ARM: Autonomous Replication Management in Jgroup

Hein Meling
Hein.Meling@item.ntnu.no

Bjarne E. Helvik
Bjarne.E.Helvik@item.ntnu.no

Department of Telematics
Norwegian University of Science and Technology
N-7491 Trondheim, Norway

Abstract

We present the design and implementation of a replication management framework for partition-aware applications based on Jgroup. Jgroup offers an extension to Java RMI based on the group communication paradigm, enabling development of dependable applications in partitionable distributed systems.

The replication management framework simplifies the development of fault tolerant applications by providing exchangeable replica distribution schemes and application specific recovery strategies. The framework is extensible to several replication and recovery strategies. The only user interaction required is the creation and removal of object groups, i.e., *enabling* autonomous replication management.

1 Introduction

Middleware frameworks like CORBA [8] and Java Remote Method Invocation (RMI) [10] have significantly simplified the development of distributed applications. Although the existing middleware frameworks provide communication transparency, they lack multicast primitives and a replication framework, which enables invocation of the same method on multiple server objects (replicas). Object replication is required for providing fault tolerant and highly available services.

A common approach to provide object and process replication is the concept of a *view-oriented Group Communication System (GCS)* [12]. This approach is used by most fault tolerance frameworks, including Horus [11], Totem [5], OGS [2], JavaGroups [1] and Jgroup [4]. A GCS will typically provide a reliable communication service and a group membership service. Unfortunately, most GCSs lack the ability to autonomously distribute replicas on a set of hosts and to recover from replica failures. For instance, keeping a specified redundancy level requires manual intervention, which is undesirable in most cases. Avoiding such manual intervention requires complex recovery protocols, and implementing such protocols is an error-prone task which should not be left to the application developer.

This paper addresses this challenge as it presents the design and a partial implementation of a replication management framework built on top of Jgroup [4]. The replication manager presented here provides a simple interface for a management application for creating object groups in a distributed system. After creation the object group becomes an entity requiring no user interaction, unless manual removal of the group is desired. That is, the replication manager will deal with both replica distribution according to an exchangeable distribution scheme as well as replica recovery, based on a group specific recovery policy. For instance, some object groups have less demanding dependability requirements and may tolerate a weaker recovery policy than other groups. The properties of our replication management framework described above, enables autonomous replication management.

The Eternal [6] and DOORS [7] implementations of the Fault Tolerant CORBA specification [9] provide a similar recovery facility, however, they do not allow application specific recovery policies nor do they autonomously distribute replicas.

Jgroup is a GCS for partitionable environments that integrates the object group paradigm and distributed objects based on Java RMI, allowing clients to interact with object groups by invoking methods on them. Jgroup provides a *Partitionable Group Membership Service (PGMS)*, a *State Merging Service (SMS)* and as mentioned above a *Group Method Invocation (GMI)* service. The latter comes in two flavors, internal and external. The Internal GMI (IGMI) service allows group members to communicate by multicasting and the External GMI (EGMI) allows clients to communi-

cate with a group as a single entity. See [4] for details.

The rest of this paper is structured as follows. Section 2 gives an architectural overview of the replication management framework, while Section 3 describes the core components of the replication manager. Section 4 concludes the paper and outlines some issues for future work.

2 Architectural Overview

Figure 1 shows the core components of our replication management framework and how they relate to the various parts of Jgroup [4]. Our contributions are shown as non-shaded parts, while the shaded parts are Jgroup components [4]. A brief overview is given below. Figure 1 also illustrates the creation of an application replica on host *alpha*. The replication manager (RM) achieves this by invoking the `createReplica` method on the execution daemon (ED) running on *alpha*, which will create a new instance of the replica.

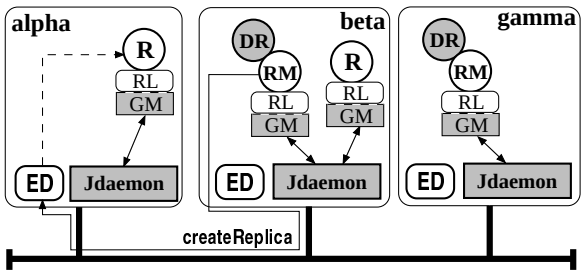

Figure 1. *Replication Management Framework*

The core components are:

- An **Execution Daemon (ED)** must be running on all hosts in the system that should be able to host application replicas. The execution daemon is based on Java RMI and it is used by the replication manager to create replicas on remote hosts.

