A Distributed Approach to Autonomous Fault Treatment in Spread

Hein Meling and Joakim L. Gilje
Department of Electrical Engineering and Computer Science
University of Stavanger, 4036 Stavanger, Norway
Email: hein.meling@uis.no | jgilje@jgilje.net

Abstract

This paper presents the design and implementation of the Distributed Autonomous Replication Management (DARM) framework built on top of the Spread group communication system. The objective of DARM is to improve the dependability characteristics of systems through a fault treatment mechanism. Unlike many existing fault tolerance frameworks, DARM focuses on deployment and operational aspects, where the gain in terms of improved dependability is likely to be the greatest.

DARM is novel in that recovery decisions are distributed *to each individual group deployed in the system, eliminating the need for a centralized manager with global information about all groups. This scheme allows groups to perform fault treatment on themselves. A group leader in each group is responsible for fault treatment by means of replacing failed group members; the approach also tolerates failure of the group leader. The advantages of the distributed approach is: (i) no need to maintain globally centralized information about all groups which is costly and limits scalability, (ii) reduced infrastructure complexity, and (iii) less communication overhead. We evaluate the approach experimentally to validate its fault handling capability; the recovery performance of a system deployed in a local area network is evaluated. The results show that applications can recover to their initial system configuration in a very short period of time.*

1. Introduction

A common technique used to improve the dependability characteristics of systems is to *replicate* critical system components whereby their functions 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 [2, 11, 5, 3, 15] 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. In traditional fault tolerance frameworks, one relies on the system administrator (or the application) being able to replace failed replicas before they have all been exhausted, which would otherwise cause a system failure.

In this paper we present a novel architecture for *distributed* autonomous replication management (DARM), aimed at improving the dependability characteristics 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. Autonomous fault treatment in DARM is accomplished by mechanisms for localizing failures and system reconfiguration. Reconfiguration is handled by DARM without any human intervention, and according to application-specific dependability requirements. Hence, the cost of developing, deploying and managing highly available applications based on Spread/DARM can be significantly reduced.

The architecture builds on our previous experience [8, 10] with developing a prototype that extends the Jgroup [11] object group system with fault treatment capabilities. The new architecture is implemented on top of the Spread group communication system (GCS) [2] and relies on a distributed approach for replica distribution (placement), thereby eliminating the need for a centralized replication management infrastructure used in our previous work and also in other related works [18, 19, 17].

Distributed replica placement enables deployed applications (implemented as groups) to implement autonomic features such as self-healing by performing fault treatment on themselves, rather than having a (replicated) centralized component monitoring each deployed application. The

fault treatment mechanism represents a non-functional aspect and to separate it from application concerns it is implemented as a separate component (a factory) and a small library that can easily be linked with any Spread application. DARM provides automated mechanisms for performing management activities such as replica distribution among sites and nodes, and recovery from replica failures. DARM is also able to restore a given redundancy level in the event of a network disconnection.

DARM offers *self-healing* [12], where failure scenarios are discovered and handled through recovery actions with the objective to minimize the period of reduced failure resilience; *self-configuration* [12], where objects are relocated/removed to adapt to uncontrolled changes such as failure/merge scenarios, or controlled changes such as scheduled maintenance (e.g. software and OS upgrades).

Organization: Section 2 provides some background and in Section 3 presents the DARM architecture. In Section 4 an experimental evaluation is provided for two separate Spread configurations. Section 5 compares DARM with related work and Section 6 concludes.

2. Background 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 clients and servers. The set of nodes, N, that may host application services and infrastructure services, in the form of *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 DARM implementation presented in this paper uses the Spread GCS [2]. A GCS enables processes (clients in Spread terminology) with a common interest to join a shared group for communications purposes, and can guarantee certain reliability properties (e.g. total order delivery) for the communication services it provides. Spread also provides a group membership service. Therefore we assume the same system and failure model as the one assumed by Spread. Long-lasting network disconnections may occur in which certain communication failure scenarios may disrupt communication between multiple sets of replicas forming *partitions*. Replicas in the same partition can communicate, whereas they cannot communicate with replicas in other partitions. When communication between partitions is re-established, we say that they *merge*. Replicas may also *crash*, whereby they simply stop generating output. The Spread GCS follows the open group model, meaning that a client is not required to join a specific group before sending messages to that group. This property is utilized by DARM.

