An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM system

Bjarne E. Helvik[1], Hein Meling[2], and Alberto Montresor[3]

[1] Department of Telematics, Q2S Centre, Norwegian University of Science and Technology, O.S. Bragstadsplass 2A, N-7491 Trondheim, Norway, Email: bjarne@q2s.ntnu.no
[2] Department of Electrical and Computer Engineering, University of Stavanger, N-4036 Stavanger, Norway, Email: meling@acm.org
[3] Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy, Email: montresor@CS.UniBO.IT

Abstract. Jgroup/ARM is a middleware framework for operating dependable distributed applications based on Java. Jgroup integrates the distributed object models of Java RMI and Jini with the *object group communication* paradigm, enabling the construction of groups of replicated server objects that provide dependable services to clients. ARM provides automated mechanisms for distributing replicas to host processors and recovering from replica failures.
This paper describes an approach based on *stratified sampling* combined with *fault injections* for estimating the dependability attributes of a service deployed using the Jgroup/ARM middleware framework. A first experimental evaluation is performed focusing on a service provided by a triplicated server, and indicative predictions of various dependability attributes of the service are obtained. The evaluation shows that a very high availability and MTBF may be achieved for services based on Jgroup/ARM.

Keywords: *Fault Injection, Probabilistic Modeling and Evaluation, Measurement-based Evaluation, Failure Recovery, System Fault Tolerance, FT Middleware.*

1 Introduction

The *group communication* paradigm for the development of dependable distributed applications has received considerable attention in recent years [19, 10, 20, 1]. Middleware frameworks based on this paradigm have been integrated with modern distributed object models like Java RMI and Jini [19, 6], and are currently being deployed in web-based business applications. Assessing and evaluating the dependability characteristics of such frameworks, however, have not received an equal amount of attention.

To fill this void, this paper presents an extensive evaluation of Jgroup/ARM [19, 5, 17], one such framework. Jgroup enables the construction of dependable applications based on *groups* of replicated server objects, that cooperate in order to provide dependable services to their clients. Communication inside a group, as well as communication between clients and server objects, is based on *group method invocations* that are executed by all members of the group. ARM builds on Jgroup, providing mechanisms for the automated management of groups, by distributing replicas to host processors and recovering from replica failures.

In the evaluation, dependability attributes are predicted through a *stratified sampling* [14] approach. A series of experiments are performed; in each of them, one or more faults are injected according to an accelerated homogeneous Poisson process. The approach defines strata in terms of the number of near-coincident failure events that occur in a fault injection experiment. By near-coincident is meant failures occurring before the previous is handled. Hence, *a posteriori stratification* is performed where experiments are allocated to strata after they are carried out. This as opposed to the more common prior stratification where strata are defined before the experiment. Three strata are considered, i.e., single failures, and double and triple near-coincident

failures. The system under study is assumed to follow a crash failure semantics. For the duration of an experiment, the events of interest are monitored, and post-experiment analysis is performed to construct a single global timeline of fault injections and other relevant events. The timeline is used to compute trajectories on a predefined state machine.

Depending on the number of injected faults, each experiment is classified into one of the strata, and various statistics for the experiments are obtained. These statistical measures are then used as input to an estimator of dependability attributes, including *unavailability*, *system failure intensity* and *down times*. The approach may also be used to find periods with reduced performance due to fault handling. An additional benefit of this thorough evaluation is that the fault handling capability of Jgroup/ARM has been tested extensively, enabling the discovery of rarely occurring implementation faults of both the distributed service under study and the Jgroup/ARM framework itself.

Fault injection is a valuable and widely used means for the assessment of fault tolerant systems, see for instance [2, 3, 11]. Previously, stratified sampling has been used in combination with fault injection experiments to estimate fault tolerance coverage, as presented in [9]. Furthermore, for testing specific parts of a system, fault injection triggers has been used on a subset of the global state space [8]. These approaches are very useful in testing and evaluating specific aspects of a system. However, our objective is to perform an overall evaluation of the system and its ability to handle processor failures and hence, random injections of crash failures in a operational system and post stratification is applied.

