) (see Figure 10 for the interface) maintains a table of hosts on which the Execution Daemon (ED) is running. Each RM replica contains a local DM that can be probed to check if a given host is still operational or is suspected to have crashed. DM also keeps a local cache of remote references to the EDs in the system that RM will use when invoking the createReplica (˜ ) method on some ED. ED on hosts that join the distributed system (after DM has been initialized) will notify their presence to RM, which transforms it into an invocation at DM, enabling it to update its local tables. The event correlator contains a *Host Presence Handler* (HPH) which keeps the DM informed about the existence of hosts running a ED. Hence, DM avoids probing for operational hosts continuously, and instead can check availability once a problem is detected.

## 4.7 Replica Distribution Scheme

Figure 11 shows an example system configuration. The replicas have been uniformly distributed over the hosts and domains in such a way that failures can be tolerated in most situations. To facilitate replica placements as shown in Figure 11, a replica distribution scheme must be implemented according to the interface shown in Figure 8. RM uses this interface to assign replicas to a set of hosts on which it can allocate replicas, based on the *initial redundancy* specified in the AppInfo object. This interface enables us to create replicated objects according to application-specific requirements and also to dynamically change the number of replicas of a group as needed.

Various distribution strategies may be applied. In the current version, replicas are distributed evenly among hosts and domains. This will increase application availability in the event of network partitioning when application replicas can provide service continuously in multiple partitions, and state divergences can be resolved through a state merging protocol (cf. Section 3.3).

Although the initial distribution scheme is based on the static content of the system configuration file presented in Section 3.4, the internal tables of the distribution scheme implementation are updated dynamically according to changes in the distributed system. The set of hosts assigned for a given replica will always be disjoint, thus if the total number of hosts available is below the required initial redundancy, the requested replica assignment cannot be satisfied. If the specified redundancy cannot be satisfied, or if all hosts have become unavailable, an exception is thrown.

## 4.8 Event Correlator and Recovery Handler

The event correlator log various events collected through the Events interface, and forward these to their corresponding handlers. In particular, the *Recovery Handler* (RH) analyzes view change events obtained from the RecoveryLayer (¸ ) of group members in the distributed system, and tries to determine what happened. Once the correlation timeout delay associated with RH expires, and given that (a subset of) the logged view change events can be attributed to some failure, RH invokes the application-specific recovery strategy (as specified in Figure 4) that will attempt to rectify the consequence of the failure.

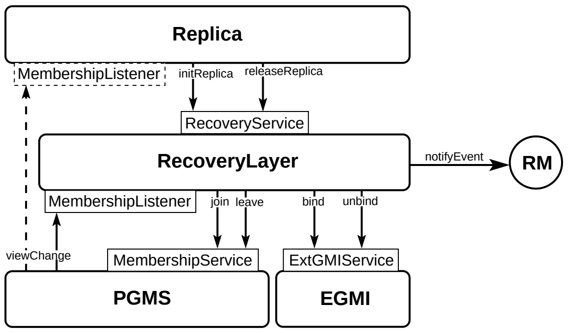

**Figure 12. Recovery Layer**

### 4.9 Recovery Layer

Server replicas that wish to be recoverable through the replication manager must configure their group managers to use the recovery layer. In Figure 12 we illustrate the recovery layer including the interface it provides to replicas and the layers on which it depends.

The recovery layer receives viewChange events on behalf of the replica, and forwards these to RM (using the notifyEvent method also shown in step ¸ of Figure 3). Based on these notifications, the RM determines the need for recovery. To avoid that all members of a group send the same view change to RM, the recovery layer implements a scheme in which a leader is elected using the total ordering of members defined over the current view. The recovery layer elect the first member of the view as the leader. The recovery layer also simplifies replica configuration by requesting to join the object group to which the application belongs, and binding it with the dependable registry. Configuration data such as the group identifier and service name for the application is obtained from the application configuration file, shown in Figure 4. Thus, modifying these parameters can be done without recompiling the application.