Spread consists of two main parts, a daemon and the Spread library, `libspread`, as illustrated in Figure 1. The Spread daemon is used to forward messages between group members (Spread clients) whereas the Spread library is used for developing Spread clients. The library includes functions for connecting to the daemon as well as communicating with other Spread clients.

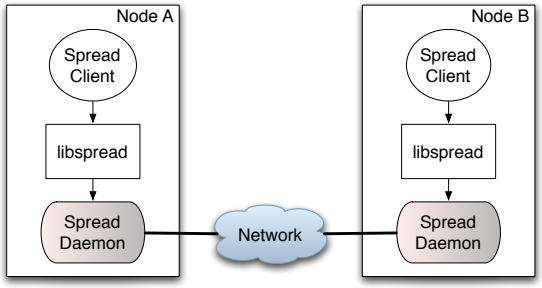

Figure 1. Spread client-daemon relationship.

The Spread daemon is not required to be running on each node in a distributed system as Figure 1 shows, however it is a typical configuration when using Spread for replication. Hence in this paper we assume that each node is running a Spread daemon. The daemon uses a configuration file to discover other daemons in the system. A Spread network may span several network sites. Within each site, the daemon will communicate with its neighboring daemons using either broadcast or multicast traffic depending on the capabilities of the LAN environment. However, unicast is used for communication between daemons in different sites. For the latter case, a master daemon is selected to mediate messages on behalf of all daemons within a site.

3. Architecture of DARM

Figure 2 shows a simple deployment scenario where three applications, A, B, and C are replicated and allocated to various nodes in the system. Notice that a single node may host several replicas of different types.

To support DARM, all nodes must be running a factory (in addition to the Spread daemon), and the application must be linked with the DARM library, called `libdarm`. The factory is responsible for creating replicas on behalf of `libdarm` when DARM requires this. As Figure 3 show, `libdarm` is intended to replace `libspread` when viewed from a Spread client perspective. That is, `libdarm` actually wraps around `libspread` intercepting connection and membership messages. In addition, `libdarm` provide several functions needed to dynamically configure the policies for autonomous operation of each group.

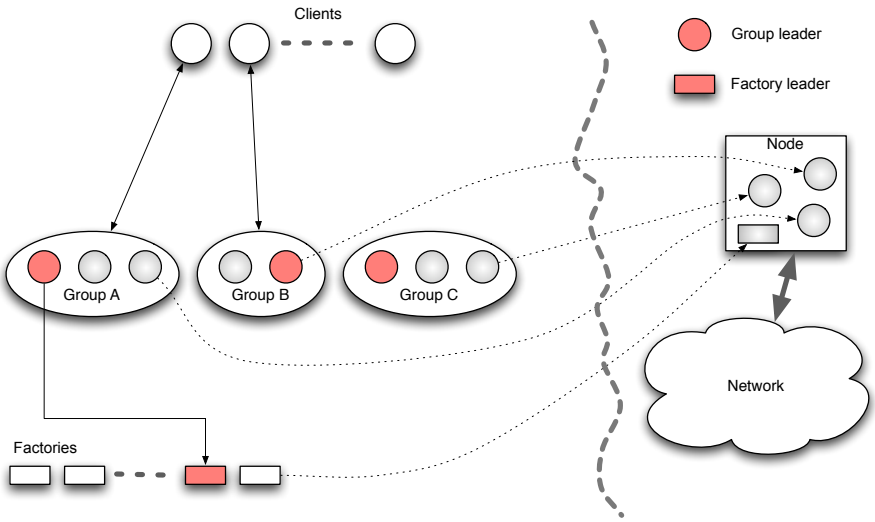

Figure 2. A simple deployment scenario with three DARM applications.

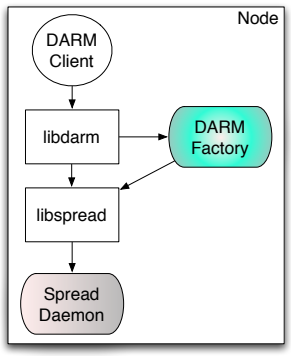

Figure 3. DARM client dependencies.

The DARM library can be considered an agent acting on behalf of DARM; it is collocated with each replica and is responsible for collecting and analyzing failure information obtained from view change events generated by the Spread group membership service, and to initiate on-demand system reconfiguration according to the configured policies. It is also responsible for removal of excessive replicas. Overall, the interactions among the components shown in Figure 3 enable the DARM agent to make proper recovery decisions, and allocate replicas to suitable nodes in the target environment.