- **Replication Manager (RM)** is the main component of our system and its tasks include, replica distribution, failure recovery and interaction with client management applications through the replication manager interface.

- **Dependable Registry (DR)** [3, 4] is a replicated registry service included in Jgroup. It enables a dynamic set of replicated remote objects to register themselves under the same name, forming an object group, which can later be retrieved by clients. This enables clients to communicate with the whole group as a single entity.

- A **Group Manager (GM)** [4] provides group communication facilities to server objects. Group managers are organized in layers, and each layer provides some functionality relating to the group. For instance a membership layer is included with Jgroup, providing an interface to the membership service implemented in the Jgroup daemon. The group managers also deals with dispatching external and internal group method invocations. Each group manager is associated with only one server object.

- The **Jgroup daemon (Jdaemon)** [4] implements basic group communication facilities such as failure detection, group membership and multicast. Jdaemons are also organized as layers and they communicate to exchange messages and reachability information.

- **Replication Layer (RL)** is a GM layer, however in Figure 1 it is shown separately. The replication layer supports application replicas by hiding group connection details and recovery information.

- **Application Replica (R)** provides the actual service functionality and must be implemented using the replication layer. An application replica may also want to make use of the state merging GM layer, included with Jgroup for the purpose of state transfer/merging.

2.1 Replication of Critical Services

As shown in Figure 1, the replication manager is replicated for fault tolerance, and thus the RM replicas must keep their internal state consistent through the use of a state merging protocol. In addition, the dependable registry [3] is replicated. The dependable registry is required by the replication layer in order to lookup the replication manager. Thus, the DR replicas should be collocated with the RM replicas. This will avoid the occurrence of partitions that separates the RM and DR replicas, preventing the system to make progress.

2.2 Distributed System Configuration

During initialization the replication manager reads a configuration file that specifies all *hosts* that are able to execute replicas. The configuration file is XML based, and we specify domain and host names separately. See Figure 2 for a simple example of the network shown in Figure 5 on page 5.

In addition, the multicast address used within each domain may be specified, enabling Jgroup to take advantage of the underlying multicast communication facilities provided within each domain. If no multicast address is specified, the default is to send a point-to-point message to all recipients.

```
<DistributedSystem version="1.0">
  <Domain name="item.ntnu.no">
    <Mcast>239.0.0.2</Mcast>
    <Host>alpha</Host>
    <Host>beta</Host>
    <Host>gamma</Host>
  </Domain>
  <Domain name="idi.ntnu.no">
    <Host>delta</Host>
    <Host>theta</Host>
  </Domain>
</DistributedSystem>
```

Figure 2. *Example XML configuration file*

2.3 Execution Daemon

To facilitate the implementation of our replication manager, we deploy a simple and small execution daemon on all hosts specified in the configuration file. The execution daemon make use of the Java RMI registry [10] facility, allowing the replication manager to obtain a reference to each of the execution daemons in the system. Having this set of references, the replication manager can invoke methods to create and remove replicas on the hosts running the execution daemon.

Note that the execution daemon is neither dependent on any part of Jgroup, *e.g.*, the dependable registry [3], nor the replication manager. It can thus be used to create instances of DR and RM replicas during the bootstrap phase.

In the current implementation, the execution daemon will create replicas in separate Java Virtual Machines (JVMs). This has the advantage that if a replica causes the JVM to crash, the execution daemon will still be available and thus be able to create new replicas.

An alternative approach is to start several replicas in the same JVM, but this has the drawback that a single faulty replica will bring down all replicas in that JVM. The scalability of this approach is of course more flexible than the former.

