SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management

Hein Meling[1,*], Alberto Montresor[2], Bjarne E. Helvik[3] and Ozalp Babaoglu[4]

[1] *Department of Electrical Engineering and Computer Science, University of Stavanger, 4036 Stavanger, Norway. Email: hein.meling@uis.no*
[2] *Department of Information and Communication Technology, University of Trento, via Sommarive 14, 38050 Povo, Italy. Email: alberto.montresor@dit.unitn.it*
[3] *Centre for Quantifiable Quality of Service in Communication Systems (Q2S), Norwegian University of Science and Technology, O.S. Bragstadsplass 2E, 7491 Trondheim, Norway. Email: bjarne@q2s.ntnu.no*
[4] *Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy. Email: babaoglu@cs.unibo.it*

SUMMARY

This paper presents the design and implementation of Jgroup/ARM, a distributed object group platform with autonomous replication management, along with a novel measurement-based assessment technique which is used to validate the fault handling capability of Jgroup/ARM.

Jgroup extends Java RMI through the group communication paradigm and has been designed specifically for application support in partitionable systems. ARM aims to improve the dependability characteristics of systems through a fault treatment mechanism. Hence, ARM focuses on deployment and operational aspects, where the gain in terms of improved dependability is likely to be the greatest. The main objective of ARM is to localize failures and to reconfigure the system according to application-specific dependability requirements. Combining Jgroup and ARM can significantly reduce the effort necessary for developing, deploying and managing dependable, partition-aware applications.

Jgroup/ARM is evaluated experimentally to validate its fault handling capability; the recovery performance of a system deployed in a wide area network is evaluated. In this experiment multiple nearly-coincident reachability changes are injected to emulate network partitions separating the service replicas. The results show that Jgroup/ARM is able to recover applications to their initial state in several realistic failure scenarios, including multiple, concurrent network partitionings.

KEY WORDS: Fault Tolerance, Fault Treatment, Replication and Recovery Management, Measurement-based Assessment, Middleware, Remote Method Invocation, Group Communication.

*Correspondence to: Department of Electrical Engineering and Computer Science, University of Stavanger, 4036 Stavanger, Norway. Email: hein.meling@uis.no

Received
Revised

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. Distributing replicas geographically is often effective for rendering failures independent. However, the nodes and the network that interconnects them are hardly static. Nodes may fail, they may be removed or inserted, and network partitionings may occur, with nodes temporarily divided into isolated subsets unable to communicate with each other. Providing highly dependable services cost-effectively in such an environment requires autonomous failure management and replication management. The objective of this paper is to present a system meeting the above requirements and to validate its fault-management performance.

Distributed object-based middleware platforms such as CORBA [45] and J2EE [51] hold the promise of simplifying the complexity of networked applications and their development effort. Their ability to exploit commercial off-the-shelf components, to cope with heterogeneity and to integrate legacy systems makes them particularly attractive for building application servers and e-business solutions. Yet, they remain unsuitable for implementing replication since the required “one-to-many” interaction model has to be simulated through multiple one-to-one interactions. This not only increases application complexity, but also degrades performance. This shortcoming has been recognized by numerous academic research projects [17, 5, 42], and also by the Object Management Group [44]. In the various proposals, distributed objects are replaced by their natural extension *distributed object groups* [12]. Clients interact with a server’s object group transparently through remote method invocations (RMI), as if it were a single, non-replicated remote server object. Global consistency of the object group (visible through results of RMI) is typically guaranteed through a *group communication system* (GCS) [11].

This paper describes the design and implementation of *Jgroup/ARM*, a novel middleware platform based on object groups for developing, deploying and operating distributed applications with strict dependability requirements. A thorough experimental assessment of Jgroup/ARM is provided, validating its fault handling capability when exposed to complex failure scenarios. We also report on an experience with using Jgroup to provide fault tolerant transactions.

Jgroup [37] is a group communication service that integrates the Java RMI distributed object model with object groups. Apart from “standard” group communication facilities, Jgroup includes several features that make it suitable for developing modern networked applications. Firstly, Jgroup supports *partition-awareness*: replicas living in disjoint network partitions are informed about the current state of the system, and may take appropriate actions to ensure the availability of the provided service in spite of the partitioning. A network partition occur when failures render communication between subsets of nodes impossible. By supporting partitioned operation, Jgroup trades consistency for availability, whereas other systems take a *primary partition* approach [11], ensuring consistency by allowing only a single partition to make progress. A number of application areas [3, 56, 13] stand to benefit from trading consistency for availability in partitioned group communication, e.g. a replicated naming service may operate in disconnected partitions without compromising consistency when merging the partitions [36]. A *state merging service* is provided to simplify the re-establishment of a consistent global state when partitions merge. Jgroup is unique in providing a uniform object-oriented programming interface (based on RMI) to govern *all* object interactions both within an object group as well as interactions with clients.

The *Autonomous Replication Management* (ARM) framework [32, 31] extends Jgroup with automated mechanisms for performing management activities such as distributing replicas among nodes and recovering from replica failures, thus reducing the need for human interactions. These mechanisms are essential to operate a system with strict dependability requirements, and are largely missing from existing group communication systems [17, 5]. ARM achieves its goal through three core paradigms: *policy-based management* [49], where application-specific replication policies along with a system-wide distribution policy are used to enforce the service dependability requirements; *self-healing* [39], where failure scenarios are discovered and handled through recovery actions with the objective to minimize the period of reduced failure resilience; *self-configuration* [39], where objects are relocated/removed to adapt to uncontrolled changes such as failure/merge scenarios, or controlled changes such as scheduled maintenance (e.g. OS upgrades), as well as software upgrade management [50]. For brevity, this latter issue is not covered in the paper. Our approach is based on a non-intrusive system design, where the operation of deployed services is completely decoupled from ARM during normal operation. Once a service has been installed, it becomes an "autonomous" entity, monitored by ARM until explicitly removed. This design principle is essential to support a large number of object groups.

The Jgroup/ARM framework shares many of its goals with other fault tolerance frameworks, notably Delta-4 [46], AQuA [47] and FT CORBA [44]. The novel features of Jgroup/ARM when compared to other frameworks include: autonomous management facility based on policies, support for partition awareness, and interactions based solely on RMI.

Paper organization: Section 2 introduces the system model and gives an architectural overview of Jgroup/ARM. In Section 3, we describe the Jgroup distributed object model and provide an informal specification. In Section 4, we describe the ARM framework. Section 5 presents the experimental evaluation of Jgroup/ARM when exposed to network failures. Section 6 presents experience with developing with Jgroup, while Section 7 compares Jgroup and ARM with related work. Section 8 concludes the paper.

2. Jgroup/ARM Overview

The context of this work is a distributed system comprising a collection of nodes connected through a network and hosting a set of client and server *objects*. The system is *asynchronous* in the sense that neither the computational speed of objects nor communication delays are assumed to be bounded. Furthermore, the system is unreliable and failures may cause objects to *crash*, whereby they simply stop functioning. Once failures are repaired, objects may return to being *operational* after an appropriate *recovery* action. Byzantine failures are not covered [27]. Communication channels may omit to deliver messages; a specific communication substrate is included in Jgroup to retransmit messages, also using alternative routes [37]. Long-lasting *partitionings* may also occur, in which certain communication failure scenarios may disrupt communication between multiple sets of objects forming *partitions*. Objects within the same partition can communicate among themselves, but cannot communicate with objects in other partitions. When communication between partitions is re-established, 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/ARM has been designed to simplify the development and operation of partition-aware, dependable applications by abstracting complex system events such as failures, recoveries, partitions, merges and asynchrony into simpler, high-level abstractions with well-defined semantics.

Jgroup is a middleware framework aimed at supporting dependable application development through replication, based on the *object group* paradigm [12, 30]. In this paradigm, a set of server objects (or *replicas*) form a group to coordinate their activities and appear to clients as a single server. The current implementation can also be used as a basis for a multi-tier architecture, where one server group acts as a client towards another server group, following the same approach as in Eternal [41]. The core facilities of Jgroup [37] include a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS).

The task of the PGMS is to provide replicas with a consistent view of the group's current membership, to be used to coordinate their actions. Reliable communication between clients and groups is handled by the GMIS and is based on the concept of a *group method invocation* (GMI) [37]. GMIs result in methods being executed by the replicas forming the group. To clients, GMIs are indistinguishable from standard RMI: clients interact with the object group through a *client-side group proxy* that acts as a representative object for the group, hiding its composition. The group proxy maintains information about the replicas composing the group, and handles invocations on behalf of clients by establishing communication with one or more replicas and returning the result to the invoking client. On the server side, the GMIS enforces reliable communication among replicas. Finally, the task of SMS is to support developers in re-establishing a consistent global state when two or more partitions merge by handling state diffusion to other partitions.

Jgroup also includes a mechanism for locating object groups. The standard Java RMI registry [53] is not suitable for this purpose: it cannot be replicated and it does not support the binding of multiple server references to a single service name. The Jgroup *dependable registry* (DR) solves these issues and is designed as a drop-in replacement for the RMI registry.

The ARM framework supports seamless deployment and operation of dependable services. The set of nodes that may host applications and ARM-specific services is called the *target environment*; within it, issues related to service deployment, replica distribution and recovery from failures are autonomically managed by ARM, following the rules of user-specified distribution and replication policies. Maintaining a fixed redundancy level is a typical requirement specified in the replication policy. Fig. 1 illustrates the core components and interfaces supported by the ARM framework: a system-wide replication manager (RM), a supervision module associated with each of the managed replicas, an object factory deployed at each of the nodes in the target environment, and an external management client used to interact with the RM.

The *replication manager* is the main component of ARM; it is implemented as a distributed service replicated using Jgroup. Its task is to keep track of deployed services in order to collect and analyze failure information, and reconfigure the system on-demand according to the configured policies. The *supervision module* is the ARM agent co-located with each Jgroup replica which is responsible for forwarding view change events generated by the PGMS to the RM. It is also responsible for decentralized removal of excessive replicas. The purpose of *object factories* is mainly to act as bootstrap agents; they enable the RM to install and remove replicas, as well as to respond to queries about which replicas are hosted on the node. The *management client* provides system administrators with a management interface, enabling them to install and remove dependable applications in the

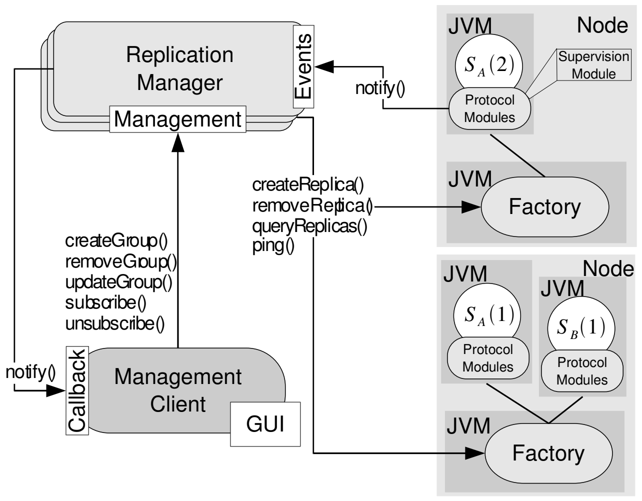


Figure 1. Overview of ARM components and interfaces.

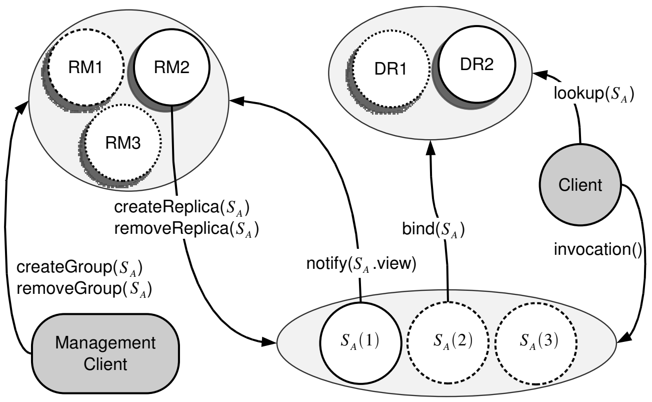


Figure 2. The Jgroup/ARM architecture.

system and to specify and update the distribution and replication policies to be used. It can also be used to obtain monitoring information about running services. Overall, the interactions among these components enable the RM to make proper recovery decisions, and allocate replicas to suitable nodes in the target environment.

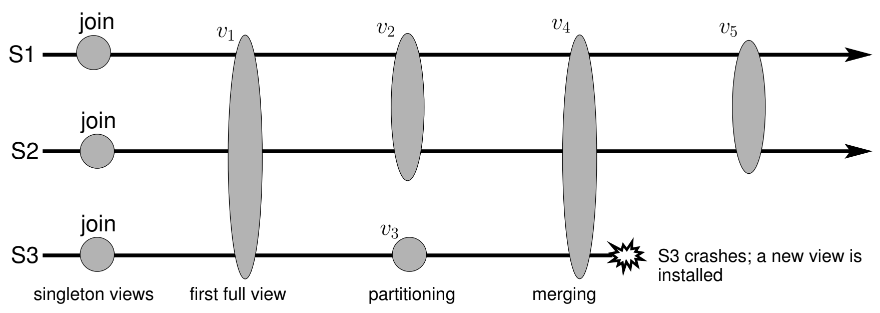


Figure 3. PGMS behavior. Servers $S1$, $S2$ and $S3$ join the group, forming view v_1; immediately after, servers $S1$ and $S2$ are partitioned from $S3$. The PGMS reacts by installing two views v_2 and v_3. Later, the partitioning disappears, and nodes are enabled to form again a view v_4 including all members. Finally, server $S3$ crashes, causing view v_5 to be installed, including only the surviving members.