Figure 4 shows an example of a common failure-recovery sequence, in which node $N1$ fails, followed by a recovery action causing DARM to install a replacement replica at node $N4$. In the centralized ARM implementation [8], the recovery action was performed by a centralized replication manager (RM), which would have a complete view of all installed applications within the target environment. In DARM, the factory is responsible for recomputing the replica allocations in response to a failure, based on policy requirements and the current load on the various nodes.

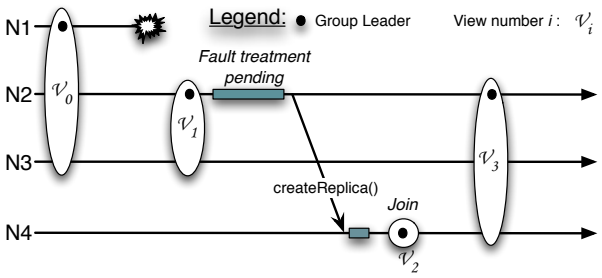

Figure 4. A crash failure-recovery sequence.

3.1. The Factory

The main purpose of the factories is to enable installation of service replicas (Spread/DARM clients) on demand. In addition, the factories keep track of the load on the local node, and also the availability status of nodes. This information is used to determine the best node on which to place replicas.

The factories all join a common factory group immediately after its initialization and thus depend on the Spread daemon. Hence the factory must be started after the daemon, but before starting any replicated services. The factories will send and receive factory internal messages using the factory group, in addition to receiving replica create re-

quests from `libdarm`. The factory does not maintain any state that needs to be preserved in case of factory failures. Thus the factory can simply be restarted after a node repair and support new replicas.

A factory leader is selected amongst the members of the factory using the total ordering of members received from the group membership service, i.e. the first member in the list is selected. Every factory keeps track of which factories are running and at which site they physically belong. The factory leader uses this information along with load information when it is asked to create a replica. The active configuration is kept at each factory since every factory has to be prepared to act as a leader.

As Figure 5 shows, the factory group receives create replica requests from `libdarm` clients; all factory replicas receive such requests, however only the factory leader is involved in deciding on which site and node a new replica should be created. When the factory leader has decided which node should start a new replica, it will send a private message to the factory running at that node to complete the request from `libdarm`. The factory leader uses a replica placement policy (see Section 3.2) to decide on which node a new replica should be created. To help the factory make informed decisions on where to place a new replica, `libdarm` attaches information about its associated group's current configuration when sending a request to the factory group. This information includes a list of nodes currently running a replica (obtained from the current view), and the number of replicas in each site.

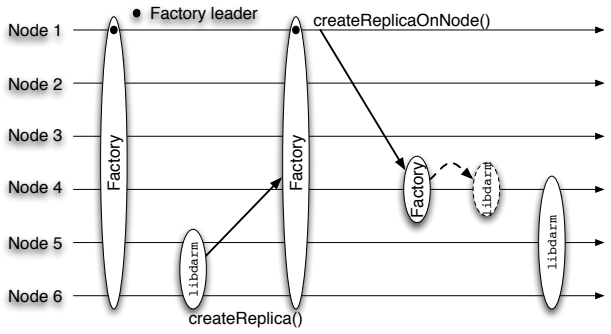

Figure 5. Create replica request.

3.2. The Replica Placement Policy

In DARM several policy types are defined to support the autonomy properties: (1) the *replica placement policy*, (2) the *remove policy*, and (3) the *fault treatment policy*, all of which are specific to each deployed service. Alternative policies can be added to the system.

The purpose of a replica placement 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 configurable and is determined at runtime. The placement policy in DARM is as follows:

1. Find the site with the least number of replicas of the given type.
2. Ignoring nodes already running the service, find the node in the candidate site with the least load.

This placement policy will avoid collocating two replicas of the same service on the same node, while at the same time it will disperse the replicas evenly on the available sites. In addition, the least loaded nodes are selected in each site. The same node may host multiple distinct service types. The primary objective of this policy is to ensure available replicas in all *likely* network partitions that may arise. Secondly, it will load balance the replica placements evenly over each site. The placement policy is implemented in the factory.

3.3. The Replica Remove Policy

DARM may optionally be configured with a remove policy to deal with any excessive replicas that may have been 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 dependability requirements of the application.

