An Architecture for Self-healing Autonomous Object Groups

Hein Meling

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

Abstract. Jgroup/ARM is a middleware for developing and operating dependable distributed Java applications. Jgroup integrates the distributed object model of Java RMI with the *object group* paradigm, enabling construction of replicated servers that offer dependable services to clients. ARM aims to improve the dependability characteristics of systems through fault treatment, focusing on operational aspects where the gain in terms of improved dependability is likely to be the greatest. ARM offers two core mechanisms: recovery from *node, object* and *network failures* and distribution of replicas. ARM identifies failures and reconfigures the system according to its dependability requirements.
This paper proposes an enhancement of the ARM framework in which replica placement is performed in a *distributed* manner, eliminating the need for a centralized manager with global information about all object groups. Instead each autonomous object group handles their own replica placement based on information from nodes. Assuming that multiple objects groups are deployed in the system, this constitutes a *distributed replica placement* scheme. This scheme enables the implementation of self-healing object groups that can perform fault treatment on themselves. Advantages of the approach: (a) no need to maintain global information about all object groups which is costly and limits scalability, (b) reduced infrastructure complexity, and (c) less communication overhead.

1 Introduction

Networked computer systems are prevalent in most aspects of modern society, and we have become dependent on such computer systems to perform many critical tasks. Moreover, making such systems *dependable* is an important goal. However, dependability issues are often neglected when developing systems due to the complexities of the techniques involved. A common technique used to improve the dependability characteristics of systems is to *replicate* critical system components whereby the functions they perform are repeated by multiple replicas. Replicas are often distributed geographically and connected through a network as a means to render the failure of one replica independent of the others. However, the network is also a potential source of failures, as nodes can become temporarily disconnected from each other, introducing an array of new

problems. The majority of previous projects [1–5] have focused on the provision of middleware libraries aimed at simplifying the development of dependable distributed systems, whereas the pivotal deployment and operational aspects of such systems have received very little attention.

This paper presents an architecture for *Distributed Autonomous Replication Management* (DARM), aimed at improving the dependability of systems through a self-managed fault treatment mechanism that is adaptive to network dynamics and changing requirements. Consequently, the architecture improves the deployment and operational aspect of systems, where the gain in terms of improved dependability is likely to be the greatest, and also reduces the human interactions needed. The architecture builds on our experience [6, 7] with developing a prototype that extends Jgroup [2] with fault treatment capabilities. The new architecture relies on a distributed approach for replica distribution (placement), thereby eliminating the need for a centralized management infrastructure used in our previous work [6, 7]. Distributed replica placement enables deployed applications (implemented as object groups) to implement autonomic features such as self-healing by performing fault treatment on themselves. Fault treatment represents a non-functional aspect and is easily implemented as a separate protocol module to separate it from application concerns.

Jgroup [2] is a group communication service that integrates the Java RMI distributed object model with object groups. It supports *partition-awareness*: replicas placed 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. By supporting partitioned operation, Jgroup trades consistency for availability, whereas other systems takes a *primary partition* approach [8], ensuring consistency by allowing only a single partition to make progress. A *state merging service* is provided to simplify the re-establishment of a consistent global state when partitions merge.

DARM offers automated mechanisms for performing management activities such as distributing replicas among sites and 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 [3, 4, 2]. DARM achieves its goal through three core paradigms: *policy-based management* [9], where application-specific distribution and fault treatment policies are used to enforce dependability requirements; *self-healing* [10], where failure scenarios are discovered and handled through recovery actions with the objective to minimize the period of reduced failure resilience; *self-configuration* [10], 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 [11]. DARM follows a non-intrusive system design, where the operation of deployed services is decoupled from DARM during normal operation. Once a service is installed, it becomes an "autonomous" entity, monitored by DARM until explicitly removed. This design principle enables support for a large number of object groups. The

Jgroup/DARM framework shares many of its goals with other fault tolerance frameworks, notably Delta-4 [12], AQuA [13], FT CORBA [14], and our previous implementation called ARM [6]. The novel features of Jgroup/DARM when compared to other frameworks include: autonomous management facility based on policies, distributed replica distribution and fault treatment, support for partition awareness, and interactions based solely on RMI.