Next, we briefly describe a minimal Jgroup/ARM deployment, as shown in Fig. 2. Three different groups are present. The DR and RM services are the main infrastructure components and are both required in all Jgroup/ARM deployments. The figure also illustrates a service labeled S_A that is implemented as a simple object group managed through Jgroup/ARM. Finally, two clients are shown: one client interacts with the S_A object group, while the other is the management client used to interact with the RM. Object factories are not shown, but are present at each node. The main communication patterns are shown as graph edges. For example, object groups must notify the RM about changes in the current group composition; the RM may respond to such changes through recovery actions, as defined in the replication policy. When joining the system, replicas must bind themselves to the same name in the dependable registry, to be looked up later by clients. After obtaining references to object groups, clients may perform remote invocations on them. The object group reference hides the group composition from the client.

3. The Jgroup Middleware

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 it [37]. In the following, only short informal descriptions are provided.

3.1. The Partition-aware Group Membership Service

An object group is a collection of server objects that cooperate in providing a distributed service. For increased flexibility, the group composition is allowed to vary dynamically as new servers are added and existing ones are removed. A server contributing to a distributed service becomes *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. System asynchrony and failures may cause each member to have a different perception of the group's current membership. The task of the PGMS is to track voluntary variations in the membership, as well as involuntary changes due to failures and repairs of servers and communication links. All membership changes are reported to members (server objects) through the *installation* of *views*. A view consists of a membership list and a unique view identifier, and corresponds 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 [2] for a detailed discussion). 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, it is required that a view be installed only after agreement is reached on its composition among the servers included in the view. Finally, the 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 admits coexistence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications. Fig. 3 shows an example of the PGMS behavior caused by partition, merge and crash failure events.

3.2. The Group Method Invocation Service

Jgroup differs from existing object group systems due to its uniform communication interface based entirely on GMIs: 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 transparent, as much as possible, to standard RMI. Clients should not be required to be aware that they are invoking a method on a group of servers instead of a single one. On the other hand, servers have different requirements for group invocations, such as obtaining a result from each server in the current view.
- *Efficiency:* Having identical specifications for EGMI and IGMI would have required that clients become members of the group, resulting in poor system scalability. Therefore, Jgroup follow the open group model [24], allowing EGMI to have slightly weaker semantics than those of IGMI. 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 [4] (See Section 3.2.2 for an example).

When developing a dependable distributed service, methods are subdivided into an *internal remote interface* (containing methods to be invoked through the IGMI service) and an *external remote interface*

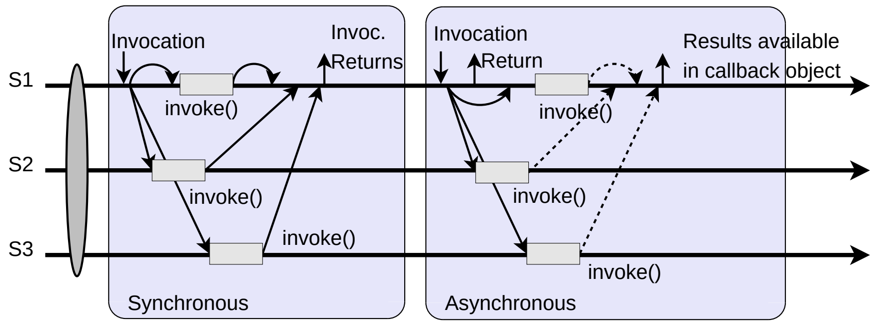


Figure 4. In synchronous IGMI, the invoking server is delayed until all servers have returned a result (or an exception). In asynchronous IGMI the invocation returns immediately, whereas result values must be obtained later through a callback object.

(containing methods to be invoked through the EGMI service). Group method invocations are handled by *proxy objects*, whose task is to enable clients and servers to communicate with the entire group using only local invocations. A proxy object provides the same interface as the distributed service that it represents, and translates GMIs into remote communications with the members of the group.

Proxy objects for IGMI and EGMI are obtained differently. While IGMI proxies are located on the server side and are provided by the local Jgroup runtime, EGMI proxies are located on the client-side and are obtained from a *registry service*. The registry enables multiple servers to register themselves under the same identifier, so as to compose a group. Clients perform lookup operations on the registry by specifying the identifier of the desired service. Jgroup features a *dependable registry* (DR) [36] that is a replicated version of the Java RMI registry.

In the following, we discuss the details of IGMI and EGMI, and how internal invocations substitute message multicasting as the primary communication paradigm. In particular, we describe the reliability guarantees provided by the two GMI implementations. They are derived from similar properties previously defined for message delivery in message-based group communication systems [2]. In this context, we say that an object (client or server) *performs* a method invocation at the time it invokes a method on a group; we say that a server object *completes* an invocation when it terminates executing the associated method.

3.2.1. Internal Group Method Invocations

IGMIs are performed by a server group member on the group itself, and unlike traditional RMI, IGMI returns an array of results rather than a single value. IGMI come in two flavors: *synchronous* and *asynchronous*, shown in Fig. 4. For synchronous IGMI, the invoking server remains blocked until an array containing the result from each server that completed the invocation can be assembled and returned to the invoker. 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 returned its result. Therefore, in asynchronous IGMI the invoking server does not block, but instead specifies a *callback* object that will be notified when return values are ready from servers completing the invocation.

The servers forming a group that completes an IGMI is said to satisfy a variant of "view synchrony", which has proven to be an important property for reasoning about reliability in message-based systems [7]. Informally, view synchrony requires two servers that install the same pair of consecutive views to complete the same set of IGMIs 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 IGMIs they have completed in the current view. The view synchrony property enables a server to reason about the state of other servers in the group using only local information such as the history of installed views and the set of completed IGMIs. Note that view synchrony, by itself, does not require invocations to be ordered; appropriate ordering semantics may be obtained using additional protocols.

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, or with an application-specific exception (i.e., exceptions associated with the actual code executed, such as an illegal argument). Furthermore, if an operational server S completes an IGMI in a view, all servers included in that view will also complete the same invocation, or S will install a new view excluding those servers that are not able to complete it. Since installed views represent the current failure scenario as perceived by the 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. Hence, IGMI guarantees that all result values have been computed during the same view.

3.2.2. External Group Method Invocations

The external interface contains the set of methods that can be invoked by clients. Jgroup supports four distinct protocol types for EGMI, illustrated in Fig. 5. Invocations based on the *anycast protocol* are completed by at least one server of the group, unless there are no operational servers in the client's partition. This protocol is suitable for implementing methods that do not modify the shared group state, as in query requests to interrogate a database. Invocations based on the *multicast protocol* are completed by every server of the group that is in the same partition as the client. Multicast invocations are suitable for implementing methods that are commutative and may update the shared group state. The *atomic protocol* adds total ordering to the multicast protocol. Total ordering is implemented according to the ISIS algorithm [8]. Both atomic and multicast protocols require that methods have deterministic behavior. The *leadercast protocol* enables a client to invoke only the group *leader*, whose state will be communicated to the other servers in the group before returning a reply to the client. It is suitable for methods that update the shared group state, yet may exhibit non-deterministic behavior. The leader is selected deterministically based on the current view, e.g., the first member in the view.

In many fault-tolerant systems, different replication styles are supported at the *class* level [42, 47], meaning that all the methods of a particular implementation must follow the same replication style. Jgroup takes a different approach: when implementing an external interface, the developer may specify per-method invocation protocol using Java annotations. For example, `@Leadercast public void foo()` specifies a method `foo()` using the leadercast protocol. This allows for maximal flexibility,

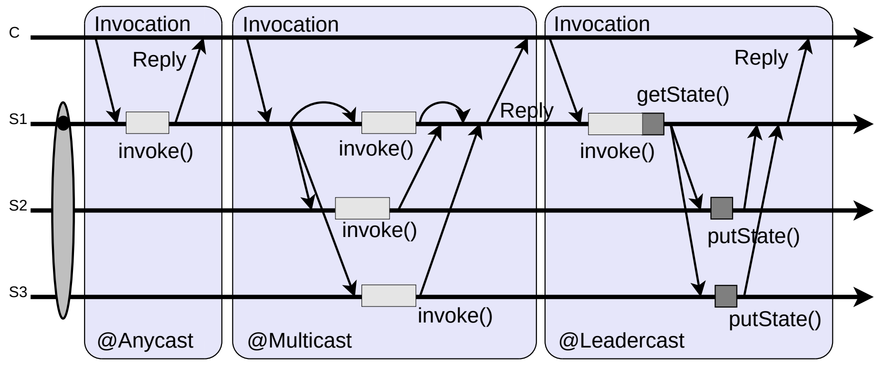


Figure 5. The supported EGMI method invocation protocols. The atomic protocol is not shown, as it is a specialization of multicast.

as developers may select the appropriate invocation protocol at the method level, and even provide different implementations with alternative protocols. Note however, that Jgroup does not perform any checks on the semantics of individual methods, e.g., if two methods are co-dependent, the developer must select a consistent protocol for all co-dependent methods. Per method replication protocols were also proposed in [16].

Our implementation of Jgroup guarantees that EGMI is *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 all versions of EGMI. In the case of multicast, leadercast and atomic EGMI, Jgroup also guarantees view synchrony as defined in the previous section. IGMI and EGMI 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-side group proxy 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 size would no longer be determined by the number of servers (degree of replication), but by the number of clients, which could be very large.

One of the goals of Jgroup has been transparent server replication from the client's point of view. This requires that from a programmer's perspective, EGMI should be indistinguishable from traditional RMI. This has ruled out consideration of alternative definitions for EGMI including multi-value results or asynchronous invocations.

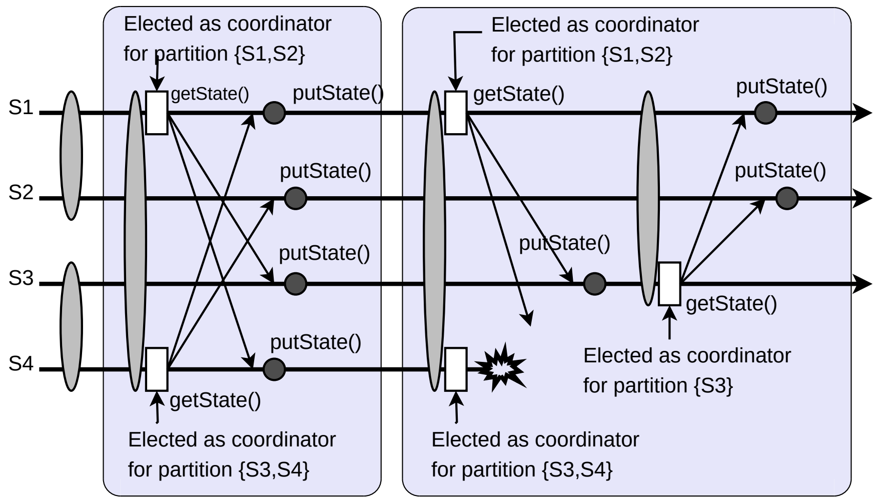

Elected as coordinator for partition {S1,S2}
getState()
putState()
S1
S2
S3
S4
putState()
putState()
putState()
getState()
Elected as coordinator for partition {S3,S4}
Elected as coordinator for partition {S1,S2}
getState()
putState()
putState()
putState()
getState()
Elected as coordinator for partition {S3}
Elected as coordinator for partition {S3,S4}

Figure 6. Two runs of the state merge algorithm: (1) partitions merge with no failures; (2) partitions merge while a coordinator fails.

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 systems and is independent of any design choice we have made for Jgroup. Being based on a PGMS, Jgroup admits partition-aware applications that are able to cope with multiple concurrent views. During partitioning, application semantics dictate which of its services remain available. When failures are repaired and partitions merge, a new global state has to be constructed, to reconcile, to the extent possible, any divergence that may have taken place during partitioned operation.

Generally, 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 completely automate state reconciliation for arbitrary applications, a lot can be accomplished at the system level for simplifying the task [37]. Jgroup includes a state merging service (SMS) that provides support for building application-specific reconciliation protocols based on a predefined interaction model. 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 own 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 [3, 36].

Fig. 6 illustrates two runs of the state merge algorithm. The first is failure-free; $S1$ and $S4$ are elected as coordinators for their respective partitions, and successfully transfer their state. The second case shows the behavior of the state merge in the event of a coordinator crash ($S4$). In this case, the PGMS will detect the crash, and eventually install a new view. This will be detected by the SMS, that will elect a new coordinator for the new partition, and finally complete the state merge algorithm. 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: (1) 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; (2) 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. This is intrinsic to partition-awareness, and is not a limitation of SMS.