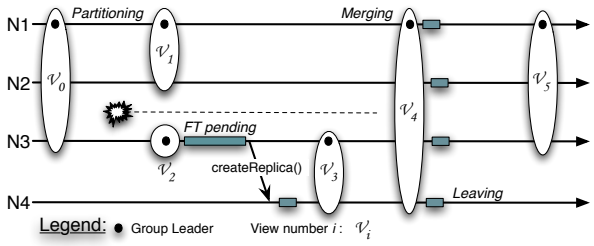

Figure 6. A sample network partition failure-recovery scenario where $R_{\max} := 3$ and $R_{\min} := 2$. The partition separates nodes $\{N1,N2\}$ from $\{N3,N4\}$.

Let $\mathcal{V}$ denote a view and $|\mathcal{V}|$ its size. If $|\mathcal{V}|$ exceeds the maximum redundancy level $R_{\max}$ for a duration longer than a configurable time threshold T_{rm}, `libdarm` will request one excessive replica to leave the group. If more than one replica needs to be removed, each remove is separated

by T_{rm} seconds. The choice of which replicas should leave is made deterministically based on the view composition, enabling decentralized removal. This mechanism is shown in Figure 6, where the dashed timelines indicate the duration of the network partition. After merging, `libdarm` detects one excessive replica, and elects $N4$ to leave the group. The remove policy is implemented in `libdarm`.

3.4. The Fault Treatment Policy

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 maximum (R_{max}) 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. A recovery delay, T_{rec}, is used to avoid premature activation of the fault treatment mechanism. Alternative policies can easily be defined, e.g. to maintain R_{min} in a primary partition only. The fault treatment policy is implemented in `libdarm` using the factory to create replacement replicas on-demand.

3.5. The DARM Library

A Spread/DARM client is linked with `libdarm` to enable DARM functionality. It is primarily the leader replica that performs operations using `libdarm`, e.g. to initiate recovery. Only a single instance of an application service needs to be started manually (or through a management interface) to bootstrap a replicated service. That is, once an application has been started, `libdarm` will request that the configured number of replicas be started using the factories. This enables automatic deployment of replicas within the configured distributed system.

The library is designed to be compatible with the Spread 4.0 C API. That means all functions that are available as SP_* functions in `libspread` have an identical function called DARM_* with the exact same function signature and return values. This obviously requires minor modifications and recompilation of existing Spread clients to take advantage of `libdarm`. This is due to the flat namespace in the C programming language used to implement DARM. There are ways to circumvent this drawback [13], but this has not been our priority.

Most functions with a DARM-prefix are just forwarded to the corresponding SP-function. The functions that do DARM related processing are DARM_connect() and DARM_receive(); these are discussed below. In addition, several functions are provide for runtime configuration, e.g. DARM_set_minimum(), DARM_set_maximum() and DARM_set_recovery_delay(). These functions are used to configure DARM to maintain a minimum/maximum number of group members and to set the recovery delay.

For DARM_connect(), `libdarm` will intercept the connection call to verify and finalize the runtime configuration of DARM, and also joining the DARM private group associated with the current application. When connecting, client requests to override membership messages will be ignored, since membership messages are essential for monitoring the current state of the group and is required by DARM.

The DARM_receive() function will call SP_receive(), and after receiving a message, DARM_receive() will determine if the message belongs to the DARM private group or is an application message. Application messages are simply forwarded to the DARM client, while DARM messages are handled within `libdarm`. To avoid returning control to the client without any message, `libdarm` will continue to call SP_receive() until an application message is received, processing any intermediate DARM messages internally.

Membership messages delivered to the DARM private group are used to decide whether fault treatment is needed. For example, if a membership message is a *join* message, the replica count is increased by one. The DARM leader will either ask the factory group to create a new replica if the replica count is lower than the minimum threshold, or ask a member of its own group to self-terminate if the replica count is above the maximum threshold. The DARM library will elect a leader using the same approach as used by the factories.

4. Experimental Evaluation

The recovery performance of Spread/DARM is evaluated experimentally with respect to network disconnection events. In this paper focus is on the duration required to recover to the configured minimal redundancy level, R_{min}, in each network partition that arise in the fault injection campaign. In the following, we present the target system for the experiments followed by our findings.

4.1. Target System

Figure 7 shows the target system for our measurements. It consists of three sites (network segments) denoted x, y and z, all of which are on the same LAN[1]. Each site has three nodes denoted $x1, x2, x3$, etc. Hence, a total of nine nodes are used, each of which run the Spread daemon and the DARM factory. In addition, an external node runs a *fault injector* application using a modified version of the Spread tool `spmonitor` to inject network disconnections on the nodes in the system to emulate partition failures.