Delta-4 provide fault treatment mechanisms similar to those of ARM [21]. Fault injections were also used in Delta-4 [4], focusing on removal of design/implementation faults in fault tolerance mechanisms. However, we are not aware of reports on the evaluation of the fault treatment mechanisms in Delta-4, comparable to those presented herein. The fault injection scheme used in this work, combined with post-experiment analysis also facilitate detection of implementation faults, and in addition allows for systematic regression testing.

The AQuA [23, 22] framework is based on CORBA and also support failure recovery. Unlike Jgroup/ARM, it does not deal with partition failures and relies on the open group model [13] which limits its scalability with respect to supporting a large number of groups. The evaluation of AQuA presented in [22] only provide the various delays involved in the recovery time. In this paper, focus is on estimating dependability attributes of services deployed through ARM.

Organization. Section 2 provides an overview of the features of the Jgroup/ARM framework relevant to this paper. Section 3 describes the target system for our measurements, while Section 4 presents the measurement setup and strategy, together with the associated estimators. Experimental results are given in Section 5, and Section 6 concludes the paper.

2 The Jgroup/ARM Middleware

2.1 Jgroup

Jgroup [5] integrates the Java RMI and Jini *distributed object models* with the *group communication* paradigm and includes numerous innovative features that make it suitable for developing modern network applications. In Jgroup, applications are based on collections of replicated *server objects* that cooperate to provide a dependable service. For increased flexibility, the group composition is allowed to vary dynamically as new servers *join* and existing servers *leave* the group, either voluntarily or by crashing. Members of the group are kept informed about the current group composition through a *group membership service* (GMS).

Communication facilities for the object group are provided by the *group method invocation service* (GMIS), that enables the execution of remote method invocations on all members of an object group. Jgroup is unique in providing such uniform object-oriented programming interface to govern *all* object interactions, including those within an object group as well as interactions with external objects (clients). Both the GMS and the GMIS have been formally specified, admitting formal reasoning about the correctness of applications based on these services [5, 19]. Due to space

constraints, however, in the following only a short informal description is provided, focusing on the properties that are needed to understand this paper.

At any time, the *membership* of a group includes those servers that are operational and have joined, but have not yet left the group. Asynchrony of the system and failures may cause each member to have a different perception of the group's current membership. The task of the GMS is to notify members about variation in the group membership. These notification are called *view changes*; we say that a node *installs* a view when such notification is delivered to it. A view consists of a list of nodes along with a unique identifier, and corresponds to the group's current composition as perceived by members included in the view.

View changes must satisfy the following requirements [5]. 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. Next, a view can be installed only after agreement is reached on its composition among the servers included in the view. Finally, GMS must guarantee that two views installed by two different servers be installed in the same order. Note that the GMS defined for Jgroup admits coexistence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications.

Group method invocations must satisfy a variant of *view synchrony*, that has proven to be an important property for reasoning about reliability in message-based systems [7]. Informally, view synchrony requires that two servers that install the same pair of consecutive views agree to complete the same set of invocations 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 invocations they have completed in the current view.

The agreement properties of Jgroup (on view composition and invocation execution) enable a server to reason about the state of other servers in the group using only local information such the history of installed views and the set of completed group method invocations.

2.2 Autonomous Replication Management (ARM)

Most existing object group systems do not include mechanisms for distributing replicas to host processors or recovering from replica failures. Yet, these mechanisms are essential for satisfying application dependability requirements such as maintaining a fixed redundancy level. The ARM framework provides a replicated dependability manager that enables the autonomic management of complex applications based on object groups [15, 17]. When installed, an object group becomes an "autonomous" entity being maintained by ARM, until it is explicitly removed. During its life, an object group provides service to clients completely decoupled from the ARM infrastructure.

ARM handles both replica distribution, according to an extensible *distribution policy*, as well as replica recovery, based on a *replication policy*. The replication policy is group-specific, and allows the creation of object groups with varying dependability requirements and recovery needs. The distribution policy is specific to each ARM deployment, and requires configuration of the set of processors on which replicas can be created.