The complete specification and implementation of SMS are given in [37]. Here we 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. Implementation Details

Each replica is associated with one *group manager* for each group that the replica has joined. The group manager represents the Jgroup runtime and is composed of a set of *protocol modules* implementing the services described above, in addition to other internal services (such as failure detection, message multicasting, etc.). Protocol modules may interact with (1) other local modules, (2) a corresponding remote module within the same group, and (3) external entities such as clients. Local interactions, such as the EGMI cooperating with the PGMS to enforce view synchrony, are governed through internal service interfaces; each module *provides* a set of services, and *requires* a set of services to work. The set of protocol modules to be used can be declaratively specified at deployment time. This allows maximal flexibility in activating required services. The module configuration is integrated into the ARM policy management, as described in Section 4.3.

The algorithms implementing the PGMS and the basic multicast communication facilities are discussed in a previous paper [2]. Herein we provide a few highlights of their main features, followed by a brief description of the modules implementing the GMI service and the dependable registry.

The agreement algorithm proposed in [2] is based on the rotating coordinator scheme, with the coordinator driving three communication rounds. An *eventually perfect* failure detector [10] (FD) is employed, whose definition has been opportunely modified in order to cope with partitions.

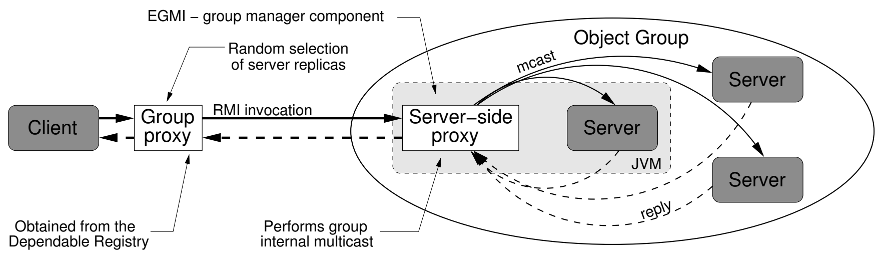

EGMI – group manager component
Random selection of server replicas
Object Group
mcast
Server
Client
Group proxy
RMI invocation
Server–side proxy
Server
JVM
Server
reply
Obtained from the Dependable Registry
Performs group internal multicast

Figure 7. Details of the proxy usage for EGMI.

The resulting agreement algorithm is *indulgent* [21]: safety properties are never violated, while accuracy/liveness properties are linked to the quality of the FD. This enables us to circumvent the FLP impossibility result [20]: two processes belong to the same view as long as they are able to communicate. A sophisticated, self-adapting mechanism is employed to measure the round-trip time between each pair of processes, effectively evaluating their reachability.

The Java Virtual Machine (JVM) offers multiple threads, and although rare, a thread may fail due to an unhandled exception or error leaving a partially failed replica in the system. If this happens, the entire JVM is taken down and recovery from that replica failure is left to the ARM framework.

To perform EGMIs on an object group, a client must first obtain a *group proxy* from the registry. The client may then invoke a method on the group proxy. The group proxy forwards the invocation to the *server-side proxy* on a selected replica, the *contact server*. The contact server is responsible for implementing the required invocation protocol, e.g., for multicast the contact server forwards the invocation to all members in the current view and collects replies and returns a single response to the invoking client. This process is illustrated in Fig. 7.

The server-side proxy is an example of a protocol module; it is also responsible for IGMIs between servers, that are managed in a similar way, but with slightly different semantics, in particular for return value management, as described in Section 3.2.1.

The group proxy maintains a (possibly outdated) view of the system. Constantly updating the view on all the clients (and in the registry) would be extremely expensive, and would turn Jgroup into a closed-group system [24]. For this reason, group proxies updates themselves by piggybacking information during invocations, yielding faster failover [33].

Finally, the dependable registry [36, 37] is an actively replicated database, preventing the registry from becoming a single point of failure. It is replicated using Jgroup itself, and clients and servers access the DR transparently through EGMI; a small bootstrap mechanism is provided for this. This means that in case of partitioning, the DR may be split up among several partitions; when the partition disappears, the state merging service automatically reconciles possibly diverged instances of the database. The database maintains a mapping from a service's name n to the set of replicas $S(n)$ (a list server references) providing that particular service. When a server s joins the group named n, it is added to $S(n)$ through a bind() operation. Clients look up the database asking for the set of servers associated with a specific service name, and obtain as response a proxy object (not unlike standard RMI) containing such information. The membership service and the DR maintain the same information. However, the DR is lazily synchronized with the membership service, since otherwise it would have to participate as a full member in all groups. This synchronization is presented in [33]. Note that $S(n)$ for a given n may contain a number of stale servers, unless the DR is updated in any way. This would cause clients to attempt invocations to stale servers (before removing them), which introduces a significant failover latency. Therefore, periodic updating of the DR is introduced [33] as well.

4. The ARM Framework

This section is organized as follows. We first describe the main elements of the ARM architecture introduced in Section 2. After this panoramic view, we focus on algorithmic details like failure analysis, replication of the RM, etc.

4.1. The Replication Manager

The tasks of the replication manager are (1) to provide interfaces for installing, removing and updating services; (2) to distribute replicas in the target environment, to (best) meet the operational policies for all services (see Section 4.3); (3) to collect and analyze information about failures, and (4) to recover from them. The RM is designed as a central controller, enabling consistent decisions on replica placement and recovery actions. For increased fault-tolerance, however, it is replicated using Jgroup and exploits its own facilities for self-recovery and to bootstrap itself onto nodes in the target environment (see Section 4.7).

Being based on the open group model adapted by Jgroup, external entities are able to communicate requests and events to the RM without the need to join its group, avoiding delays and scalability problems inherent to the closed group model [24].

Two EGMI interfaces are used to communicate with the RM. The Management interface is used by the management client to request group creation, update and removal. The Events interface is used by external components to provide the RM with relevant events for performing its operations. Some of the supported events are described in Section 4.5 and in Section 4.4.

4.2. The Management Client

The management client enables a system administrator to install or remove services on demand. The management client may also perform runtime updates of the configuration of a service. Currently, updates are restricted to changing the redundancy level attributes. It is foreseen that updating the service configuration can be exploited by ARM to support self-optimization.

Additionally, the management client may subscribe to events associated with one or more object groups deployed through ARM. These events are passed on to the management client through the Callback interface, permitting appropriate feedback to the system administrator. Currently, two management client implementations exist, one providing a graphical front-end to ease human interaction, and one that supports defining scripts to perform automated installations. The latter was used to perform the experimental evaluation in Section 5.

```
<Service name="ARM/ReplicationManager">
  <Param name="ServiceMonitorExpiration" value="3"/>
  <ProtocolModules>
    <Module name="GroupMembership"/>
    <Module name="StateMerge"/>
    <Module name="EGMI"/>
    <Module name="Supervision">
      <Param name="GroupFailureSupport" value="no"/>
      <Param name="RemoveDelay" value="5"/>
    </Module>
  </ProtocolModules>
  <DistributionPolicy class="DisperseOnSites"/>
  <ReplicationPolicy class="KeepMinimalInPartition">
    <Redundancy initial="3" minimal="2"/>
  </ReplicationPolicy>
</Service>
```

Figure 8. A sample service configuration description for the RM.

4.3. Policies for Replication Management

The policy-based management paradigm [49] is aimed at enabling administrators to specify how a system should autonomically react to changes in the environment — with no human intervention. These specifications are called *policies*, and are typically defined through high-level declarative directives describing how to manage various system conditions.

ARM requires that two separate policy types be defined in order to support the autonomy properties: (1) the *distribution policy* which is specific to each ARM deployment, and (2) the *replication policy* which is specific to each service deployed through ARM. Alternative policies can be added to ARM. The policies used in the current prototype is just the minimum set.

The purpose of a distribution policy is to describe how service replicas should be allocated onto the set of available sites and nodes. Generally, two types of input are needed to compute the replica allocations of a service: (1) the target environment, and (2) the number of replicas to be allocated. The latter is obtained at runtime from the replication policy. Currently, ARM supports only one distribution policy (DisperseOnSites) that will avoid co-locating two replicas of the same service on the same node, while at the same time trying to disperse the replicas evenly on the available sites. In addition, it will try to keep the replica count per node to a minimum. The same node may host multiple distinct service types. The objective of this distribution policy is to ensure available replicas in each *likely* network partition that may arise. More advanced distribution policies may be defined by combining the above policy with load balancing mechanisms.

Each service is associated with a replication policy, whose primary purpose is to describe how the redundancy level of the service should be maintained. Two types of input are needed; (i) the target environment, and (ii) the initial/minimal redundancy level of the service. Let R_{init} and R_{min} denote the initial and minimal redundancy levels. Currently, only one replication policy (KeepMinimalInPartition) is provided whose objective is to maintain service availability in all partitions. That is to maintain R_{min} in each partition that may arise. Alternative policies can easily be defined, for example to maintain R_{min} in a primary partition only. Or a policy may interpret group failures as a design fault symptom and revert to a previous implementation of the service if such exists.

Policy specifications are part of a sophisticated configuration mechanism, based on XML, that enables administrators to specify (1) the target environment, (2) deployment-based configuration parameters, and finally (3) service-specific descriptors. Fig. 8 shows a small portion of a configuration file, describing the configuration of the replication manager service.

The service configuration permits the operator to define and specify numerous attributes to be associated with the service. Some of the attributes that can be specified include: the *service name*, the *replication policy* to be used in case of failures, the *redundancy* levels to be used initially and to be maintained (minimal), and the set of *protocol modules* used by the service.

Prior to installation of a service, its service configuration is compiled into a runtime representation and passed to the RM. The RM maintains a table of deployed services and their corresponding runtime configurations, allowing the configuration of a service to be modified at runtime. This is useful to adapt to changes in the environment.

4.4. Monitoring and Controlling Services

Keeping track of service replicas is essential to enable discovery of failures and to rectify any deviation from the dependability requirements. Fig. 9 illustrates the ARM failure monitoring architecture. The architecture follows an *event-driven* design in that external components report events collectively to the RM, instead of the RM continuously probing individual components. Both supervision modules and factories are allowed to report failure information to the RM (see Section 4.5 for additional details.) Communication patterns are illustrated with arrows in the figure.

To exploit synergies with existing Jgroup components, tracking is performed at two levels of granularity: groups and replicas. At the group level, the leader of a group is responsible for notifying the RM of any variation occurred in the group membership (event ViewChange.) In this way, the failure detection costs incurred by the PGMS are shared with the RM. Note that membership events cannot discriminate between crash failure and network partition failures. Unfortunately, group-level events are not sufficient to cover *group failure* scenarios in which all remaining replicas fail before being able to report a view change to RM. This can occur if multiple nodes/replicas fail in rapid succession; it may also happen if the network partitions such that only one replica remains in a partition, followed by the failure of that replica.

Both tracking mechanisms are managed by supervision modules, that must be included in the set of protocol modules associated with any Jgroup replicas. View installations are intercepted by the supervision module and reported to the RM through ViewChange events. To avoid that all members of a group report the same information, only the group leader (see Fig. 9 and Fig. 10) multicasts this information to the RM. Based on this information, the RM determines the need for recovery, as discussed in Section 4.6.

An example of a common failure-recovery sequence is shown in Fig. 10, in which node $N1$ fails, followed by a recovery action causing the RM to install a replacement replica at node $N4$.

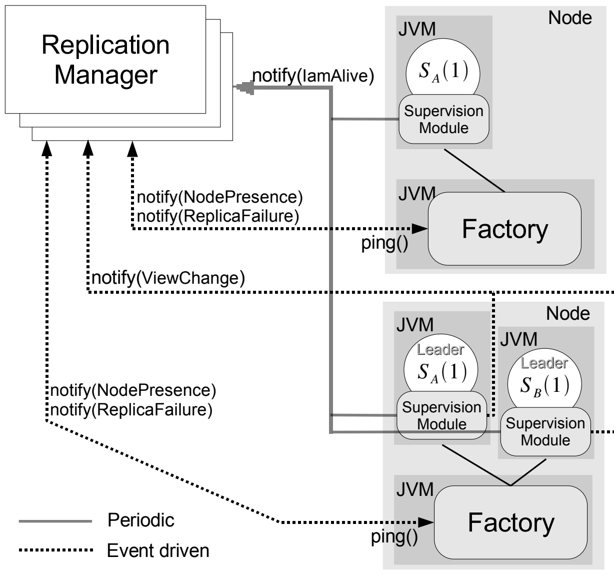


Figure 9. The ARM failure monitoring architecture. ViewChange events are reported only by the leader replica, while IamAlive notifications are performed by all replicas when group failure handling is needed.