[1] Although the nodes are on the same LAN, Spread has the ability to simulate network partitioning faults using the `spmonitor` tool.

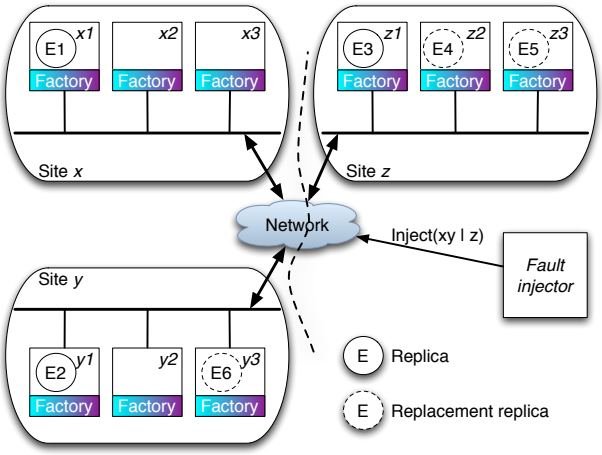

Figure 7. Target system for the experiments, also illustrating a network partition scenario.

A simple evaluation service, that sends a small group internal message every second, is deployed in the target system following the policy described in Section 3.2, with $R_{\text{max}} := 4$ and $R_{\text{min}} := 3$, i.e. the service will have three replicas initially. Furthermore, the DARM fault treatment policy (see Section 3.4) will try to maintain at least three replicas in each partition. Finally, if $|\mathcal{V}| > R_{\text{max}}$, `libdarm` will determine if the local replica should to be removed due to excessive replicas. DARM will continue to remove replicas, one at a time, until the expression evaluates to false. The recovery and remove delays are both set to 2 seconds.

The fault injector application randomly selects the site to isolate from the other two sites and invokes the modified `spmonitor`, thereby creating a network partition. Then, once all partitions have recovered to their specified minimal redundancy level, the fault injector will inject a reconnection event to merge the network partitions into a fully connected network.

4.2. Experimental Results

In the following, DARM is tested in two different Spread configurations, denoted *default* and *fast*. The fast Spread configuration requires recompilation after adjusting various timeouts that are hardcoded into Spread; the fast timeout parameters are given in [1]. As the experiments shows, these changes have a significant impact on the recovery performance of DARM. In both configurations, 200 iterations with partition/merge injection events were performed. All relevant events such as partition injection, merge injection, replica creation/removal and membership changes were logged for each of the sites. This gives us 600 measurements for merge events, since each site measures the delays for each iteration. For the partition measurements there are fewer measurements since recovery is not always needed when a partition occurs. This can happen when the partition is injected on a fully connected network with $|\mathcal{V}| = 4$ replicas; this may result in one partition already containing three replicas and thus no recovery is needed.

Results for the fast Spread configuration is shown in Figure 8. The plots in the top row shows the time of the various measured events in the recovery cycle after the injection of a network partition (which occur at time 0). The plot to the left shows the results for the partition with two live replicas, where one new replica is created to recover to the minimal redundancy level $R_{\text{min}} = 3$, whereas the plot on the right shows the results of the partition with only one live replica in need of two new replicas to reach $R_{\text{min}} = 3$. The results are sorted according to the total time needed to reach the *final view* (fully recovered). Hence, the right-most curves (in the top row plots) are the empirical cumulative distribution function (CDF) of this time. The proceeding curves show their relative contribution to the recovery delay. The *partition detection* curve shows the time it takes to detect that a network partition has occurred from the time of the actual injection; this is the part where most of the variability is introduced. The time to create a new replica is nearly deterministic and is due to the 2 second recovery delay. Creating two replicas introduces an additional 2 second delay as shown in the right-most plot. These delays are used to avoid that DARM triggers unnecessary recovery actions due to transient disconnections. The plots in the middle row of Figure 8 show the density estimate for the same observations. Note that the time between the creation of a new replica and the installation of the final view is very small, hence the view agreement delay is not shown in the plots.

The bottom left plot in Figure 8 shows the time of the various events in the remove cycle after the injection of a network merge. As for the partition case the results are sorted according to the last curve, the total time needed to reach the final *merged view*. The *merge detection* curve is the time that Spread takes to report that the network has been reconnected. The merge event will render the network fully connected with $|\mathcal{V}| = 6$ replicas; three in each of the previous partitions. Hence, DARM will remove two of the replicas to reach the $R_{\text{max}} = 4$ threshold. Each removal is separated by the 2 second remove delay. The bottom right plot shows the density estimates for the same observations.