Organization: Section 2 presents the system model and Section 3 gives an overview of Jgroup/DARM. In Section 4 the DARM framework is described. Section 5 compares DARM with related work and Section 6 concludes.

2 System Model and Assumptions

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 set of nodes, N, that may host application services and infrastructure services, in the form of server objects (or *replicas*), is called the *target environment*. The set N is comprised of one or more subsets, N_i, representing the nodes in site i. Sites are assumed to represent different geographic locations in the network, while nodes within a site are in the same local area network. A node may host several different replica types, but it may not host two replicas of the same type.

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, they may return to being *operational* after an appropriate *recovery* action. Byzantine failures are not considered. Communication channels may omit to deliver messages; a communication substrate handles message retransmission, also using alternative routes [2]. Long-lasting *partitionings* may also occur, in which certain communication failure scenarios may disrupt communication between multiple sets of objects forming *partitions*. Objects in 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/DARM is 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.

3 Jgroup/DARM Overview

Jgroup [2] supports dependable application development by means of replication, based on the *object group* paradigm [8], where a set of server objects form a group to coordinate their activities and appear to clients as a single server.

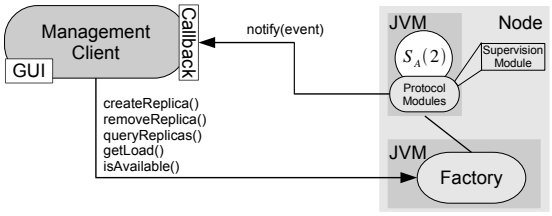

Fig. 1. Overview of DARM components.

Jgroup provides a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS). The PGMS provides replicas with a consistent view of the group's current membership, enabling coordination of their actions. Reliable communication between clients and groups is handled by the GMIS and takes the form of *group method invocations* (GMIs) [2], that result in methods being executed by replicas forming the group. To clients, GMIs are indistinguishable from ordinary 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 proxy maintains information about the group composition, and handles invocations on behalf of clients by establishing communication with replicas and returning the result to the invoking client. On the server side, the GMIS enforces reliable communication among replicas. The SMS facilitate re-establishing a consistent shared state when partitions merge by handling state diffusion among partitions. Jgroup also includes a *dependable registry* (DR) allowing clients to locate object groups.

The ARM framework presented in [7, 6] supports seamless deployment and operation of dependable services. Within the target environment, issues related to service deployment, replica distribution and recovery from failures are autonomically managed by ARM, following the rules of user-specified distribution and fault treatment policies. Maintaining a fixed redundancy level is a typical requirement specified in the fault treatment policy.

In this paper, DARM is proposed in which fault treatment and replica distribution is performed in a distributed manner, rather than relying on a centralized (but replicated) replication manager (RM) component to handle these important mechanisms. The RM implemented in ARM [6] maintains global information about all object groups, which is costly as complex protocols are needed to maintain consistency across RM replicas, and each object group must report view changes to the RM replicas. This imposes an additional delay before fault treatment is activated, but more importantly it also limits the scalability (no. of groups) that can be supported by ARM. The proposed algorithm for distributed replica placement enables the implementation of a distributed fault treatment mechanism. However, it also introduces additional challenges with respect to appropriate load balancing of replicas on the nodes in the target environment.

Fig. 1 illustrates the core components and interfaces supported by the DARM framework: a supervision module associated with each application replica (S_A), an object factory deployed at each node in the target environment, and a management client used to interact with object factories to install/remove replicas.