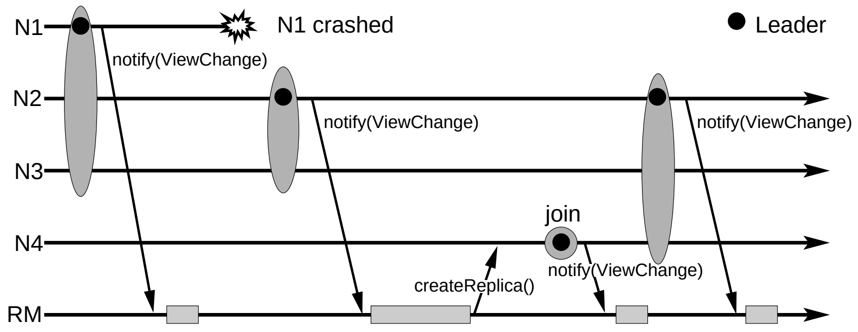


Figure 10. An example failure-recovery sequence.

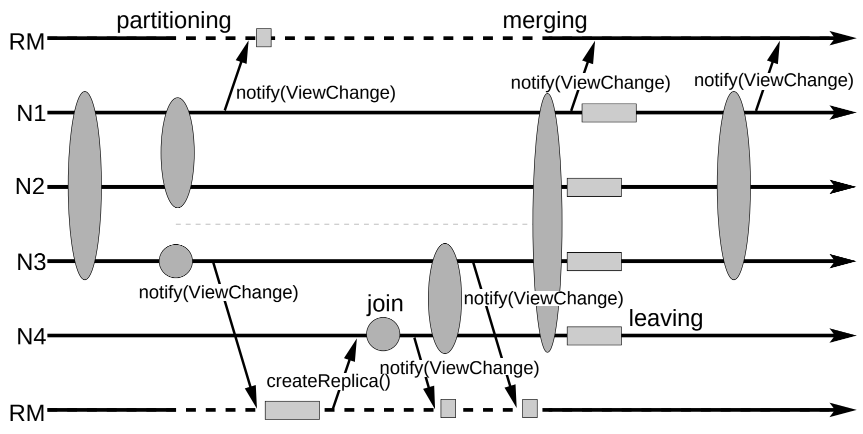


Figure 11. A sample network partition failure-recovery scenario. The partition separates nodes $\{N1,N2\}$ from $\{N3,N4\}$.

The supervision module also provides a controlling part. Let $\mathcal{V}$ denote a view and $|\mathcal{V}|$ its size. If $|\mathcal{V}|$ exceeds the initial redundancy level R_{init} for a duration longer than a configurable time threshold (RemoveDelay in Fig. 8), the supervision module requires one excessive replica to leave the group. If more than one replica needs to be removed, each remove is separated by the RemoveDelay. The choice of which replicas should leave is made deterministically based on the view composition; in this way, the removal can be performed in a decentralized way, without involving the RM. This mechanism is illustrated in Fig. 11, where the dashed timelines indicate the duration of the network partition. After merging, the supervision module detects one excessive replica, and elects $N4$ to leave the group. The reason for the presence of excessive replicas is that during a partitioning, the RM may have installed additional replicas in one or more partitions to restore a minimal redundancy level. Once partitions merge, these replicas are in excess and no longer needed to satisfy the replication policy.

To handle group failures, a lease renewal mechanism is embedded in the supervision module, causing all replicas to issue renew (IamAlive) events periodically to prevent ARM from triggering recovery, as illustrated in Fig. 12. If an expected renew event is not received, ARM will activate recovery. Group failures are extremely rare and typically become even less likely for larger groups. Thus, the renewal period is set to grow exponentially with the group size. This keeps the failure detection time short for small groups that are more likely to fail without notifying the RM, while reducing the number of renew events for larger groups that are less likely to experience a group failure. Hence, the overhead of this mechanism can be made insignificant compared to traditional failure detectors. Note that Jgroup/ARM do not provide support for reestablishing state in case of a group failure. Hence, recovering from a group failure is mainly useful to stateless services; alternatively the service may provide its own persistence mechanism. Since the lease renew mechanism may not be useful to all services and does in fact induce some overhead, it can be (de)activated through the GroupFailureSupport property (see Fig. 8).

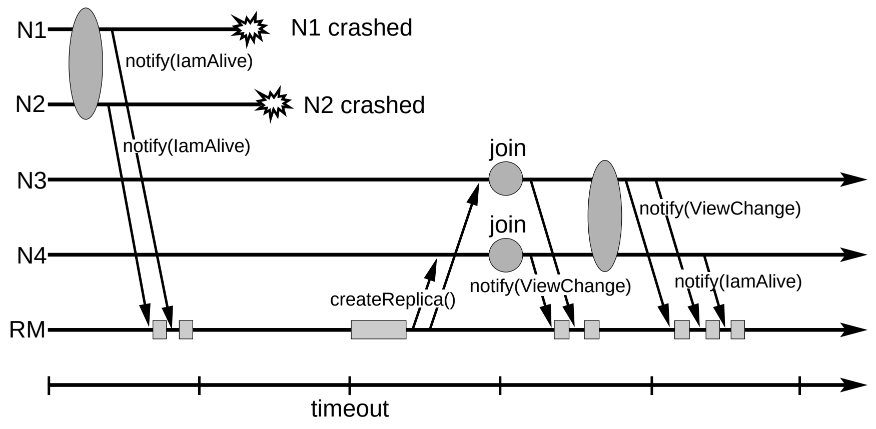


Figure 12. A simple group failure scenario. The timeout indicates that the expected renew event was not received, and hence ARM activates recovery.

4.5. The Object Factory

The purpose of object factories is to facilitate installation and removal of service replicas on demand. To accomplish this, each node in the target environment must run a JVM hosting an object factory, as shown in Fig. 1. In addition, the object factory is also able to respond to queries about which replicas are hosted on the node. The factory also provide means for the RM to keep track of available nodes. The factory maintains a table of local replicas; this state need not be preserved between node failures since all replicas would have crashed as well. Thus, the factory can simply be restarted after a node repair and continue to support new replicas.

Object factories are not replicated and thus do not depend on any Jgroup or ARM services. This allows the RM to bootstrap itself onto nodes in the target environment using the same distribution mechanism used for deploying other services. The RM may create new replicas and remove old ones by invoking the factory of a node. Replicas normally run in separate JVMs, to avoid that a misbehaving replica causes the failure of other replicas within a common JVM.

During initialization, each factory looks for a running RM in the target environment; if present, a NodePresence event is sent to the RM to make it aware of the newly available node. If the RM is not present when the factory is created, the registration of the new node is postponed until the RM is started. At that point, all nodes in the target environment will be probed (the ping() method) for running factory objects by the RM. Together, these two mechanisms enable the RM to become aware of all nodes that are capable of hosting replicas. This probing mechanism is also used by ARM to determine if a node is available before selecting it to host a replica. In addition, the factory monitors the connection between the factory and the replica process, and sends a ReplicaFailure event to the RM if the replica process fails. This is primarily used by ARM to detect replica startup failures.

4.6. Failure Recovery

Failure recovery is managed by the RM, and consists of three parts; (i) determine the need for recovery, (ii) determine the nature of the failures, and (iii) the actual recovery action. The first is accomplished through a *reactive* mechanism based on service-specific timers, while the last two use the abstractions of the replication and distribution policies, respectively.

The RM uses a timer based *Service Monitor* (SM) to keep track of the installed replicas. When deploying a service, a new SM timer instance is associated with that service. If the scheduled expiration time of the SM timer is reached, the recovery algorithm is invoked. To prevent activating unnecessary recovery actions, the SM timer must be rescheduled or canceled before it expires. The ViewChange events reported by the supervision module are used to determine if a SM timer should be rescheduled or canceled. If the received view $\mathcal{V}$ is such that $|\mathcal{V}| \geq R_{\text{min}}$, the SM timer is canceled, otherwise the SM is rescheduled to await additional view changes. Since each service has a separate SM timer, the RM is able to handle multiple concurrent failure activities in separate services, including failures affecting the RM itself.

When deploying a service, the RM will instantiate a service-specific replication policy. During its operation, the RM receives events and maintains state associated with each of the deployed services, including the redundancy level of services. This state can be used by the replication policy to determine the need for recovery.

Upon expiration of the SM timer and detecting that the service needs recovery, the recovery algorithm is executed with the purpose of determining the nature of the current failure scenario. Recovery is performed through three primitive abstractions: *restart*, *relocation* and *group failure handling*. Restart is used when the node's factory remains available, while relocation is used if the node is considered unavailable, and group failure handling is only used if all replicas have failed. The actual installation of replacement replicas is done using the distribution policy.

4.7. Replicating the Replication Manager

The RM is a centralized, yet critical component in our framework. If it were to crash, future replica failures would not be recovered from, severely damaging the dependability characteristics of the system. Also, it would prevent the installation of new services for the duration of its downtime. Therefore, the RM must be replicated for fault tolerance, and it must be able to recover from failures affecting the RM itself, including network partition failures. Careful consideration is required when replicating the RM; one needs to consider the consistency between RM replicas in face of non-deterministic input as well as the merging of states after a network partition.

To make appropriate (recovery) decisions, the RM relies on non-deterministic inputs, such as the SM timers. These inputs are affected by events received by the RM as shown in Fig. 9. Hence, to prevent RM replicas from making inconsistent decisions, only the group *leader* is allowed to generate output. The leadercast protocol is used for the methods that perform non-deterministic computations that always update the state, e.g. createGroup(), while multicast protocol is used for the notify() method. Stronger invocation protocols are not required for these RM methods since invocations related to different groups are commutative. Although notify() is a multicast method, only the RM leader replica is allowed to perform the non-deterministic part of the processing and inform the *follower* replicas if necessary. For example, only the leader performs recovery actions, while the followers are informed about the new location of the replica.

As the replication protocols are implemented on top of the group membership module, leader election can be achieved without additional communication, simply by using the total ordering of members defined in the current view. If the current leader fails, a new view will be installed excluding the current leader, and in effect a follower replica will become the new leader of the group and will be able to resume processing. This also applies to the RM group, ensuring that it can perform self-recovery should the leader RM replica fail.

Since the RM is designed to tolerate network partition failures, it may in *rare* circumstances cause temporary inconsistencies due to EGMI events being handled in multiple concurrent views. However, in most cases, inconsistencies will not occur since each replica of the RM is only "connected" to replicas within its own partition. That is, most events (e.g. the location of replicas determined from view change events) received by the RM replicas reflect the current network partitioning. Hence, a potential inconsistency will be recovered from as soon as additional events cancel them out. If an inconsistency were to persist long enough to cause the RM to activate an unwarranted recovery action, the supervision module would detect this and remove the excessive replicas. Hence, the application semantics of the RM described above enables it to tolerate partition failures; a feature that by far outweighs the sacrifice of slightly weaker consistency. The impact of weaker consistency can only result in higher redundancy levels.

When merging from a network partition scenario, the RM invokes a reconciliation protocol using the state merging service (see Section 3.3), to merge the locations of service replicas. This is feasible since the location of service replicas in each merging partition will, after the merging, be visible to all RM replicas in the merged partition. In addition, the reconciliation algorithm also restarts the SM timers of the involved services, since the RM leader replica of the merged partition might have received information about new services during reconciliation. The latter is primarily a safety measure to prevent premature recovery actions.

The RM relies on the DR to store its object group reference, enabling RM clients such as the supervision module, factory and management client to query the DR to obtain the group reference of the RM. Due to this dependency, ARM has been configured to co-locate RM and DR replicas in the same JVM (see Fig. 2). This excludes the possibility that partitions separate RM and DR replicas, which could potentially prevent the system from making progress.

As mentioned previously, the RM exploits its own embedded recovery mechanism to handle self-recovery in case of RM replica failures. The exception being that the RM cannot tolerate a group failure, since it makes little sense sending IamAlive events to itself.

5. Experimental Evaluation

The recovery performance of Jgroup/ARM has been evaluated experimentally with respect to both node and network failures [31]. An extensive study of its crash failure behavior is presented in [23] and the main findings are summarized in Section 5.5. The study in this paper focuses on the ability to tolerate network instability and partitioning due to network failures. This is done by introducing series of multiple network partitions and merges, which may be near-coincident and occur before the previous has been completely handled by ARM, i.e. the system stabilizes. Two series of experiments are carried out, series (a) with a sequence of two changes and series (b) with four changes of the network connectivity. In the following we present the target system, the experiments and state machine used for the evaluation, followed by our findings.

5.1. Target System

Fig. 13 shows the target system for our measurements. It consists of three sites denoted x, y and z, two located in Stavanger and one in Trondheim (both in Norway), interconnected through the Internet. Each site has three nodes denoted $x1, x2, x3$, etc. Initially nodes $x1, y1, z1$ host the RM, nodes $x2, y2, z2$ host the *monitored service* (MS) and nodes $x3, y3, z3$ host the *additional service* (AS). The latter was added to assess ARM's ability to handle concurrent failure recoveries of more services. An external node hosts the *experiment executor*; for details see [31]. The paper presents ARM's performance with respect to our *subsystem of interest*, the MS service, whose state space contains 10-20 objects. However, all services, including the RM are monitored. The policies used in the experiments are those described in Section 4.3. In all the experiments and for all services, $R_{\text{init}} := 3$ and $R_{\text{min}} := 2$, i.e. all services have three replicas initially and ARM seeks to maintain at least two replicas in each partition.