The same experiments were also performed for the default Spread configuration as shown in Figure 9. Like in the fast configuration, most of the variability is introduced in the partition/merge detection phase. The replica create and final view are strongly correlated to the partition detection delay. Another thing to note is that the default configuration has a significantly longer detection time both for partition

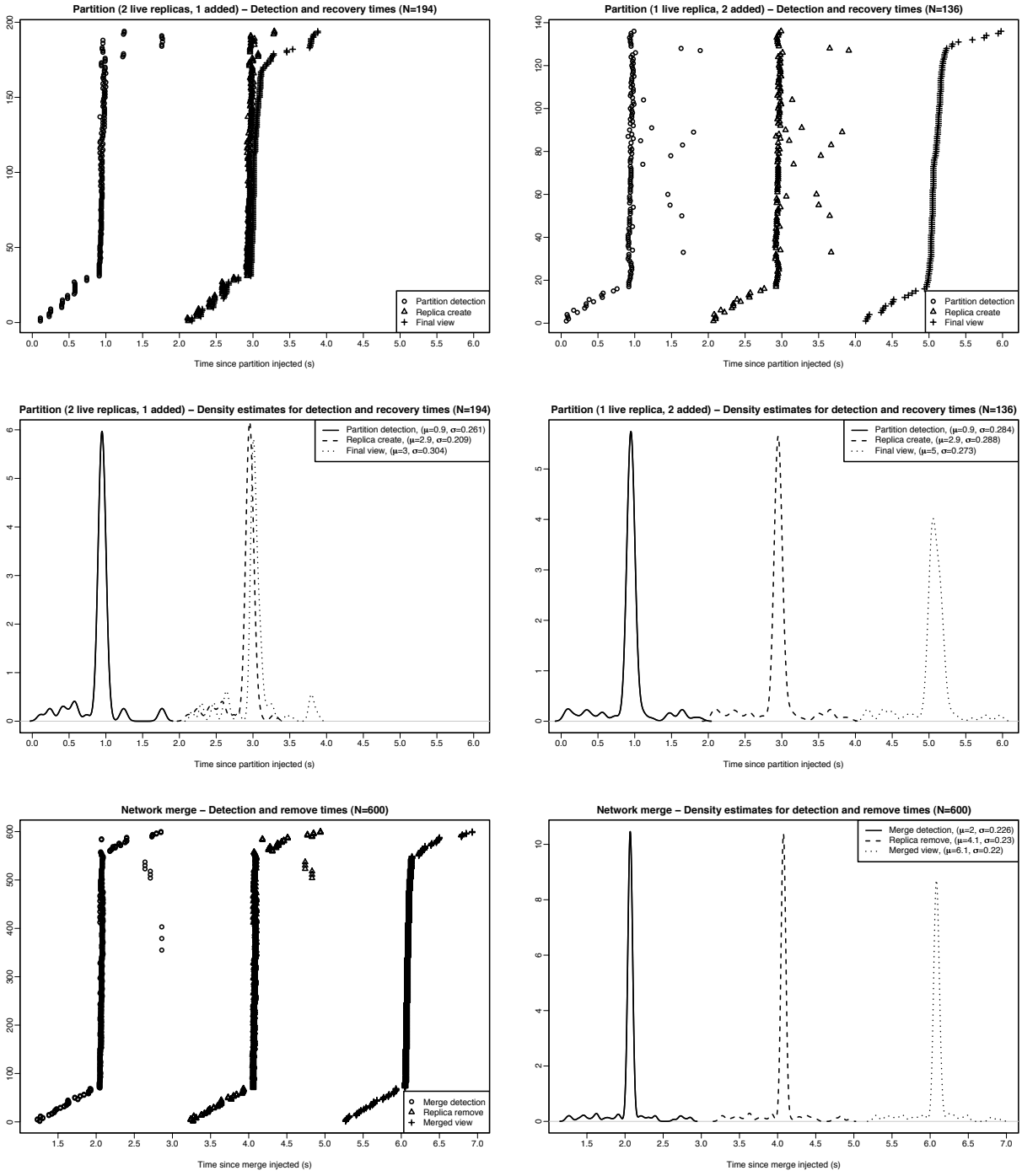

Figure 8. Fast Spread configuration

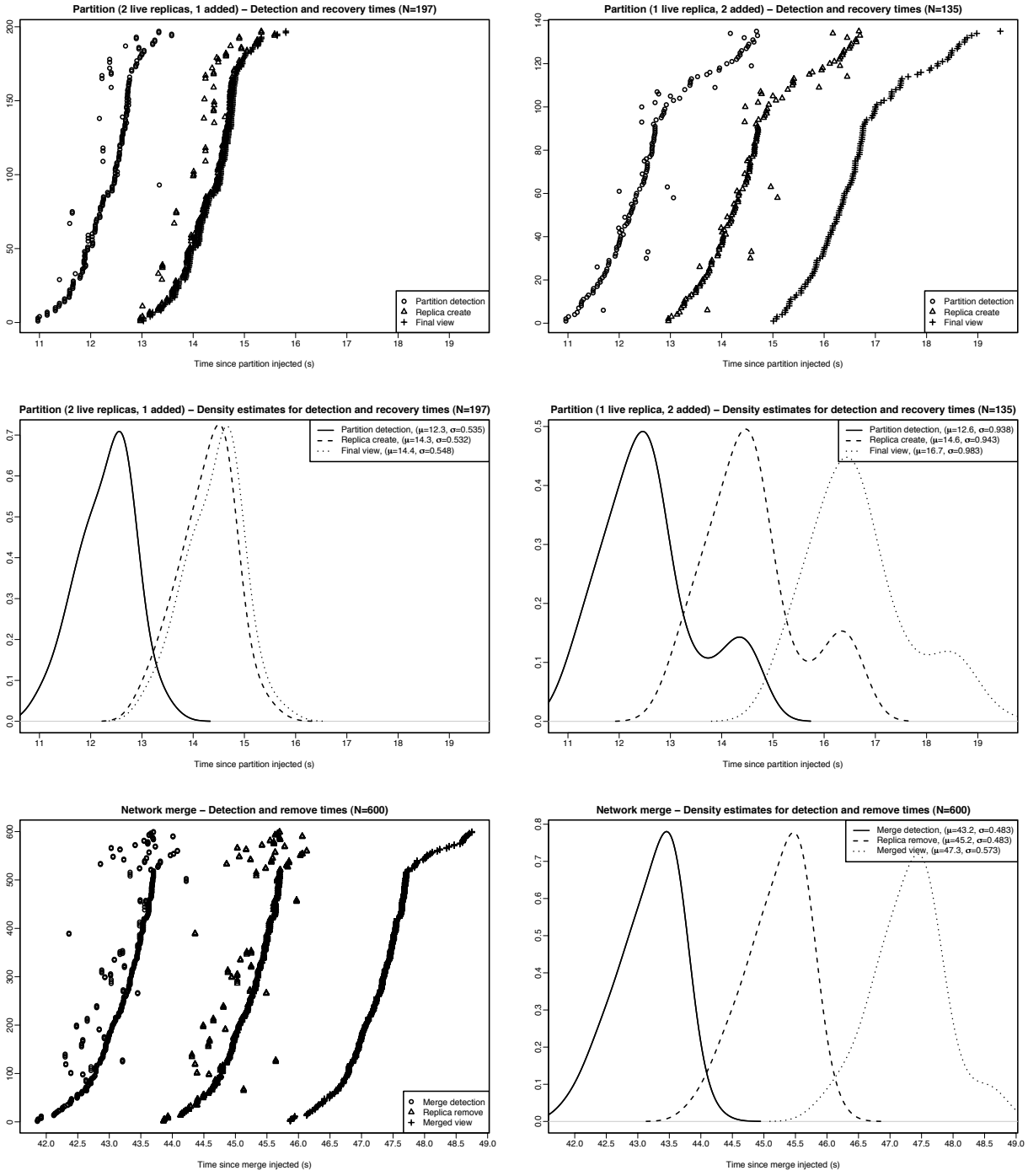

Figure 9. Default Spread configuration

and merge; these delays are due to Spread being designed for WAN configurations mandating longer timeouts. The mean (μ) and standard deviation (σ) for the various events are provided in the legend of the density plots. Although the default configuration has a very long detection time, the DARM recovery delay is fixed to 2 seconds. In the default configuration, Spread also has a much larger variance in its detection phases, whereas the fast configuration has very little variance. This is illustrated by the narrow density curves for the fast configuration, as opposed to the default configuration where the density curves are much wider.

In summary, the experimental evaluation has demonstrated that recovery from network partition failures by ensuring sufficient redundancy levels at all times is easily handled by DARM in a robust and timely manner.