The ARM framework consists of several components: a system-wide *replication manager* (RM), *recovery modules* deployed at each of the managed replicas, and *object factories* deployed at each of the processors. Recovery modules are responsible of forwarding view change notifications to the RM, that will interpret this information, potentially triggering group-specific actions like replica creation or removal. Together, these components form a *failure monitoring* facility, whose goal is to re-establish desired system dependability properties after failures.

Fig. 1 gives a simplified overview of a typical ARM-based deployment, and associated communication patterns. The system contains a single replicated service, named MS, that is managed by ARM. In each of the groups, a *leader* replica is elected. Every view change event generated by the group communication system is reported by the recovery module of the MS leader to the RM using notifyEvent(). The RM interprets the received event, and the leader issues a createReplica() or removeReplica() call to the object factories of a selected set of processors, depending on the distribution policy. The RM provides an external interface, composed of method createGroup() and

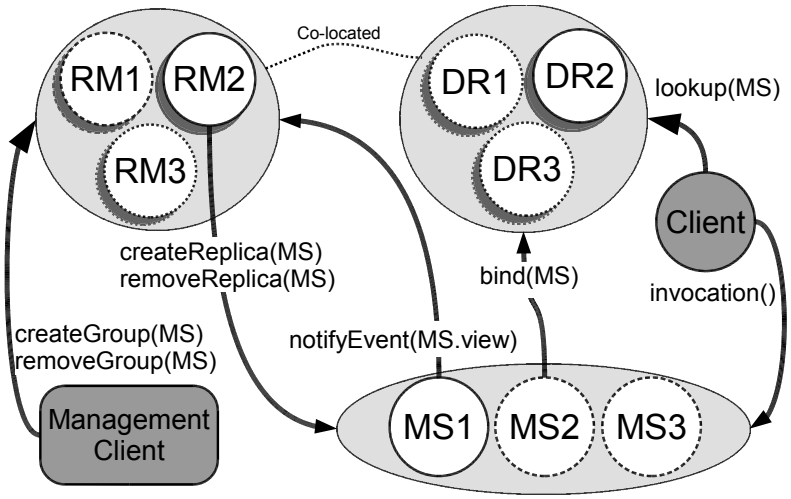


Fig. 1. The Jgroup/ARM architecture

removeGroup(), that enables the management client to start or stop services. Fig. 1 also illustrates how servers bind their reference in the dependable registry (DR) service, and how clients query this naming service to obtain information required to communicate with the server group through method invocations. Note that replicas of the RM and DR groups are co-located on the same set of nodes. For simplicity, the illustration of the target system (see Fig. 2) depict only the RM group.

In addition to the above, there is also a mechanism embedded in the recovery module for recovering from scenarios in which the whole application group has failed. This is accomplished using a lease renewal technique, requiring that the leader of each deployed group issues a renew event periodically to prevent the RM group from triggering recovery.

The RM group provides self-recovery by reusing the same mechanisms that are in place to track other applications, except for the lease renewal. Another important feature of ARM is its ability to handle multiple concurrent failures, even failures of the ARM framework itself, as long as at least one RM replica remains.

3 Target System

Fig. 2 shows the target system for our measurements. It consists of a cluster with a total of $n = 8$ identical processors, initially hosting a single server replica as shown. In the experiments, ARM uses a distribution policy that will avoid co-locating two replicas of the same type, and at the same time it will try to keep the replica count per processor to a minimum. Different services may share the same processor.

The ARM infrastructure (i.e., the RM group) is located on processors (1-3). Processors 5-7 host the *monitored service* (MS), while processors 4 and 8 host the *additional service* (AS). The latter was added to assess ARM's ability to handle multiple concurrent failure recoveries at different groups, and to provide a more realistic scenario. Finally, an external machine hosts the *experiment engine* that is used to run the experiments; for a description of the experiment engine, please refer to Section 4.2. The replication policy for all the deployed services requires that ARM tries to maintain a fixed redundancy level (RM:=3, MS:=3, AS:=2), with the RM group being at least as fault-tolerant as the remaining components of the system.