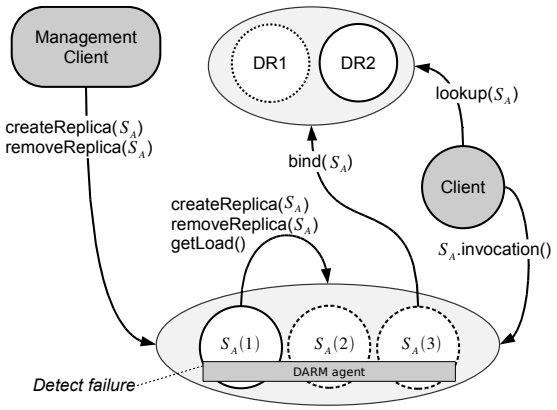

Fig. 2. The Jgroup/DARM architecture.

The *supervision module* is the DARM agent co-located with each replica which is responsible for collecting and analyzing failure information obtained from view change events generated by the PGMS, and reconfigure the system on-demand according to the configured policies. It is also responsible for decentralized removal of excessive replicas. The *object factories* enable the management client to install/remove replicas, as well as to respond to queries about replicas on the local node, and its current load. The *management client* provide administrators with an interface through which to install and remove applications in the system and to specify and update the distribution and fault treatment policies to be used. It can also be used to obtain monitoring information about running services. Overall, the interactions among these components enable the DARM agent to make proper recovery decisions, and allocate replicas to suitable nodes in the target environment.

Next, a brief description of a minimal Jgroup/DARM deployment is given, as shown in Fig. 2. Only two different groups are shown. The DR service represent the naming service infrastructure component and is required in all Jgroup/DARM deployments. In addition, each application service must contain the DARM agent, the supervision module, as discussed above. The figure also illustrates a service labeled S_A that is implemented as a simple object group managed through Jgroup/DARM. Finally, two clients are shown: one client interacts with the S_A object group, while the other is the management client used to create and remove object groups by interacting with object factories. Object factories are not shown, but are present at each node in the target environment. The main communication patterns are shown as graph edges. For example, the DARM agent associated with an object group detect failures by monitoring the current membership of the group, and activate fault treatment actions as needed to recover from various failure scenarios. When joining the system, replicas must bind themselves to the same name (e.g. S_A) 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.

4 The DARM Framework

This section describes the main elements of the DARM architecture and provide an informal discussion of the algorithms related to failure analysis and recovery. Algorithms are provided in [15]. The DARM architecture borrows parts of its infrastructure from ARM [6], and where appropriate the differences between the two are explained.

4.1 The Management Client

The management client enables a system administrator to install or remove services on demand. The initial deployment of replicas is handled by the management client using the distribution policy discussed below. The management client may also perform runtime updates of the configuration of a service. In the Jgroup/ARM implementation [6], updates are restricted to changing the redundancy level attributes. Additionally, the management client may subscribe to events associated with one or more object groups. These events are passed 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 experimental evaluations using the original centralized ARM implementation as reported in [16, 7].

4.2 Replication Management Policies

Policy-based management [9] 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. Policy-based management architectures are often organized using two key abstractions [17]: a manager component and a set of managed resources controlled by the manager. Typically the manager is a centralized entity and is called the *policy decision point* (PDP), and the managed resources are called *policy enforcement point* (PEP). In the DARM architecture, the decision and enforcement points can easily be co-located on the managed resources enabling implementation of decentralized policies.

In DARM two separate policy types are defined to support the autonomy properties: (1) the *distribution policy* and (2) the *fault treatment policy*, both of which are specific to each deployed service. Alternative policies can be added to the system. The policies used here 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. 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 fault treatment policy. The distribution policy in DARM is similar to

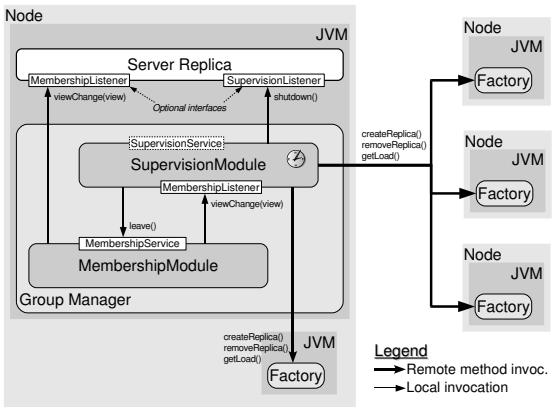

Fig. 3. The Distributed ARM architecture.