5. Related Work

Fault treatment techniques were first introduced in the Delta-4 project [18]. 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. Cristian [4] sketched a service availability concept where services are sorted according to their importance (vital vs non-vital), and allocated to processors with the objective to maintain availability of the most important services should a processor fail. Failed vital services were to be restarted on another processor. Unlike recent approaches, Cristian made the assumption that services were not replicated; also no implementation were presented.

None of the most prominent Java-based fault tolerance frameworks [3, 2] 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 [17] 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. DARM as presented in this paper enable distributed fault treatment. Eternal [14] 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 to our knowledge not been published. DOORS [16] 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 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 [20] 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 reboot or replace the node, since there is no support for relocating the replicas to other nodes. AQuA [19] 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. 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. ARM [10, 8] 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 was first proposed as a concept in [9], and has since been adapted and implemented using the Spread GCS as presented in this paper. Additional technical details can be found in [6]. DARM essentially supports the same features as ARM, but instead uses a distributed approach to perform replica placement. 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 groups in the system, which is required in all the frameworks discussed above. Furthermore, none of the other frameworks that support recovery, except ARM, has been rigorously evaluated experimentally with respect to fault treatment. Compared to Jgroup/ARM, the recovery performance of Spread/DARM is about 2.5 seconds faster (fast config.) The main difference lies in the view agreement delay which is significant in Jgroup even for small groups. Although neither system has been tested in scenarios with many services and large groups, we believe that Spread would outperform Jgroup in such configurations.

Virtualization can also be used to support a recovery mechanism. In virtualization, multiple virtual machines (running on physical servers) can host applications with hot standby copies in other virtual machines (on distinct physical servers). These standbys can be configured to take over for a production server during outages. The advantage of virtualization is that applications are decoupled from the replication and recovery mechanisms, and developers need not consider these challenges. However, the drawback is that stateful applications must write updates to a common stable storage to ensure their availability to the standbys, making the storage component a single point of failure. With DARM, each node can host their own stable storage

and provide state transfer functions to ensure consistency across replicas.

6. Conclusions and Future Work

We have presented the design, implementation and an initial experimental evaluation of Spread/DARM. The approach taken in DARM is based on making recovery decisions within each group itself, making the groups provide seamless self-healing. The experimental results indicate that DARM is able to recover from a network partition in less than 6 seconds when only one replica remains in the partition. DARM perform recovery actions based on predefined and configurable policies enabling self-healing and self-configuration properties, ultimately providing autonomous fault treatment.

Currently, work is underway to perform even more elaborate experimental evaluations based on our previous experience with a post-stratification technique [7] to estimate system dependability characteristics. This analysis will consider both node and network failures as well as more elaborate system configurations. We are also planning to add support for live reconfiguration of the underlying sites and nodes that can support replicas, as well as dealing with group failures, where all members of a group fail before any of its members is able to react. The DARM system is available as open source from http://darm.ux.uis.no/.

Acknowledgements

This work was partially supported by a scholarship from Telenor iLabs. The authors wish to thank B. Helvik and the anonymous reviewers for useful comments on this paper.

References

[1] Very fast Spread. http://commedia.cnds.jhu.edu/pipermail/spread-users/2004-July/002114.html.

[2] 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.

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

[4] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–188, 1991.

[5] 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.

[6] J. L. Gilje. Autonomous Fault Treatment in the Spread Group Communication System. Master's thesis, University of Stavanger, June 2007.

[7] 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.

[8] 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.

[9] H. Meling. An Architecture for Self-healing Autonomous Object Groups. In *Proc. of the 4th International Conference on Autonomic and Trusted Computing*, volume 4610 of *LNCS*, pages 156–168. Springer-Verlag, July 2007.

[10] H. Meling, A. Montresor, B. E. Helvik, and Ö. Babaoğlu. Jgroup/ARM: A distributed object group platform with autonomous replication management. *Softw., Pract. Exper.*, Available online, to appear in print.

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

[12] R. Murch. *Autonomic Computing*. On Demand Series. IBM Press, 2004.

[13] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using Interceptors to Enhance CORBA. *Computer*, 32(7):62–68, 1999.

[14] 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.

[15] 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.

[16] 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.

[17] OMG. Fault Tolerant CORBA Specification. OMG Document ptc/00-04-04, Apr. 2000.

[18] D. Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, Feb. 1994.

[19] Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. AQuA: an adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1):31–50, Jan. 2003.

[20] 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.