2.4 Replication Layer

Figure 3 shows the default layer composition and the interfaces supported by the Replication Layer. A replica only has to invoke the `initReplica` method on the replication layer in order to become a member of its group. To compute the correct group identifier for the replica, the replica class name and command line arguments must be passed to the `initReplica` method. The replication layer deals with invoking the `join` method on the PGMS, assigning the correct group identifier, and binding this replica instance with the dependable registry through the EGMI layer. The PGMS and EGMI layers are parts of the Group Manager shown in Figure 1.

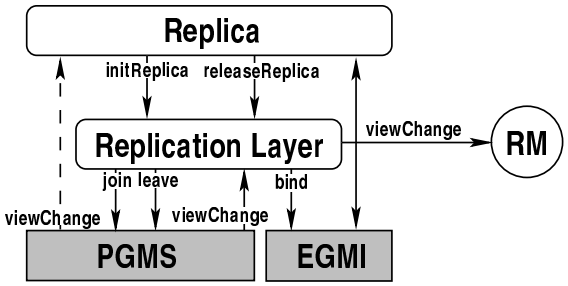

Figure 3. *Replication Layer*

The replication layer receives `viewChanges` on behalf of the replica, and will forward these to the replication manager, who will determine the need for recovery based on the `viewChanges` it receives. This is also shown in Figure 7 on page 6.

The replica can also communicate directly with the PGMS and EGMI layers. For instance, the replica may want to subscribe for `viewChanges` as shown by the dashed arrow in Figure 3. Also a replica will likely want to use the State Merging Service provided with Jgroup [4].

3 The Replication Manager

The core components of the replication manager includes:

- Distribution Scheme
- Correlator
- Recovery Service

These comprise the replication manager and are accessed through the replication manager interface. These components are discussed in the following subsections.

3.1 Replication Management Interface

The replication manager implements a simple external interface, as shown in Figure 4(a). This interface enables a management client to create and remove groups from the system. The `createGroup` method is invoked using the *anycast* semantics of the Jgroup EGMI service. That is, only a single RM replica will actually perform the `createReplica` method on the selected execution daemons, as illustrated in Figure 1 on page 2. However, the internal tables of all the RM replicas will be updated through an internal *multicast* method specified in the internal interface in Figure 4(b).

```
interface ReplicationManager
    extends ExternalGMIListener
{
    int createGroup(ReplicaData rd)
    void removeGroup(int gid)
}
```

(a) External interface

```
interface InternalRM
    extends InternalGMIListener
{
    boolean updGroup(ReplicaData rd)
}
```

(b) Internal interface

Figure 4. *Replication Management Interfaces*

The `ReplicaData` argument passed to the `createGroup` method contains information required by the execution daemon to create a replica instance. In addition, it includes information about replication and recovery style, initial and minimal redundancy for the group. This information is stored in the RM and will be used by the recovery service, as discussed further in Section 3.4. The replication style parameter will include both active and passive replication, however, the current implementation only supports active replication.

3.2 Replica Distribution Scheme

Figure 5 on page 5 shows an example system configuration. The replicas are uniformly distributed on the hosts and domains in such a way that failures can be tolerated in most situations.

To facilitate such replica placements, a replica distribution scheme must be implemented according to the interface shown in Figure 6. The replication manager

```
interface DistributionScheme
{
    String[] getHosts(ReplicaData rd)
}
```

Figure 6. *Distribution Scheme Interface*

uses this interface to get a set of hosts on which it can allocate replicas, based on the specified *redundancy* in the `ReplicaData` parameter. This enables us to create redundant objects according to application specific requirements on a per object group basis and also to dynamically change the redundancy of a group as needed.

Since the RM uses this interface, it enables easy replacement of the replica distribution scheme, thus providing different implementations according to the need of a system. For instance, a load monitoring based distribution scheme could assign replicas to the hosts with lowest system load.

Our current distribution scheme implementation takes a different approach. We only keep track of available hosts and also the number of replicas assigned to each host. Replicas of the same type will be placed on separate domains if enough domains are available for the specified redundancy. This is to increase availability in the event of network partitioning. Note that we assume that application replicas can deliver the service continuously in multiple partitions, and that any state divergence can be resolved through a state merging protocol, possibly supported by the Jgroup State Merging Service [4].