the one used in ARM [7]: DisperseOnSites 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, the least loaded nodes in each site is selected. The same node may host multiple distinct service types. The primary objective of this policy is to ensure available replicas in all *likely* network partition that may arise. Secondly, it will load balance the replica placements evenly over each site. A distribution policy algorithm is given in [15].

Each service is associated with a fault treatment policy, whose primary purpose is to describe how the redundancy level of the service should be maintained. Two inputs are needed: (1) the target environment, and (2) the initial (R_{init}) and minimal (R_{min}) redundancy level of the service. The current fault treatment policy called KeepMinimalInPartition has the objective to maintain service availability in all partitions, i.e. to maintain R_{min} in each partition that may arise (see [15] for details). Alternative policies can easily be defined, e.g. to maintain R_{min} in a primary partition only.

Policy specifications are part of a sophisticated configuration mechanism, based on XML, enabling administrators to specify (1) the target environment, (2) deployment-specific parameters, and (3) service-specific descriptors.

4.3 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 availability status of a node (factory) can also be checked by invoking the isAvailable() method on the factory. This method is used by the distribution policy to determine if a node is available before selecting it to host a replica, whereas the getLoad() method obtains load information about the node.

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,

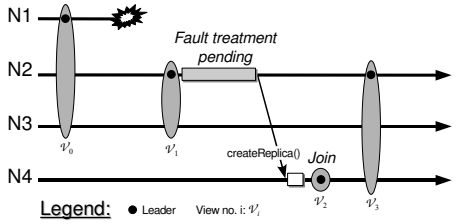

Fig. 4. An example crash failure-recovery sequence where $R_{\min} := 3$.

the factory can simply be restarted after a node repair and support new replicas. Furthermore, object factories are not replicated and thus do not depend on any Jgroup or DARM services. Replicas run in separate JVMs, to avoid that a misbehaving replica causes the failure of other replicas within a common JVM.

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. The purpose of DARM is (1) to distribute service replicas in the target environment, to (best) meet the operational policies for all services (see Section 4.2); (2) to collect and analyze information about failures, and (3) to recover from them.

Fig. 3 shows the Distributed ARM architecture. The architecture follows an *event-driven* design in that events are reported to the supervision protocol module rather than having to continuously probe individual components. Hence, the supervision module exploit synergies with existing Jgroup modules, the membership module in particular. Applications that wish to support fault treatment must include the supervision module in its protocol composition.

The supervision module operates on group-level events, also called *view change* events received from the membership module. A group leader (associated with each application service) is responsible for detecting failures and activating fault treatment actions (see Section 4.6). In this way, the failure detection costs incurred by the PGMS are shared with other modules that need membership information. The group leader is elected implicitly by the total ordering of the group members, hence there is no additional cost of leader election. If the group leader fails, a new group leader is implicitly elected by the total ordering of group members in the new view installed by the group. 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 fault treatment can be activated. This can occur if multiple nodes fail in rapid succession, or if the network partitions, e.g., leaving only one replica in a partition whom fails shortly thereafter. A solution to this could be to have the various groups monitor each other using a *lease renew* mechanism similar to the approach taken in the centralized ARM [6] architecture, where the centralized manager tracks all groups.