The replica can also communicate directly with the PGMS and EGMI layers. For instance, the replica may want to subscribe for viewChange events as shown by the dashed arrow in Figure 12. Also a stateful replica will likely want to use the State Merging Service described in Section 3.3.

As described in Section 4.4, when installing an application group using the management client, two or more RM replicas may in rare circumstances operate in concurrent views, causing the RM to create the same set of application replicas on distinct subparts of the distributed system. When merging from such a rare scenario, the recovery layer of the application will detect any excessive replicas, and after reconciliation of their state, will request the excessive replicas to leave the group. The choice of which replicas to leave the group, is based on their respective position within the view.

### 4.10 Recovery Strategy

It is an objective to allow different quality of service requirements for the various applications, and thus the characteristics provided by a recovery strategy may also differ. Hence, in the ARM framework, the recovery mechanism used, is specific to each application. A template for implementing various recovery schemes, i.e., the recovery strategy interface, is shown in Figure 13. The Recovery Handler (RH), used to correlate view change events, will invoke appropriate methods of the recovery strategy interface in order to provide recovery for the given application (AppInfo).

For instance, KeepMinimalInPartition is an implementation of the recovery strategy interface, with the goal to maintain a *minimum redundancy* level in each partition. When a single replica failure is detected, the restartReplica method is invoked. If the host can be confirmed (through the DM) to still be available, the KeepMinimalInPartition recovery strategy may optionally restart the replica on the same host, i.e., providing a providing fast recovery.

Whenever the RH detect that one or more hosts have become unavailable, it will invoke the

```
interface RecoveryStrategy {
    boolean restartReplica(AppInfo app, Host host);
    boolean relocateReplica(AppInfo app, HostSet hosts);
    boolean groupFailure(AppInfo app, HostSet hosts);
}
```

**Figure 13. Recovery Strategy Interface**

relocateReplica method. Note that relocating a replica means to create a replacement replica on a different host. In, for instance, the KeepMinimalInPartition recovery strategy, (some of) the failed replicas will be reassigned to a different set of hosts, to maintain the minimum redundancy requirement of the affected applications. Comprehensive recovery strategies where performance issues and the failure history of the various hosts is taken into account, is for further study.

A group failure, i.e., the entire group fails, will typically be the consequence of a software design fault. In this case the groupFailure method will be invoked. Group failures are detected when the RH check and find all replicas in an application group to have failed, and that the execution daemon on each of the respective hosts are still available. Recovery should take place also in this case. The common approach is to restart the group and in most cases the condition which activated the logical fault will not reoccur. More advanced approaches involves different replica versions implementing the same specification. For instance, during a live software upgrade, a custom recovery scheme could revert to an old version if a group failure is detected.

## 5 Experimental Results

In this section we present performance and viability measurements for Jgroup and ARM. For Jgroup the measured quantity was the round-trip latency associated with an EGMI invocation-response, averaged over 1000 round trips. For ARM we measured the time needed to detect and recover from a host crash failure with varying correlation delay, averaged over 10 runs.

The application used for our tests consists of a simple echo server, implementing an external interface containing a single method byte[] test(byte[] b) that takes an array of bytes as input and returns the same array as output. Two different versions of the same method are provided, one based on the the anycast semantics, while the other based on the multicast semantics.

The test system was composed of ten machines connected by a 100-Mbs Ethernet network. Each machine contains a 800 MHz Intel Pentium III processor, equipped with 128 MB of RAM. Each machine runs Linux Redhat 7.2; the installed JDK is SUN Microsystems 1.3. Figure 14 shows a comparison between the performance of standard Java RMI and external GMI invocations with anycast semantics. Six different experiments were performed, three related to Java RMI and three related to GMI. In the RMI experiments, a single server is used, while the number of clients varies from one to five. In the GMI experiments, a group of five servers is used, while the number of clients varies in the same way. The results show that when a single client is present, Java RMI is faster than Jgroup GMI; this is motivated by the fact that additional layers are involved in the processing of remote invocations. When the number of clients increase, the performance of Jgroup GMI outperforms Java RMI, as invocations are load balanced among servers. Figure 15 shows the performance of group method invocations with multicast semantics. Here, a group composed of a variable number of replicas (from one to five) is receiving multicast GMI invocations from a single client. The results show that the multicast semantics is more expensive than the unicast one, as expected; this is caused by the additional reliability protocols needed to guarantee view synchrony. Note that the case with a group composed of just one host is special, as there the message is never multicast to other members. This explains by the different performance figures we have obtained.