The measurement engine enables the simultaneous observations of all services in the target system, including the ARM infrastructure. In the following, however, we will focus our attention on the MS service, that constitutes our *subsystem of interest*. This subsystem will be the subject of our observations and measurements, with the aim of predicting its dependability attributes. Note that focusing on a particular subsystem of interest is for simplifying presentation. Observations of

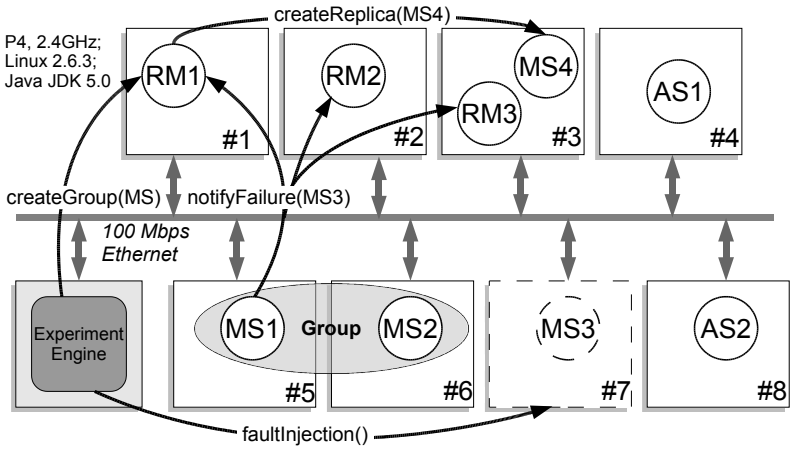


Fig. 2. Target system illustrated

several subsystems could be done simultaneously and estimates/predictions of all services and the ARM infrastructure may be obtained during the same experiment.

3.1 The State Machine

There is a set of states which we can observe and which are sufficient to determine the dependability characteristics of the service(s) regarded. Note that these are not necessarily all the operational states of the complete system, but the set of states associated with the MS service. Thus, the failure-recovery behavior of the MS service can be modeled according to the *state diagram* in Fig. 3, irrespective of the states of the ARM and AS subsystems. The state diagram is not used to control fault injections based on triggers on a subset of the global state space as in [8]; instead it is only used during offline, a posteriori analysis of fault injection experiments based on random sampling. In the analysis, the trajectories on the state model, and the time spent in each state is used together with mathematical tools to determine the dependability characteristics of the MS service.

We define a service to be *unavailable* (squared states) if none of the group members have installed a view, and *available* (circular states) if at least one member has installed a view. Each state is identified by $X\sharp$ and a tuple $(x\mathbf{r}, y\mathbf{v})$, were x is the number of installed replicas ($\mathbf{r}$), and y is the number of members in the current view ($\mathbf{v}$) of the server group. In the diagram, we consider only events that may affect the availability of the service, such as *view changes*, *replica creations* as seen from the perspective of ARM, and *replica failures* as perceived by the corresponding MS nodes that fails. View changes, in particular, are denoted by *view-i*, where i is the cardinality of the view. In addition, *fault injection* events may occur in any of the states, however for readability they are not included in the figure.

As a sample failure-recovery behavior, consider the trajectory composed of the state transitions with dashed lines, starting and ending in $X0$. This is the most common trajectory. For simplicity the *view-i* events in the diagram reflect the series of views as seen by ARM, and do not consider the existence of concurrent views. So, after recovering from a failure (moving from state $X4$ to state $X3$), the newly created member will install a singleton view and thus be the leader of that view, sending a *view-1* event to ARM (from state $X3$ to state $X6$). Only after this installation (required by the view synchrony property) a *view-3* event will be delivered to ARM, causing a transition from state $X6$ to $X0$. The above simplification does not affect the availability of the service. It is assumed that client requests are only delayed during failure-recovery cycles as long as the service is in an operational state [16]. Such delays are not considered part of the availability measure as opposed to [12]. Further analysis of these client perceived delays is in preparation and will be included in a future paper.