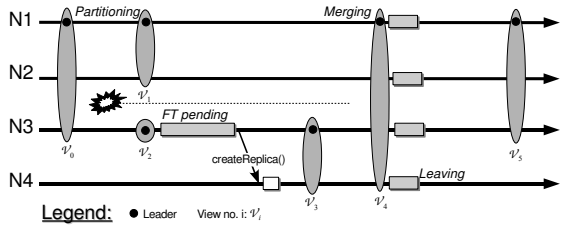

Fig. 5. A sample network partition failure-recovery scenario where $R_{\text{init}} := 3$ and $R_{\text{min}} := 2$. The partition separates nodes $\{N1,N2\}$ from $\{N3,N4\}$.

Both tracking mechanisms can be managed by supervision modules. View changes are received by the supervision module of all replicas in the group, but only the group leader activates the fault treatment action, e.g. to replace a failed replica or remove an excessive replica, as discussed in Section 4.6 and 4.5.

An example of a common failure-recovery sequence is shown in Fig. 4, in which node $N1$ fails, followed by a recovery action causing the supervision module to install a replacement replica at node $N4$. In the centralized ARM implementation [6], the recovery action was performed by a centralized RM, which would have a complete view of all installed applications within the target environment. Recomputing the replica allocations in a distributed manner offers a considerable challenge.

4.5 The Remove Policy

The supervision module may optionally be configured with a remove policy to account for any excessive replicas that may be installed. The reason for the presence of excessive replicas is that during a partitioning, a fault treatment action 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 fault treatment policy.

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 (remove policy delay), 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 remove policy delay. The choice of which replicas should leave is made deterministically based on the view composition, enabling decentralized removal. This mechanism is shown in Fig. 5, 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.

4.6 Failure Recovery

The supervision module handles failure recovery for its associated application, as follows: (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 fault treatment and distribution policies, respectively.

The supervision module receive events and maintains the state necessary to determine the need for recovery, according to the fault treatment policy of the associated service. Each instance of the supervision module maintains a *Service Monitor* (SM) timer for its associated application service. The purpose of the SM timer is to delay the activation of a fault treatment action until the current membership has stabilized. Moreover, the recovery algorithm is invoked if the SM timer expires. To prevent activating unnecessary recovery actions, the SM timer must either be rescheduled or canceled before it expires. The SM status is updated by means of ViewChange events associated with the service: If the received view $\mathcal{V}$ is such that $|\mathcal{V}| \geq R_{\min}$, the SM timer is canceled, otherwise the SM is rescheduled, pending additional view changes. 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 two primitive abstractions: *restart* and *relocation*. Restart is used when the node's factory remains available, while relocation is used if the node is considered unavailable. The actual installation of replacement replicas is done using the distribution policy.

5 Related Work

Fault treatment techniques similar to those provided by DARM were first introduced in the Delta-4 project [12]. 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 most prominent Java-based fault tolerance frameworks [4, 1] offers mechanisms similar to those of DARM, 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 [14] specify certain mechanisms such as a generic factory, a centralized RM and a fault monitoring architecture, that can be used to implement a centralized management facilities similar to ARM [7, 6]. DARM as presented in this paper enable distributed fault treatment. Furthermore, the standard makes explicit assumptions that the system is not partitionable, a unique feature of Jgroup/DARM. Eternal [5] is probably the most complete implementation of the FT CORBA standard, and uses a centralized RM. It supports distributing replicas across the system, however, the exact workings of their replica placement approach has not been documented. DOORS [18] is a framework that provides a partial FT CORBA implementation, focusing on passive replication. It uses a centralized RM to handle replica placement and migration in response to failures. The RM 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 RM is unavailable. Also the MEAD [19] 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 as in DARM. AQuA [13] 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 DARM 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 centralized dependability manager. The ARM [20, 7, 6] framework uses a centralized RM to handle distribution of replicas (replica placement), as well as fault treatment of both network partition failures and crash failures. The ARM framework uses the open group model, enabling object groups to report failure events to the centralized manager without becoming a member of the RM group.