For our experimental runs with ARM we deployed a single RM replica within the distributed

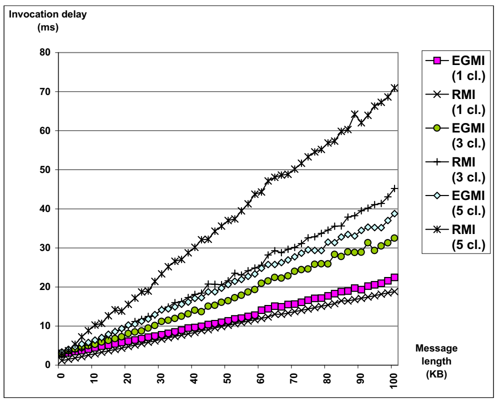

Figure 14: Performance comparison between anycast group method invocations and standard Java RMI

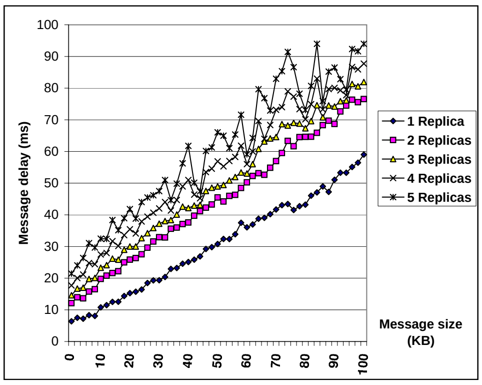

Figure 15: Experimental results for multicast group method invocations and standard Java RMI

**Table 1. Average recovery time (in milliseconds) for the echo server for different values of the correlation delay. Values in parenthesis gives the standard deviation.**

| | Correlation delay | | |
|---|---|---|---|
| Replicas | 10 | 1000 | 4000 |
| 2 | 1985.5 (80) | 2971.7 (66) | 5975.2 (74) |
| 3 | - | 3031.0 (28) | 6166.6 (140) |

system. After RM initialization the echo server was deployed through the management client. For each run we set the initial and minimal redundancy (replicas) to the same value so as to obtain recovery for single crash failures. The results in Table 1, show the recovery time in milliseconds for three different correlation delay values. The recovery time does not include the time needed to detect the failure. We can see from these figures that the actual time to activate a new replica is approximately 2 seconds, most of which stems from the view agreement protocol.

## 6 Related Work

Many other projects have also addressed the problems related to making distributed applications fault tolerant, most of which are based on group communication [8]. Primarily, research efforts [13, 19, 12, 7, 18, 6, 21] have focused on two slightly different distributed object technologies, namely CORBA [23] and Java RMI [25]. Unfortunately, most efforts have been dedicated to CORBA and resulting object group systems may be classified into three categories [12]. The *integration approach* involves modifying and enhancing an object request broker (ORB) using an underlying group communication service. CORBA invocations are passed to the group communication service that multicasts them to replicated servers. This approach was pursued by the Electra system [13]. In the *interception approach*, low-level messages containing CORBA invocations and responses are intercepted on client and server sides and mapped to a group communication service. This approach does not require any modification of the ORB, but relies on OS-specific mechanisms for request interception. Eternal [19] is based on the interception approach. Finally, the *service approach* provides group communication as a separate CORBA service. The ORB is unaware of groups, and the service can be used in any CORBA-compliant implementation. The service approach has been adopted by object group systems such as OGS [12], DOORS [10] and Newtop [18].