Although the initial distribution scheme is based on the static content of the system configuration file presented in Section 2.2, the internal tables are updated dynamically according to changes in the distributed system. In fact, for each invocation of the `getHosts` method we check if the selected hosts are available, and find alternative hosts if any hosts have become unavailable, before returning the list to the replication manager. The list of selected hosts will be disjoint, thus if the total number of hosts available is below the required threshold, we cannot satisfy the requested redundancy. If the specified redundancy cannot be satisfied, or if all hosts have become unavailable, an exception is thrown.

This strategy leaves only a small interval δ during which the replication manager might fail to create a replica, due to an unavailable host. If replica creation failed on one or more of the selected hosts, the system tables will be updated by marking the unavailable hosts. Once the system tables have been updated we reinvoke the `getHosts` method with the remaining redundancy.

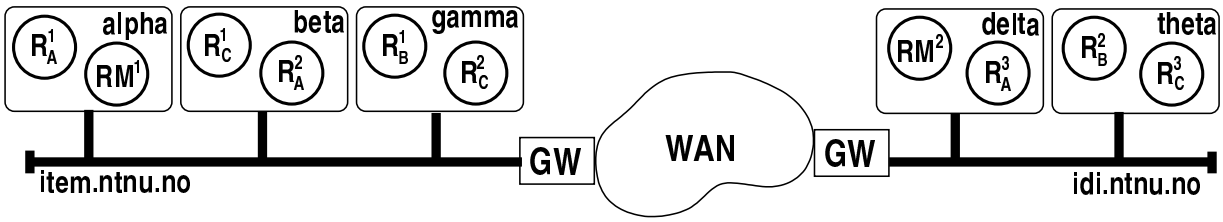

Figure 5. *Example system configuration with replicas $R_A(3)$, $R_B(2)$, $R_C(3)$ and $RM(2)$, where the number in parenthesis denotes the number of replicas. R_A^i denotes the i^{th} replica of type A.*

3.3 Correlator

The correlator collects view changes from all groups allocated by the replication manager and analyzes these to determine what happened in the system. It updates the internal group tables with this information, and invokes the recovery service, which will attempt to rectify the consequence of the failure.

In the current design (as illustrated in Figure 7 on page 6) we have placed the correlator function in the replication manager. This solution works well for a small number of failures, however it does not scale very well, since the replication manager will potentially receive a lot of view changes, for instance when a network partitioning occurs in a system with a large number of groups configured with replicas in separate domains. This will cause a burst of view changes to be forwarded to the replication manager (or correlator), and it may become a bottleneck, and thus increases the recovery time.

In order to reduce the number of view changes that needs to be propagated to the correlator, the replication layer can be used to determine the replica that will forward the view change to the correlator. For instance, the first member of the view may be the only replica to forward its view change to the correlator. This can be achieved without any agreement protocol, since each group member have already agreed to the current view.

3.4 Recovery Service

The recovery service will be invoked by the correlator when required. It will examine the group tables to determine type of recovery that is required for the affected groups and will invoke the correct recovery style. That is, each group can specify one of several recovery styles to be used based on application specific requirements.

Our current design only supports two recovery styles, but new recovery styles are easy to add. The primary recovery style will try to maintain a *minimum* redundancy level in each partition. If no replicas are available in a given partition we do not create a new one within that partition, since no state information is available for state transfer. Also, in the event that a large number of partitions occur, the group may become clogged with a large number of replicas upon a merge. The latter choice is of course application specific, and alternative recovery strategies may be used.

Let $\langle Ei, Ri, RLi \rangle$ denote a set of entities executing on host i. Figure 7 shows a sequence chart, illustrating the initial creation of two replicas on hosts 1 and 2. When host 1 crashes, as indicated by the $\times$ symbol, a `viewChange` (view ②) is propagated to the RM. This causes the replication manager to create a replacement replica on host 3.

Although both the recovery service and correlator are at work here they are not shown in the figure. The `viewChange` messages occurs as a consequence of a `viewChange` from the PGMS, as indicated by the view identifiers in the circles. Initially, hosts 1 and 2 belong to the view ①, and after the failure and recovery, hosts 2 and 3 belong to view ③. View ② is an intermediate view with host 2 only.