5.2. The experiments

The connectivity of the nodes of the target environment is called the *reachability pattern*. The reachability pattern may be *connected* (all nodes are in the same partition), or *partitioned* where failures render communication between subsets of nodes impossible. The reachability pattern may change over time, with partitions forming and merging. For the experiments, the reachability patterns are *injected* by the *experiment executor*, where the sites x, y and z are partitioned from each other.

Two series of experiments have been carried out to investigate the reconfiguration and recovery performance: **(a)** when exposed to a single partitioned reachability pattern (Fig. 14(a)), and **(b)** when exposed to a rapid succession of four reachability patterns (Fig. 14(b)).

See also Fig. 11 for an example of the expected ARM behavior during a series (a) experiment. All reachability patterns are injected at random times, the last one returning to the fully connected state. For the duration of an experiment, events of interest are monitored, and post-experiment analysis is used to construct a single global timeline of events. Based on this, density estimates for the various delays involved in detection and recovery are obtained.

This is done by establishing a global state machine representation of the behavior of the MS service under the injected reachability patterns. The state machine is used to perform sanity checks, and to identify sampling points for our measurements. Due to the large number of states, only the initial states are shown in Fig. 15(a) and a trace snapshot in Fig. 15(b).

Each state is identified by its global reachability pattern, the number of replicas in each partition and the number of members in the various (possibly concurrent) views. The number of replicas in a partition is the number of letters x, y, and z that are not separated by a $|$ symbol. The letters refer to the site in which a replica resides. The $|$ symbol indicates a partition between the replicas on its left- and right-hand side. The number in parenthesis in each partition is the number of members in the view of that partition. A partition may for short periods of time have multiple concurrent views, as indicated by the $+$ symbol. Concurrent views in the same partition are not stable, and a new view including all live members in the partition will be installed, unless interrupted by a new reachability

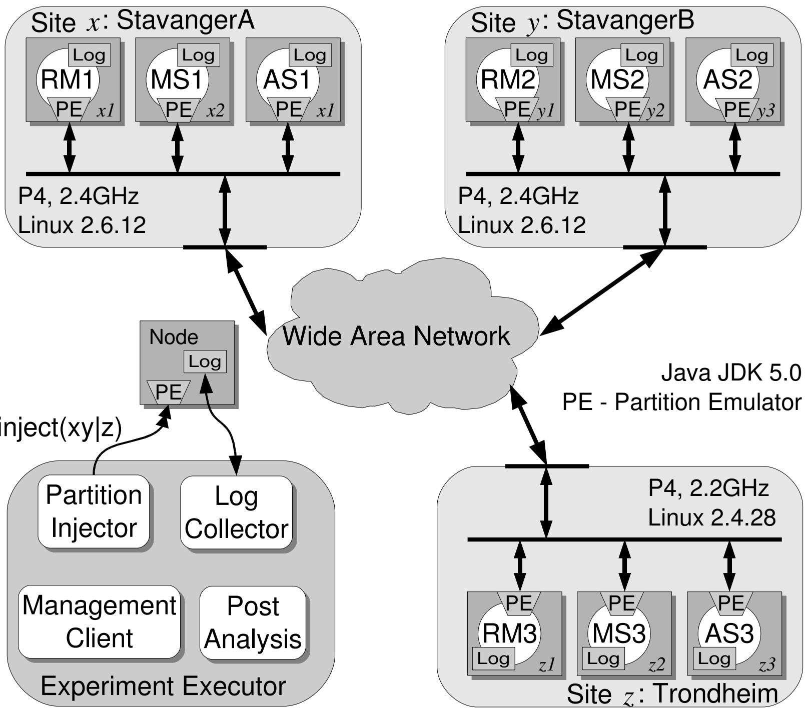


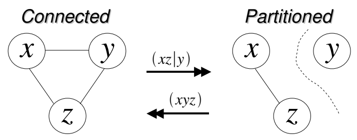
Figure 13. Target system used in the experiments.

change. Two examples: The fully connected steady state is identified by $\langle xyz(3)\rangle$ in which each site has a single replica, and all have installed a three member view. In the state $\langle xx(2) \mid yy(1+1) \mid zz(2)\rangle$ all sites are disconnected from each other and all have installed an extra replica to satisfy the $R_{\text{min}} = 2$ requirement. However, the replicas in site y have still not installed a common two-member view.

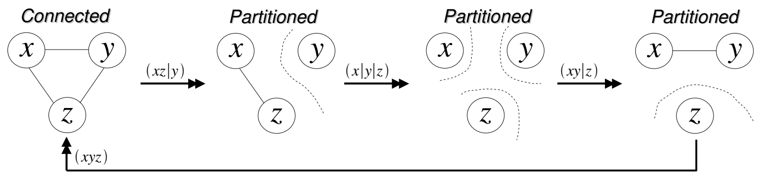
Each state can be classified as stable (bold outline in Fig. 15(a) and Fig. 15(b)) or unstable. Let R_p denote the current redundancy level of a service in partition p. A state is considered *stable* if

$$(R_p = |\mathcal{V}_p| \wedge R_{\text{init}} \geq R_p \geq R_{\text{min}}), \forall p \in \mathcal{P}$$

where $\mathcal{P}$ is the current set of partitions. In the stable states, no ARM action is needed to increase or decrease the service redundancy level. All other states are considered *unstable*, meaning that more events are needed to reach a stable state. Once in a stable state, only a new reachability change can cause it to enter into an unstable state. In Fig. 15(a) and 15(b) we distinguish between *system events* (regular arrow) and *reachability change events* (double arrow). In the evaluation, only relevant events are considered: view change, replica create/remove and reachability change events. View-c denote a view change, where c is the cardinality of the view. Reachability changes are denoted by (xyz), where a



(a) Sample sequence from experiment series (a).



(b) Sample sequence from experiment series (b).

Figure 14. Sample reachability change sequences.

| symbol indicate which sites are to be disconnected. A new reachability change may occur in any state, e.g. note the transition from the unstable state $\langle x(1) \mid yz(2)\rangle$ in Fig. 15(b), illustrating that recovery in partition x did not begin before a new reachability change arrived.

5.3. Injection Scheme

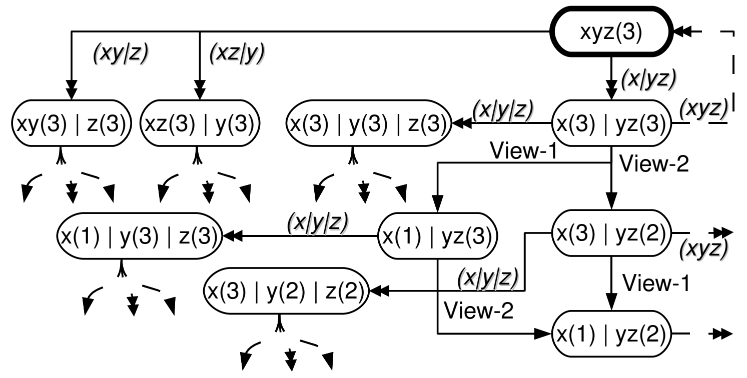
Each injection time I_i is uniformly distributed over the interval $[T_{\text{min}}, T_{\text{max}}\rangle$, where $T_{\text{min}} \leq T_{\text{max}}$ is the minimal time between two injections. In each series, T_{max} is chosen to be longer than the time needed to reach a stable state. Each experiment begins and ends in the fully connected steady state $\langle xyz(3)\rangle$. Fig. 16 shows a timeline of injection events. Let E_i be the time of the last system event before I_{i+1}. E_i events will bring the system into either a stable or unstable state. Nearly-coincident injections tend to bring the system to unstable states at E_i and vice versa. Let D_i denote the duration needed to reach a stable state after injection I_i, where D_i may extend beyond I_j, where $j > i$, before reaching stable.

In *series (a)*, the following injection scheme is applied to emulate real reachability patterns. Let $P_i^{(a)}$ be the set of reachability patterns from which injections are chosen:

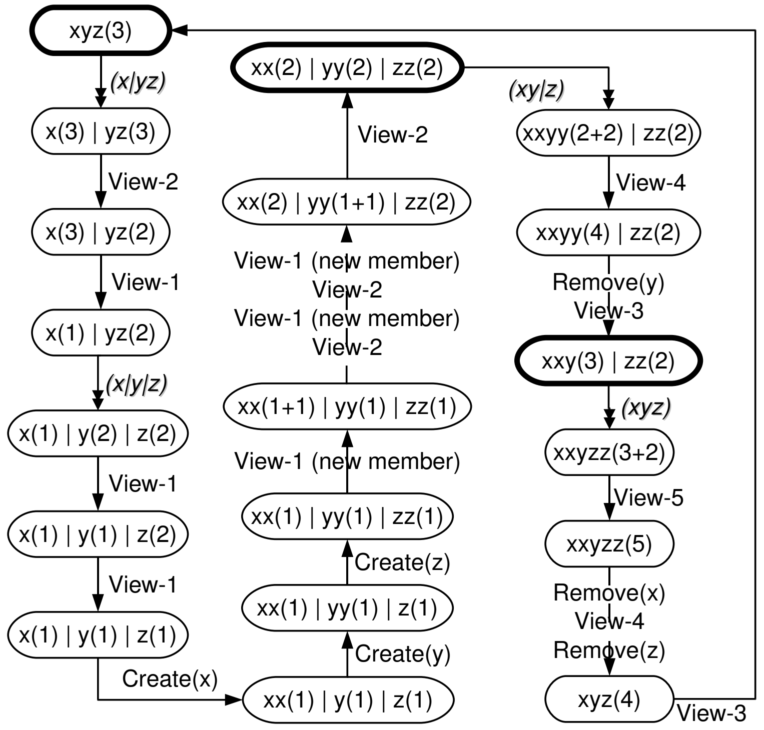
$$P_i^{(a)} = \begin{cases} \{(xy|z), (x|yz), (xz|y)\} & i = 1 \\ \{(xyz)\} & i = 0, 2 \end{cases}$$

where i denotes the injection number and $i = 0$ is the initial state. In this series, $T_{\text{min}} = 15\text{s}$ is used to ensure that the final injection (xyz) occurs after the system has reached a stable state, and $T_{\text{max}} = 25\text{s}$. In *series (b)*, the set of patterns, P_i, from which injections are chosen, is:

$$P_i = \begin{cases} \{(xy|z), (x|yz), (xz|y)\} & i = 1, 3 \\ \{(xyz), (x|y|z)\} & i = 2 \\ \{(xyz)\} & i = 0, 4 \end{cases} \quad (1)$$



(a) Excerpt of initial states/transitions of the state machine.



(b) An example state machine trace. The dashed arrows with multiple events are used to reduce the size of the figure.

Figure 15. Partial state machine for experiments.

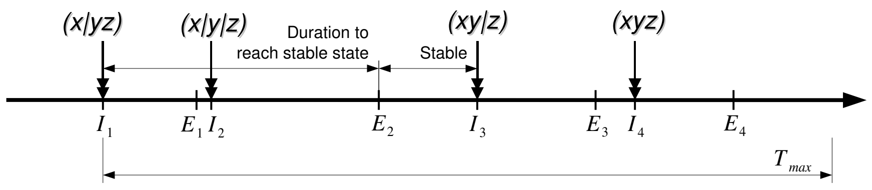


Figure 16. Sample timeline with injection events.

In this case $T_{\text{min}} = 0$s, $T_{\text{max}} = 25$s, $I_1 = 0$ and the following 3 reachability change instants are drawn uniformly distributed over the interval $[0, T_{\text{max}}\rangle$ and sorted such that $I_i \leq I_{i+1}$ yielding the ordered set $\{I_1, I_2, I_3, I_4\}$. Denote the i^{th} reachability pattern injected in the j^{th} experiment by $p_{j,i}$. These are for both types of series drawn uniformly and independently from their corresponding P_i. A sample outcome is shown in Fig. 14(b).

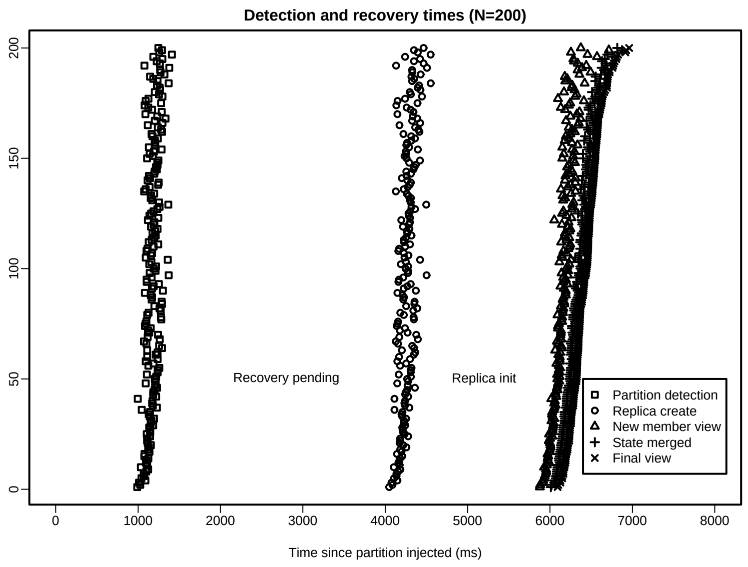
5.4. Experimental Results

The behavior of Jgroup/ARM in response to injections are interpreted as a global event trace, with a trajectory of visited states and the time spent in each of these. This allows us to extract various detection and recovery delays and to determine the correctness of trajectories. This section presents the results obtained for the system's behavior. For the ease of interpretation, the results are presented in the form of kernel density estimates of the probability density functions[†]; see for instance [29]. In addition, to give an uncolored presentation of the observations and reveal any unexpected relationships between the delays of the various stages in the recovery process, the raw data for each experiment for the series (a) experiments are included.