DARM essentially supports the same features as ARM, but instead uses a distributed algorithm to perform replica placement according to a distribution policy. This enables each group to handle their own allocation of replicas to the sites and nodes in the target environment. Thereby, eliminating the need for a centralized RM that maintains global information about all object groups in the system, which is required in all frameworks discussed above. Furthermore, 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 DARM to perform recovery actions based on predefined and configurable policies enabling self-healing and self-configuration properties, ultimately providing autonomous fault treatment.

6 Conclusions and Future Work

This paper has presented an architecture for distributed autonomous replication management based on our previous experiences with building a centralized ARM architecture [6]. The new architecture enables seamless self-healing of dependable applications through a distributed fault treatment policy implemented in the protocol modules associated with applications. There are still a few open issues in our system; e.g. how to cope with multiple applications recovering simultaneous, which may result in several new replicas being allocated to the same *least loaded* node, causing the node to become highly overloaded. This is an artifact of our distributed approach. Once the implementation has been completed, we intend to perform elaborate experimental evaluations similar to our previous work on failure recovery measurements [7, 16, 6]. That is, the injection of multiple nearly-coincident node and network failures to test the failure-recovery success rate of our system and to iron out any design and implementation flaws.

Acknowledgments. The author wish to thank Alberto Montresor and Bjarne Helvik for valuable comments on this work.

References

1. Amir, Y., Danilov, C., Stanton, J.: A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In: Proc. Int. Conf. on Dependable Systems and Networks, New York (2000)
2. Montresor, A.: System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems. PhD thesis, Dept. of Computer Science, University of Bologna (2000)
3. Felber, P., Guerraoui, R., Schiper, A.: The Implementation of a CORBA Object Group Service. Theory and Practice of Object Systems **4** (1998) 93–105
4. Ban, B.: JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University (1998)
5. Narasimhan, P., et al.: Eternal - a Component-Based Framework for Transparent Fault-Tolerant CORBA. Softw., Pract. Exper. **32** (2002) 771–788
6. Meling, H.: Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation. PhD thesis, Norwegian University of Science and Technology, Dept. of Telematics (2006)
7. Meling, H., Montresor, A., Helvik, B.E., Babaoğlu, Ö.: Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management. Technical Report No. 11, University of Stavanger (2006)
8. Chockler, G.V., Keidar, I., Vitenberg, R.: Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys **33** (2001) 1–43
9. Sloman, M.: Policy driven management for distributed systems. Journal of Network and Systems Management **2** (1994)
10. Murch, R.: Autonomic Computing. On Demand Series. IBM Press (2004)
11. Solarski, M., Meling, H.: Towards Upgrading Actively Replicated Servers on-the-fly. In: Proc. Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC 2002, Oxford, England (2002)
12. Powell, D.: Distributed Fault Tolerance: Lessons from Delta-4. IEEE Micro (1994) 36–47
13. Ren, Y., et al.: AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. IEEE Trans. Comput. **52** (2003) 31–50
14. Object Management Group: Fault Tolerant CORBA Specification. OMG Document ptc/00-04-04 (2000)
15. Meling, H.: An Architecture for Self-healing Autonomous Object Groups. Technical Report No. 21, University of Stavanger (2007)
16. Helvik, B.E., Meling, H., Montresor, A.: An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In: Proc. Fifth European Dependable Computing Conference. (2005)
17. Agrawal, D., Lee, K.W., Lobo, J.: Policy-Based Management of Networked Computing Systems. IEEE Commun. Mag. **43** (2005) 69–75
18. Natarajan, B., et al.: DOORS: Towards High-performance Fault Tolerant CORBA. In: Proc. 2nd Int. Symp. Distributed Objects and Applications. (2000)
19. Reverte, C.F., Narasimhan, P.: Decentralized Resource Management and Fault-Tolerance for Distributed CORBA Applications. In: Proc. 9th Int. Workshop on Object-Oriented Real-Time Dependable Systems. (2003)
20. Meling, H., Helvik, B.E.: ARM: Autonomous Replication Management in Jgroup. In: Proc. 4th European Research Seminar on Advances in Distributed Systems, Bertinoro, Italy (2001)