4 Conclusions and Future Work

The autonomous replication management framework presented in this paper enables automatic deployment of replicated objects in a distributed system according to the dependability policies of the service/system, and to recover from replica failures seamlessly. The replica distribution scheme is easily exchangeable, given that the distribution scheme interface is implemented. In addition, it allows recovery strategies, which may be specific to each group in the system.

The current implementation supports the basic features of group creation/removal and also the distribution scheme is complete. The focus of our current work is on the correlator and recovery service.

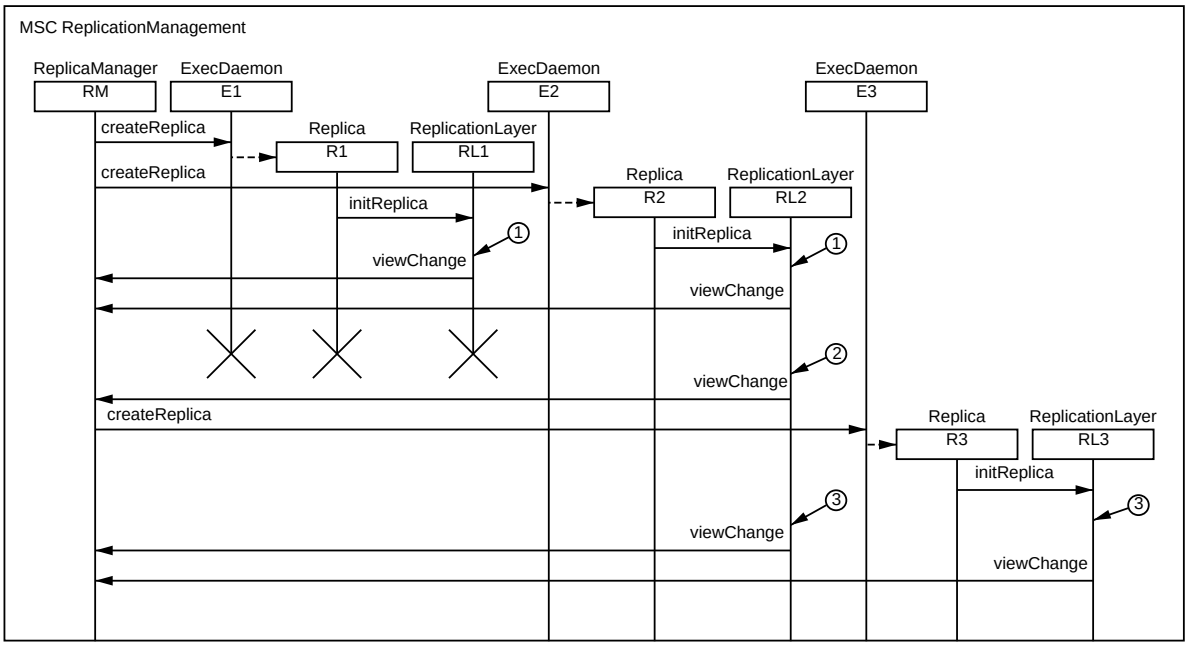

Figure 7. *Sequence chart depicting replica recovery*

Acknowledgements. We are grateful to Alberto Montresor and Frank Li for comments and suggestions on an early draft of this paper.

References

[1] B. Ban. JavaGroups User's Guide. Technical report, Cornell University, Aug. 1999.

[2] 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, 1998.

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

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

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

[6] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A Fault Tolerance Framework for CORBA. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, pages 150–157, Madison, WI, June 1999.

[7] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Second International Symposium, Distributed Objects & Applications (DOA 2000)*, pages 39–48, Antwerp, Belgium, Sept. 2000.

[8] Object Management Group. *The Object Request Broker: Architecture and Specification, Rev. 2.2*, Mar. 1998.

[9] Object Management Group. *Fault Tolerant CORBA Specification, V1.0*, Apr. 2000.

[10] Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.7*, Dec. 1999.

[11] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[12] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report MIT-LCS-TR-790, MIT, Sept. 1999.