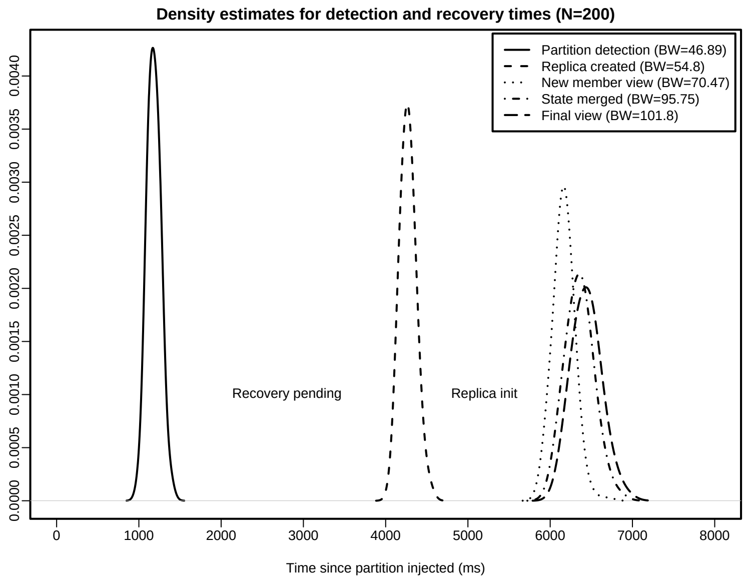
5.4.1. Experiment series (a)

In this series, a total of $N = 200$ experiments were performed. All experiments completed successfully, reaching the final state $\langle xyz(3)\rangle$. Fig. 17(a) shows the time of the various events in the recovery cycle after the injection of a network partition drawn from $P_1^{(a)}$, e.g. $(xy|z)$. The results are sorted according to the total time needed reach the *final view*. Hence, the right-most curve is the empirical CDF of this time. In the proceeding curves it is seen how the time to the other events contribute. No peculiarities is seen in the fault handling times. It is seen that most of the variability is introduced in the partition detection. The time to create a replica and to reach a new member view are close to deterministic. Some additional variability is added in the state merge. This is also observed in Fig. 17(b) that shows the density estimates for the same observations. The *partition detection* curve shows the time it takes

[†]The smoothing bandwidth (BW), given in the legend of each curve, is chosen as a trade-off between detail and smoothness.



(a) Delays per experiment for network partitioning; sorted on the *Final view* curve.



(b) Density estimates for network partition delays.

Figure 17. Network partition delays for series (a) experiments. N is the number of experiments.

to detect that a partition has occurred; that is, the time until the member in the "single site partition" installs a new view. It is this view that triggers recovery in that partition to restore the redundancy level back to $R_{\text{min}} = 2$. The *recovery pending* period is due to the 3s safety margin (ServiceMonitor expiration) ARM uses to avoid triggering unnecessary recovery actions. The *replica init* period is the time it takes to create a new replica to compensate for lack of redundancy. This period is primarily due to JVM initialization, including class loading. The *new member view* curve is the first singleton view installed by the new replica; this occurs as soon as the replica has been initialized. The *state merge* curve shows the start of the state transfer from the single remaining replica to the new replica, while the *final view* curve marks the end of the state merge after which the system is stable.

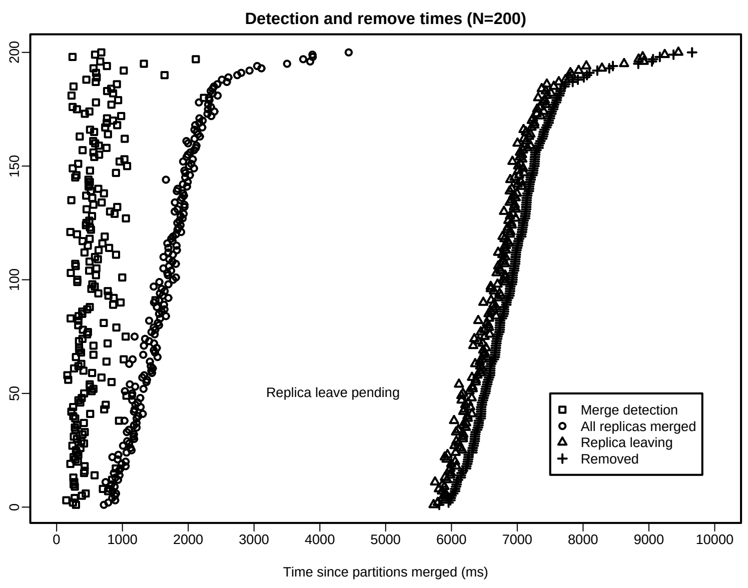
Fig. 18(a) shows the time of events in the remove cycle after the injection of the final (xyz) merge pattern. The results are sorted according to the accumulated time to the excess replica *removed* event, yielding the empirical CDF of the time to this event and an indication of how the times to the preceding events contribute to this. Fig. 18(b) shows the corresponding density estimates. *Merge detection* is the time until the first member installs a new view after the merge injection. This first view is typically not a full view containing all four members, whereas the *all replicas merged* curve is the time it takes for all members to form a common view. The tail of this curve is due to delays imposed by the membership service when having to execute several runs of the view agreement protocol before reaching the final four member view. The *replica leave pending* period is due to the 5s RemoveDelay used by the supervision module to trigger a leave event for some member (see Section 4.4). The last two curves indicate the start of the leave request and installation of a three member view which brings the system back to the steady state $\langle xyz(3) \rangle$. In this case, the three three last inter-event times has a small variability, whereas the time to *merge detection* and the interval until the *all replicas merged* event are more variable and not strongly correlated. This is mainly due to a random number of executions of the view agreement protocol before the final three member view is reached.

5.4.2. *Experiment series (b)*

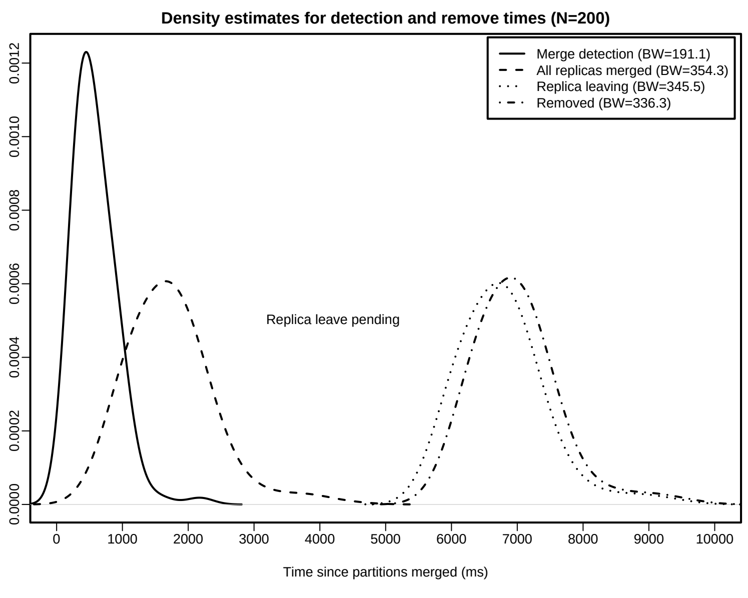
For series (b), $N = 1500$ experiments were performed and density estimates obtained. Omitted from the estimation were six experiments (0.4%) which failed to reach the final state $\langle xyz(3) \rangle$ due to a bug in the Jgroup multicast layer. Table I shows the number of observations for the different combinations of reachability changes injected during series (b) experiments. The table shows how many occurrences were observed for a particular reachability change when the injection was performed when in a stable and in an unstable state. Only density plots are included as the raw data plot do not add further insight.

The plot in Fig. 19(a) shows density estimates for D_2 when starting from a (i) stable or (ii) unstable P_1 reachability pattern and entering a $P_2[0] = (xyz)$ pattern (line 2 in Table I), i.e. a fully connected network. In case (i) (solid curve), when starting from a stable P_1 pattern the following observations have been made:

- The peak at about 6s (approximately 119 observations) is due to removal of an excessive replica installed while in the P_1 pattern. This behavior corresponds to the *removed* curve in Fig. 18(b).
- The 17 observations before the peak are due to experiments that briefly visits $P_2[0]$ before entering a P_3 pattern equal to the initial P_1 pattern. These observations do not trigger removal since they are caused by rapid reachability changes.

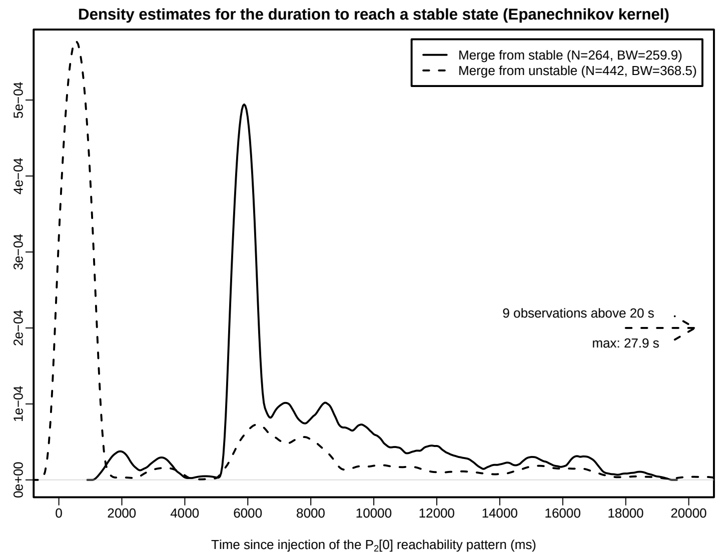


(a) Delays per experiment for network merging; sorted on the *Removed* curve.

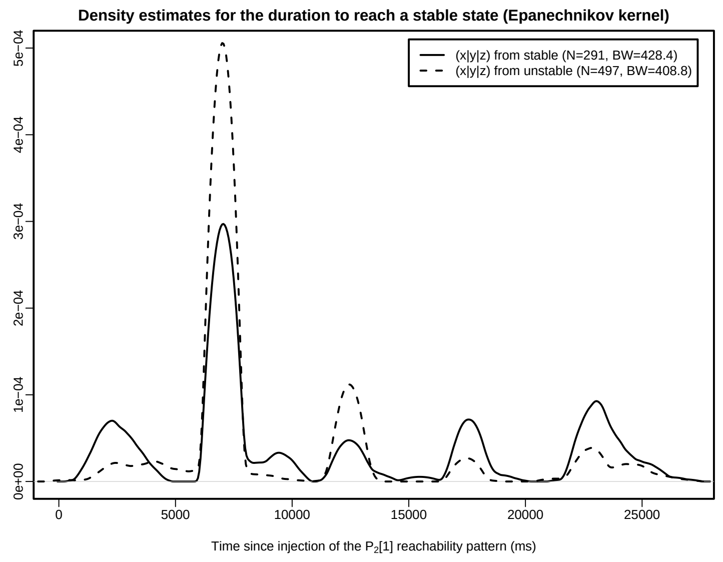


(b) Density estimates for network merge delays.

Figure 18. Network merge delays for series (a) experiments. N is the number of experiments.



(a) Duration to reach a stable state after a $P_2\,[0]$ injection.



(b) Duration to reach a stable state after a $P_2\,[1]$ injection.

Figure 19. Density estimates for the duration D_2 to reach a stable state after a P_2 injection in series (b) experiments. N is the number of experiments in each class.

Table I. Number of observations for different combinations of injections starting from an (un)stable state.

			Starting from			
	Injection	Reachability change	Unstable	Stable	Aggregate	
1	I_1	$P_0 \rightarrow P_1$	—	1494		1494
2	I_2	$P_1 \rightarrow P_2\,[0]$	442	264	706	1494
3	I_2	$P_1 \rightarrow P_2\,[1]$	497	291	788	
4	I_3	$P_2\,[0] \rightarrow P_3$	276	430	706	1494
5	I_3	$P_2\,[1] \rightarrow P_3$	500	288	788	
6	I_4	$P_3 \rightarrow P_4$	844	650		1494

- The rather long tail after the 6s peak is due to variations of the following scenario: In the $P_2\,[0]$ pattern a remove is triggered and before stabilizing a P_3 pattern is injected. Due to the remove, there is again a lack of redundancy, thus ARM triggers another recovery action. Some experiments stabilize in P_3, while others do not complete until reaching the final $\langle xyz(3)\rangle$ state in P_4. This depends on the time between reachability changes.