Unlike CORBA, the specification of Java RMI enables programmers to implement their own remote references, thus extending the standard behavior of Java RMI. We have exploited this feature by developing the concept of group reference, whose task is to manage interactions between clients and remote object groups. The resulting system provides transparent access to object groups and completely satisfies the specification of Java RMI. Other Java-based middleware systems include Filterfresh [7], JavaGroups [6] and Aroma [21]. Filterfresh shares the same goals as Jgroup: integration of the object group paradigm with the Java distributed object model. Filterfresh is rather limited, as it provides neither external remote method invocations with multicast semantics nor internal remote method invocations among servers. JavaGroups is a message-based group communication toolkit written in Java providing reliable multicast communication. The remote method invocation facility of JavaGroups is not transparent, as it is based on the exchange of objects that encode method invocation descriptions. Aroma provides transparent fault tolerance through a partially OS-specific interception approach similar to that of Eternal, and rely on an underlying group communication service implemented on the native operating system.

What distinguishes Jgroup from existing object group systems is its focus on supporting highly-available applications to be deployed in partitionable environments. Most of the existing object group systems [12, 10, 7, 21] are based on the primary-partition approach and thus cannot be used to develop applications capable of continuing to provide services in multiple partitions. Very few object group systems abandon the primary-partition model [18, 19] but do not provide adequate support for partition-aware application development.

Furthermore, to the best of our knowledge, none of the above systems provide mechanisms similar to those of ARM, to deploy dependable applications in the distributed system without user-level interaction to specify where each of the replicas should be located. However, some systems [19, 10] do provide recovery from crash failures, but do not consider partitioning failures, nor do they allow application-specific recovery strategies.

## 7 Conclusions

In this paper, the design and the implementation of Jgroup and ARM have been presented. Jgroup is an object group system that extends the Java distributed object model. ARM is an autonomous replication management framework that extends Jgroup and enables the automatic deployment of replicated objects in a distributed system according to application dependent policies.

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

Additionally, and unlike most other group communication systems, ARM provides an extensible framework enabling simplified administration of replicated applications, made possible through configurable replication policies. Each application can select their replication policy from the set of available replication styles and recovery strategies, or design custom strategies for particular needs. Each application may be associated with a different set of dependability requirements.

## References

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

[2] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[3] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, Apr. 2001.

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

[5] Ö. Babaoğlu and A. Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proc. of the ACM SIGOPS European Workshop*, pages 612–621, Dagstuhl, Germany, Sept. 1994. Also appears as ACM SIGOPS Operating Systems Review, 29 (1):62–67, January 1995.

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

[7] A. Baratloo, P. E. 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, Apr. 1998.

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

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, Dec. 2001.

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

[11] G. Collson, J. Smalley, and G. 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.

[12] 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, Jan. 1998.

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

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

[15] H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.

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

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

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

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

[20] S. J. Mullender. Interprocess Communication. In S. Mullender, editor, *Distributed Systems*, chapter 9, pages 217–250. Addison-Wesley, second edition, 1994.

[21] N. Narasimhan. *Transparent Fault Tolerance for Java Remote Method Invocation*. PhD thesis, University of California, Santa Barbara, June 2001.

[22] OMG. Fault Tolerant CORBA Using Entity Redundancy. OMG Request for Proposal orbos/98-04-01, Object Management Group, Framingham, MA, Apr. 1998.

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

[24] OMG. Fault Tolerant CORBA Specification. OMG Technical Committee Document ptc/00-04-04, Object Management Group, Framingham, MA, Apr. 2000.

[25] Sun Microsystems, Mountain View, CA. *Java Remote Method Invocation Specification, Rev. 1.7*, Dec. 1999.

[26] Sun Microsystems, Mountain View, CA. *Enterprise JavaBeans Specification, Version 2.0*, Aug. 2001.