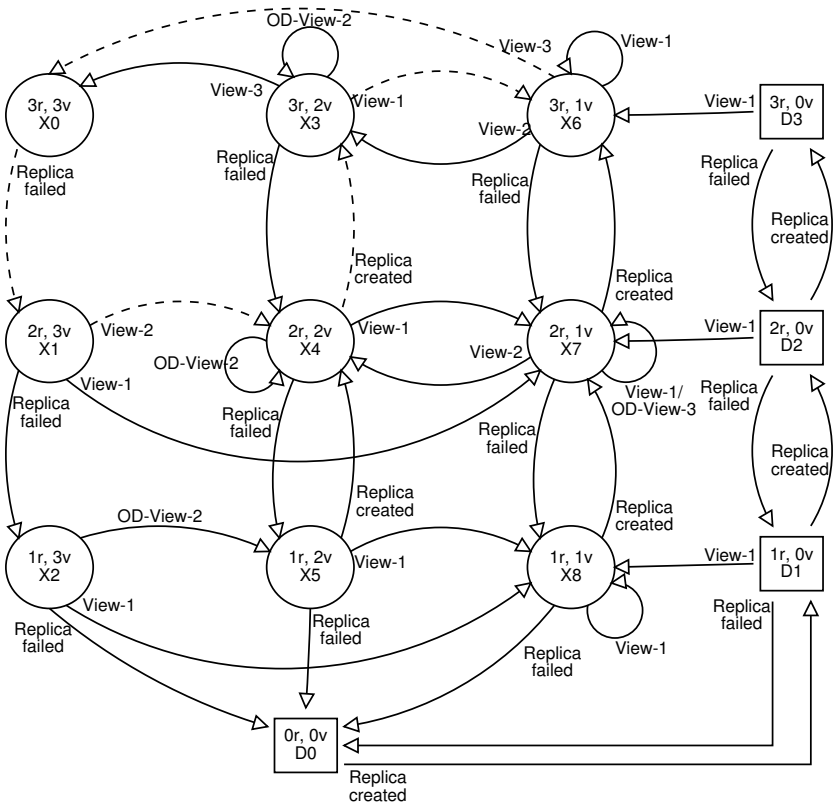


Fig. 3. State diagram illustrating a sample of the possible state changes of the MS service being measured.

Notice that some of the states have self-referring transitions on view change events. These are needed for several reasons, one being that the ARM framework may see view change notifications from several replicas, before they have formed a common view. In addition, ARM will on rare occasions receive what we call *outdated views*, that are due to minor inaccuracies in our measurements. For instance, a *view-3* event may occur while in state $X7$. This can occur if at some point we are in the $X6$ state, when a group member sends out a notification of a *view-3* event, and shortly after another member of that group fails and logs a $ReplicaFailed$ event. However, given that the *view-3* event is still in the "air", and has not yet been logged, the $ReplicaFailed$ event will appear to have occurred before the *view-3* event in the trace. To compensate for this behavior, we have inserted additional *view-i* transitions, prefixed by OD, in some of the states.

Notice also the *view-2* transition from $X2$ to $X5$. This is also due to an outdated view, and can occur if ARM triggers recovery on a *view-2* event before receiving a *view-1* event. Note that the state transitions in the diagram may not be complete as presented, however, no other transitions have been observed during our experiments. In the following, we will assume that the service has been initialized correctly into state $X0$, and thus we do not consider the initial transitions leading to this state.

4 Measurements

This section give motivation for our measurement approach. Furthermore, we discuss in detail the sampling scheme used to assess the fault handling capability of the Jgroup/ARM framework and to provide input to the prediction of dependability attributes.