For case (ii) (dashed curve), starting from an unstable P_1 pattern the following observations have been made:

- There is a peak at about 0.6s due to injections that only briefly visits the P_1 pattern, quickly reverting the partition to a $P_2\,[0] = P_0$ pattern, taking the system back to the $\langle xyz(3)\rangle$ state without triggering any ARM actions. This can happen if $P_2\,[0]$ occurs before the 3s safety margin expires. A total of 288 observations constitute this peak; 54 of these are due to two consecutive injections without intermediate system events.
- There are 7 observations below 6s that are due to ARM triggering recovery in the P_1 pattern (recovery is not completed in P_1, i.e. unstable) and is then interrupted by a short visit to the $P_2\,[0]$ pattern before entering a P_3 pattern identical to the initial P_1 pattern.
- The observations above 6s are similar to the above, except that recovery is completed in $P_2\,[0]$ leading to a $\langle xyzz(4)\rangle$ (or similar) state. Consequently, removal of the excessive replica is triggered in $P_2\,[0]$, but not completed. Hence, it enters P_3 before reaching stable. It may also enter P_4 depending on the time between the reachability changes. The variation seen for these observations are caused by the varying time between reachability changes.
- There are 37 observations above 13s. These follow the scenario above, but the P_3 pattern selected is different from P_1, causing the need to install another replica in the new single site partition. The other partition will then have three replicas. That is, it may stabilize in P_3 in a state similar to $\langle xxy(3v)|zz(2v)\rangle$, or in P_4 in the $\langle xyz(3)\rangle$ state. In latter case, two removes are needed before reaching a stable state, since there will be five replicas in the merged partition. This behavior gives the longest durations for D_2.

The plot in Fig. 19(b) shows density estimates for D_2 when starting from a (iii) stable or (iv) unstable P_1 reachability pattern and entering a $P_2\,[1] = (x|y|z)$ pattern (line 3 in Table I), i.e. a double partition. For both case (iii) (solid curve) and case (iv) (dashed curve), there are multiple peaks at approximately the same time intervals. The observations for case (iv) are mostly due to the same behaviors as in case (iii), except that the initial P_1 pattern has not reached a stable state before the $P_2\,[1]$ reachability change. Hence, we focus only on explaining case (iii):

- The small peak at approximately 2.5s (44 observations) is due to short visits to the $P_2\,[1]$ pattern without triggering recovery before entering a P_3 pattern equal to the P_1 pattern.
- The main peak at 7s (117 observations) is due to recovery in the two new single site partitions eventually leading to a $\langle xx(2)|yy(2)|zz(2)\rangle$ stable state. Recall that the initial P_1 pattern is in a stable state similar to $\langle xx(2)|yz(2)\rangle$ before the double partition injection. This peak is roughly comparable to the *final view* curve in Fig. 17(b).
- The small peak just below 10s (15 observations) is due to short visits to $P_2\,[1]$ and P_3 before stabilizing in P_4 having to remove one replica created in P_1.
- The peak at 12.5s (24 observations) is due to recovery initiated in $P_2\,[1]$, interrupted by a P_3 injection. Recovery is completed in P_3 followed by a replica remove event bringing the system back to a stable state.
- The peak at 17.5s (31 observations) is due to brief visits to $P_2\,[1]$ followed by a P_3 pattern different from P_1 triggering recovery, which eventually completes in P_4 by removing two excessive replicas before reaching stable. Recall that two removals are separated by the 5s RemoveDelay.
- The peak at 23.5s (60 observations) is due to recovery initiated in $P_2\,[1]$, which does not stabilize until reaching P_4. Three removals are needed in this case, each separated by 5s.

5.4.3. Concluding Remarks

The results obtained from our experiments show that Jgroup/ARM is robust with respect to failure recovery, even in the presence of multiple near-coincident reachability changes. Only six experiments (0.4%) failed to reach the expected final state $\langle xyz(3)\rangle$ when exposed to frequent reachability changes. Further analysis is needed to fully understand the cause of this problem and to be able solve it.

The delays observed in the experiments are mainly due to: execution of protocols, e.g. the view agreement protocol, and timers to avoid activating fault treatment actions prematurely. Premature activation could potentially occur when the system is heavily loaded or is experiencing high packet loss rates. The timers constitutes the majority of the observed delays.

5.5. Crash failures

A study where single and nearly coincident node crash failures are induced in a system of eight nodes are reported in [23, 31]. By nearly coincident failure is meant a failure which occurs before the system has recovered from the previous failure. The crash failure handling performance is summarized in Table II and the distribution of the recovery times is shown in Fig. 20. The crash injection scheme used in these experiments is similar to the scheme described in Section 5.3; the node and inject times are randomly selected.

Table II. Experimental testing of Jgroup/ARM crash failure handling performance; computed assuming a node MTBF=100 days.

No. of near coincident crashes	No. of experiments	Approximate occurrence probability	Unsuccessful coverage	Mean recovery time
1	1781	$1 - 10^{-6}$	0.11 %	8.5 s
2	793	10^{-6}	0.76 %	12.8 s
3	407	10^{-11}	2.70 %	17.4 s

A total of 3000 experiments were performed, aiming at provoking a similar number of single node failures, and two and three near coincident node failures. The results are presented in Table II. Some of the experiments aiming at provoking two and three node near coincident node failures produce less due to injections being too far apart or addressing the same node[‡]. Table II also shows the occurrence probability for one, two and three nearly coincident crash failures, given a node mean time between failure of 100 days. The unsuccessful coverage is due to experiments failing to recover correctly due to problems with the Jgroup/ARM framework (see [23] for details.)

The solid line in Fig. 20 shows that recovery from a single crash failure has a small variance. However, the seven runs in the tail have a duration above 10 seconds. These longer recovery times are due to external influence (CPU/IO starvation) on the machines in the target system. This was found by examining the cron job scheduling times, and the running time of those particular runs. Similar observations can be identified for two nearly coincident node failures, while it is difficult to identify such observations in the three failure case. The density curve for the two node failure case in Fig. 20 is clearly bimodal, with a peak at approximately 10 and another at approximately 15 seconds. The left-most peak is due to runs with injections that are close, while the right-most peak is due to injections that are more than 5-6 seconds apart. The behavior causing this bimodality is due to the combined effect of the delay induced by the view agreement protocol, and a 3 second delay before ARM triggers recovery. The injections that are close tend to be recovered almost simultaneously. The density curve for the three node failure cases has indications of being multimodal. However, the distinctions are not as clear in this case. In conclusion, injecting three nearly coincident node failures typically causes longer recovery delays. For additional details see [23, 31].

6. Replicating the Jini Transaction Manager

This section reports on an experience with using Jgroup. The case presented here aims at providing support for fault tolerant transactions by enhancing the Jini transaction service [52] with support for

[‡]This is due to the injection scheme being tailored to allow statistical prediction of the operational availability of the system by a post stratification technique [23].

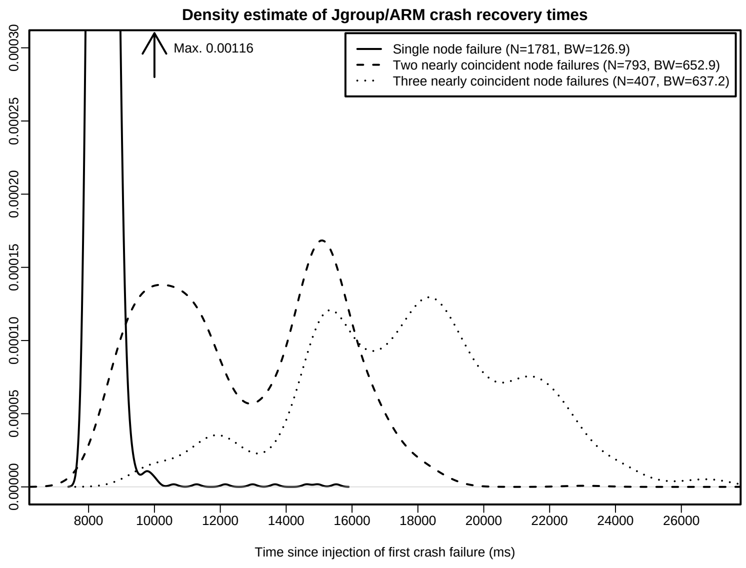


Figure 20. Density estimate of Jgroup/ARM crash recovery times.

replication using Jgroup. Our focus here is on the technical details and adjustments that were necessary to combine Jini transactions with Jgroup. The details are reported in [35, 25, **?**].

Transactions yield consistency and safety properties, whereas replication techniques offers fault tolerance and liveness, all of which are highly desirable properties. Traditionally, transactions and replication are rarely used in combination, even though these properties are complementary and when used together can achieve two additional properties [18]: (i) provide stronger consistency to replicated systems by supporting non-deterministic operation, and (ii) to provide higher availability and failure transparency to transactional systems.

The Jini middleware architecture [52] comes with a wide range of services, among them a transaction service and a lookup service, aimed at simplifying development of distributed applications. The Jini transaction service offers a transaction manager (TM) called Mahalo which provides methods for creating transactions, joining transaction participants (TPs) and to commit/abort transactions. To handle these tasks, Mahalo implements a two-phase commit (2PC) protocol [6]. The Jini transaction service makes use of Java RMI for interaction between the TM and TPs, making it a suitable candidate for integration with Jgroup.

The implementation described in [35, 25, **?**] supports passive replication of both the transaction manager as well as the transaction participants. In this context a primary partition model is assumed, disallowing replicas in nonprimary partitions from modifying their state.

In the context of Jini the TM is a centralized component involved in distributed transactions, and as such if the TM fails, the entire transaction system becomes unavailable. This is since the 2PC protocol may block due to the failure of the TM, preventing any transactions from committing. The main task of the TM is to provide participants with decisions on whether to commit or abort a given transaction. This decision relies on several inputs, among them timeouts for unresponsive participants. Such timeouts may be triggered differently on the different replicas. Furthermore, the Jini TM allows multiple threads to handle different transactions, which could cause different ordering of transactions at different replicas. Hence, the transaction processing performed by the Jini TM may introduce several sources of non-determinism, which makes the passive replication scheme the appropriate choice for a replicated TM.

To support safety, transaction systems typically makes the commit/abort decisions persistent by storing it on stable storage at the end of the prepare phase [6]. However, in [25] it is argued that, when in a replicated setting, it would be more beneficial to use the backup replicas to persist the decision. For instance, if the primary TM failed, a prepared transaction may block until the failed TM replica has recovered if a local only log is used. Furthermore, a performance gain may be obtained since a backup replica (becoming the new primary) can commit a prepared transaction immediately if the current primary replica fails.

From a technical point of view, implementing a passively replicated TM based on Jini we faced several challenges. First of all, we wanted to reuse the 2PC protocol and the main framework already provided by Jini without having to modify the TM code. This would simplify migration to new versions of the TM when another is released by the Jini project. The replicated TM, called pGahalo, therefore wraps the TM implementation class and provides a new implementation of the TM interface methods that are augmented with appropriate invocation protocols. However, not unexpectedly the TM state had to be passed to the TM backup replicas, mandating that the TM state be serializable. This was not supported in the Jini implementation since the state only reside at a single TM. Unfortunately, to accommodate this the class representing the TM state had to be changed so that it could be serialized and transmitted over the network; in addition, certain fields had to be marked as transient so that they would not be passed over the network, e.g. fields that are used only locally such as the log manager and task manager. Upon receiving a new state, these local fields are simply copied from the current TM state at the receiving replica. In addition to these changes, the design of the TM state class also contains code for the 2PC protocol. To support persistence of commit/abort decisions through the backup replicas, rather than logging to stable storage, code was introduced to ensure that the backup replicas received information about the prepared state of all participants, and also to notify the backups of the commit state of the participants.

The above is necessarily a brief and partial description of the transaction manager part of the replicated transaction system described in [35, 25, **?**]. Similar and other considerations are necessary for the transaction participants. In their work, they implemented a pair of replicated bank servers (the transaction participants) along with a client application to test the replicated transaction system. For additional details about the test runs see [25, **?**].

The main lesson learned from this experience with using Jgroup, is that it is fairly easy to use many of the Jgroup features on an existing Java RMI system. However, as described above, a transaction system is complex and it is not always straightforward to introduce replication into a system designed to be non-replicated. In some ways, replication with Jgroup cannot be made entirely transparent to the

Table III. Summary of differences between Jgroup and Java RMI application development.

	Optional	Mandatory
Jgroup initialization		One line of code.
Access to services	For implicit access; no lines of code (configured using XML).	For explicit access; complexity depends on the service (cf. state merge service).
State merge service	For stateless servers, no change necessary	For stateful servers, two additional methods must be implemented.
Method annotation	For anycast protocols	For other protocols; add one annotation per method.

application developer. Designing a system with replication in mind from the start is much easier than trying to adapt an existing design.

Developing a replicated system with Jgroup is very similar to building a Java RMI application. A summary of differences is provided in Table **??**. As the table shows, the developer must implement two server-side methods for merging server state. Care must be taken to avoid that the server state contain references to objects that are not useful or desirable to serialize or references to very large object graphs that would result in long delays during state merge. Other than state merge, the differences between developing with Jgroup instead of Java RMI are mostly negliable, unless special Jgroup services are needed.

7. Related Research

Numerous research efforts have addressed the problem of building fault tolerant distributed systems, most of which are based on group communication [7]. Mostly, efforts [30, 42, 17, 38, 5, 40] have focused on two slightly different distributed object technologies — CORBA [45] and Java RMI [53]. Below, Jgroup/ARM is briefly contrasted with similar technologies.

JavaGroups [5] is a message-based GCS written in Java providing reliable multicast communication. The RMI facility of JavaGroups is not transparent, as it is based on the exchange of objects that encode method invocation descriptions. Aroma [40] provides transparent fault tolerance through a partially OS-specific interception approach and relies on an underlying GCS implemented on the native operating system. Spread [1] is a generic message-based GCS implemented in C, but it also provides a Java API. Like Jgroup, Spread is designed especially for wide area networks. Spread uses a client-daemon architecture, where only the daemons participate in the execution of group membership changes and routing in the overlay network. A similar approach is used by Jgroup [37], but in the experiments all servers (clients in Spread terminology) were assign a separate daemon. Additionally, Spread supports low-latency forwarding mechanisms to improve performance; Jgroup currently does not support such forwarding mechanisms. Ensemble [22] is another partition-aware GCS, which follows the close group model, where each communicating entity must become a group member to communicate with the rest of the group. Jgroup uses the open group model where an entity (e.g. client) can perform invocations on the group without first becoming a group member.

What distinguishes Jgroup (in part or whole) from the toolkits mentioned above is that Jgroup is based on RMI-like communication and is implemented in its entirety in Java, and does not rely on any underlying toolkits or a special operating system. Being based on GMI rather than message multicasting simplifies the adoption of Jgroup as the fault tolerance framework for middleware environments based on Java RMI, such as Jini [52] and J2EE [51]. In this respect, Jgroup has been used to enhance the Jini transaction manager with replication support [35, 25]. Other works have focused on replication support for J2EE, including [55, 26]. Moreover, while most other systems rely on specifying a replication style (typically active or passive) on a per object basis, Jgroup allows the application designer to specify the replication protocol to use, on a per method basis. This approach is more efficient resource wise.

In the terminology of Davidson et al. [13], Jgroup uses an *optimistic-semantic* strategy. Jgroup is optimistic in that operations may be executed in different partitions, and thus global inconsistencies may be introduced. When partitions merge, states are propagated to allow application-specific detection and resolution of potential inconsistencies. Jgroup also uses the semantics of operations to improve the performance of certain operations, while maintaining the required degree of consistency. Fulfillment transactions [34], where some operations performed in a nonprimary partition are queued and performed when the network merges, is another strategy that has been proposed to handle consistency in partitioned networks. The TACT [56] framework allows applications to tune the level of availability/consistency that they require. The state merging mechanism adopted in Jgroup may be compared with the conflict detection and resolution mechanisms of Bayou [54], a weakly connected partitionable replicated storage system. In Bayou, write updates are propagated epidemically among processes through anti-entropy interactions, and application-specific *dependency check* procedures and *merge procedures* are executed to solve potential conflicts occurred during disconnections. The use of application-specific procedures in the state merging service of Jgroup reproduces the approach of Bayou; they differ, however, in how and when these updates are applied. Bayou is designed for mobile environments where disconnected operations is the norm; Jgroup is designed for wired environments where partitionings are occasional, thus providing a stronger semantics in the absence of partitionings.

Fault treatment techniques similar to those provided by ARM were first introduced in the Delta-4 project [46]. Delta-4 was developed in the context of a fail-silent network adapter and does not support network partition failures. Due to its need for specific hardware and OS environments, Delta-4 has not been widely adopted. None of the Java-based fault tolerance frameworks support mechanisms similar to those of ARM, to deploy and manage dependable applications with only minimal human interaction. These management operations are left to the application developer. However, the FT CORBA standard [44] specifies certain mechanisms such as a generic factory, a replication manager and a fault monitoring architecture, that can be used to implement management facilities. However, the standard makes explicit assumptions that the system is not partionable, a unique feature of Jgroup/ARM. Furthermore, applications are required to be deterministic when processing invocations, and it allows implementers to use proprietary low-level mechanisms to implement the standard. The latter prevents interoperability between ORB vendors [19]. Eternal [42, 41] is probably the most complete implementation of the FT CORBA standard. It supports distributing replicas across the system, however, the exact workings of their distribution approach has not been documented. DOORS [43] is a framework that provides a partial FT CORBA implementation, focusing on passive replication. It uses a centralized ReplicaManager to handle replica placement and migration in response to failures. The ReplicaManager component is not replicated, and instead performs periodic checkpointing of its state tables, limiting its usefulness since it cannot handle recovery of other applications when the ReplicaManager is unavailable. Also the MEAD [48] framework implements parts of the FT CORBA standard, and supports recovery from node and process failures. However, recovery from a node failure requires manual intervention to either reboot or replace the node, since there is no support for relocating the replicas to other nodes. ARM supports object, node and partition failures, and is able to relocate/replace replicas to automatically restore the desired redundancy level. AQuA [47] is also based on CORBA and was developed independently of the FT CORBA standard. AQuA is special in its support for recovery from value faults, while ARM is special in supporting recovery from partition failures. AQuA adopts a closed group model, in which the group leader must join the dependability manager group in order to perform notification of membership changes (e.g. due to failures). Although failures are rare events, the cost of dynamic joins and leaves (run of the view agreement protocol), can impact the performance of the system if a large number of groups are being managed by the dependability manager. Jgroup/ARM uses the more scalable open group model. Farsite [15] is a distributed file system whose replica placement scheme is designed for large scale placement of file replicas using a distributed, iterative and randomized algorithm. ARM uses a centralized algorithm to perform replica placement, which can be customized for the current distribution policy.

None of the other frameworks that support recovery focus on tolerating network partitions. Nor do they explicitly make use of policy-based management, which allows ARM to perform recovery actions based on predefined and configurable policies enabling self-healing and self-configuration properties, ultimately providing autonomous recovery.

To our knowledge, evaluation of network instability tolerance of fault-treatment systems, has not been conducted before. However, the Orchestra [14] fault injection tool has been used to evaluate a group membership protocol by discarding selected messages to test the robustness of the protocol. Loki [9] has been used to inject correlated network partitions to evaluate the robustness of the Coda filesystem [28].

8. Conclusions

We have presented the design, implementation and evaluation of Jgroup/ARM. 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 existing distributed object models, Jgroup has the primary goal of supporting reliable and highly-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 augments Jgroup with a framework enabling simplified administration of replicated services, made possible through configurable distribution and replication policies. As our experimental evaluations demonstrate, Jgroup/ARM is robust, also when exposed to multiple nearly-coincident failures. The node crash failure coverage is approximately 99.9% and with a rapid succession of four network reachability changes, 99.6% of the experiments recover to the expected steady state.

Acknowledgments

The authors wish to thank Heine Kolltveit and Rohnny Moland for commenting on the discussion of replicated transactions. We also wish to thank the anonymous reviewers for their helpful comments.

REFERENCES

1. Y. Amir, C. Danilov, and J. Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, New York, June 2000.
2. Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Trans. Software Eng.*, 27(4):308–336, Apr. 2001.
3. Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Applications. In *Proc. of the 18th Int. Conf. on Distributed Computing Systems*, pages 184–191, Amsterdam, The Netherlands, May 1998.
4. 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.
5. B. Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University, July 1998.
6. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):36–53, Dec. 1993.
8. K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distibuted Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, 1987.
9. R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7):593–605, July 2004.
10. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
11. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, Dec. 2001.
12. G. Coulson, J. Smalley, and G. S. Blair. The Design and Implementation of a Group Invocation Facility in ANSA. Technical Report MPG-92-34, Department of Computing, Lancaster University, UK, 1992.
13. S. B. Davidson, H. G.-Molina, and D. Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3), Sept. 1985.
14. S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Technical Report CSE-TR-318-96, University of Michigan, EECS Department, 1996.
15. J. R. Douceur and R. Wattenhofer. Large-Scale Simulation of Replica Placement Algorithms for a Serverless Distributed File System. In *MASCOTS*, pages 311–319, 2001.
16. P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA Objects: a Marriage Between Active and Passive Replication. In *Proc. of the 2nd Int. Conf. on Dist. Applic. and Interop. Systems*, Helsinki, Finland, June 1999.
17. 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.
18. P. Felber and P. Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *Proc. of the Int. Sym. Distributed Objects & Applications*, pages 737–754, Irvine, CA, 2002. Springer-Verlag.
19. P. Felber and P. Narasimhan. Experiences, Approaches and Challenges in Building Fault-Tolerant CORBA Systems. *IEEE Trans. Comput.*, 53(5):497–511, May 2004.
20. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

21. R. Guerraoui. Indulgent algorithms. In *In Proceedings of the 19th annual ACM Symposium on Principles of Distributed Computing Systems (PODC'00)*, pages 49–63, Portland, OR, 2000.
22. M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Jan. 1998.
23. B. E. Helvik, H. Meling, and A. Montresor. An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In *Proc. of the Fifth European Dependable Computing Conference*, volume 3463 of *LNCS*, pages 179–198. Springer-Verlag, Apr. 2005.
24. C. T. Karamanolis and J. Magee. Client-Access Protocols for Replicated Services. *IEEE Trans. Software Eng.*, 25(1), Jan. 1999.
25. H. Kolltveit. High Availability Transactions. Master's thesis, Dept. of Computer and Information Science, Norwegian University of Science and Technology, Aug. 2005.
26. S. Labourey and B. Burke. *JBoss AS Clustering*. The JBoss Group, 7th edition, May 2004.
27. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
28. R. M. Lefever, M. Cukier, and W. H. Sanders. An Experimental Evaluation of Correlated Network Partitions in the Coda Distributed File System. In *Proc. of the 22nd Symp. on Reliable Distributed Systems*, pages 273–282, Florence, Italy, Oct. 2003.
29. P. A. W. Lewis and E. J. Orav. *Simulation Methodology for Statisticians, Operation Analyst and Engineers*, volume 1 of *Statistics/Probability Series*. Wadsworth & Brooks/Cole, 1989.
30. S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, 1995.
31. H. Meling. *Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation*. PhD thesis, Norwegian University of Science and Technology, Dept. of Telematics, May 2006.
32. 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.
33. H. Meling and B. E. Helvik. Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In *Proc. of the 23rd Int. Performance, Computing, and Comm. Conf.*, Phoenix, Arizona, Apr. 2004.
34. P. M. Melliar-Smith and L. E. Moser. Surviving Network Partitioning. *IEEE Computer*, 31(3):62–68, 1998.
35. R. Moland. Replicated Transactions in Jini: Integrating the Jini Transaction Service and Jgroup/ARM. Master's thesis, Dept. of Electrical Engineering and Computer Science, University of Stavanger, June 2004.
36. 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.
37. A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.
38. 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 Int. Conf. on Dist. Applic. and Interop. Systems*, Helsinki, Finland, June 1999.
39. R. Murch. *Autonomic Computing*. On Demand Series. IBM Press, 2004.
40. N. Narasimhan. *Transparent Fault Tolerance for Java Remote Method Invocation*. PhD thesis, University of California, Santa Barbara, June 2001.
41. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal - a Component-Based Framework for Transparent Fault-Tolerant CORBA. *Softw., Pract. Exper.*, 32(8):771–788, 2002.
42. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly Consistent Replication and Recovery of Fault-Tolerant CORBA Applications. *Comput. Syst. Sci. Eng.*, 17(2), 2002.
43. B. Natarajan, A. S. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Proc. of the 2nd Int. Sym. Distributed Objects & Applications*, pages 39–48, Antwerp, Belgium, Sept. 2000.
44. OMG. Fault Tolerant CORBA Specification. OMG Document ptc/00-04-04, Apr. 2000.
45. OMG. *The Common Object Request Broker: Architecture and Specification, Rev. 3.0*. June 2002.
46. D. Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, Feb. 1994.
47. Y. Ren et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. *IEEE Trans. Comput.*, 52(1):31–50, Jan. 2003.
48. C. F. Reverte and P. Narasimhan. Decentralized Resource Management and Fault-Tolerance for Distributed CORBA Applications. In *Proc. of the 9th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2003.
49. M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994.
50. M. Solarski and H. Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *Proc. of the Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC 2002*, Oxford, England, Aug. 2002.
51. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, Nov. 2003.
52. Sun Microsystems. *Jini Architecture Specification, Version 2.0*, June 2003.
53. Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.10*, Feb. 2004.
54. D. B. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 172–182. ACM Press, 1995.

55. H. Wu, B. Kemme, and V. Maverick. Eager Replication for Stateful J2EE Servers. In *CoopIS/DOA/ODBASE (2)*, 2004.
56. H. Yu and A. Vahdat. Building Replicated Internet Services Using TACT: A Toolkit for Tunable Availability and Consistency Tradeoffs. In *Proc. of the 2nd Int. Workshop on Adv. Issues of E-Commerce and Web-based Information Systems*, June 2000.