4.1 Experiment Outline

In each experiment run, one or more faults are injected. The failure insertion pattern is as if it emerged from a Poisson process. There may be multiple near-coincident failures before the system stabilizes, i.e., a new failure may be inserted before the previous has been completely handled. This will "simulate" the rare occurrence of nearly coincident failures which may bring the service down. The Poissonian character of the inserted failures is achieved through generation of fault injection times and the selection of the set of processors in which to inject faults, according to a uniform distribution. See the Sampling Scheme in Section 4.3 on how this yields a Poisson fault process. Processors to crash are drawn from the entire target system. Hence, the injected faults may affect the ARM infrastructure itself, the monitored subsystem (MS) or the additional service (AS), all of which are being managed by the ARM framework. However, only state trajectories for the monitored subsystem are computed, and these are used for predicting various dependability attributes of MS. A beneficial "side-affect" of this sampling scheme is that it has shown to be very useful with respect to performing extensive testing of the fault handling capabilities of the Jgroup/ARM. During previous experiments several design and implementation faults have been revealed. In the experiments, we perform at most $k = 3$ fault injections during a run. Since initially all processors in the target system have allocated replicas, failures will cause ARM to reuse processors as shown in Fig. 2 where the replica of processor 7 is recreated at processor 3.

Time Constants Considered. Assuming services are deployed using the ARM framework, the crashed processors will have a *processor recovery time* (t_{PR}) which is much longer than the *service recovery time* (t_{SR}). Further, we assume that the processors will stay crashed for the remaining part of the experiment. In other words, a service replica will typically be restarted on a different processor as soon as ARM concludes that a processor crash has occurred. However, the time until the processors are recovered, is assumed to be negligible compared to the *time between failures* (t_{BF}) in a real system. Thus in the predictions it is assumed that the occurrence intensity of new trajectories (i.e. first failure in a fully recovered system) is $n\lambda$, neglecting the short interval with a reduced number of processors between t_{SR} and t_{PR}. Fig. 4 shows these relations, starting with the first failure event t_{i_1}. Furthermore, there will be no resource exhaustion, i.e., there are sufficient processors to execute all deployed services, including the ARM infrastructure.

Fig. 4. The relation between the service and processor recovery periods and the time between failures.

The Failure Trajectory. A failure trajectory is the series of events and states of the monitored subsystem following the first processor failure and until all the concurrent failure activities have concluded and all subsystems are recovered and fully replicated. The trajectory will always start and end in state $X0$ (see Fig. 3). *If* the first processor failure affects the monitored service, it causes it to leave its steady operational state $X0$ and *if* it is the last service to recover, we will see a return to the same state like in Fig. 5.

We denote the j^{th} event in the i^{th} trajectory by i_j, the time it takes place by t_{i_j} and the state after the event by X_{i_j} (corresponds to the states in Fig. 3). Note that all relevant events in the system are included, and a failure or another event does not necessarily cause a change of state in the monitored subsystem. For instance, the failure of a processor which supports only the ARM

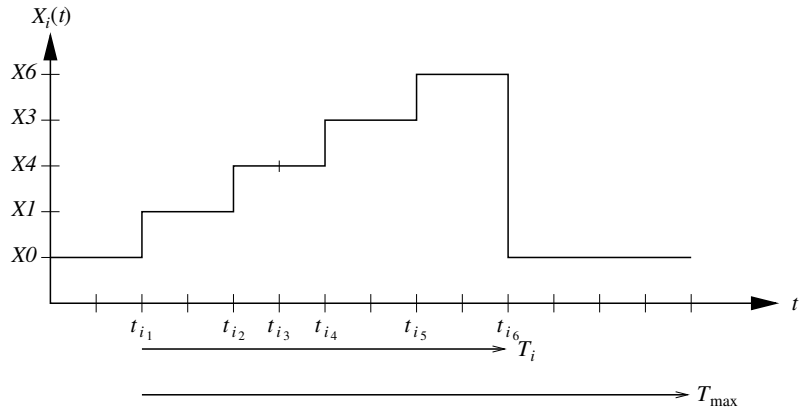


Fig. 5. Sample failure trajectory, where all but failure event t_{i_3} affects the subsystem of interest. This is the most common failure trajectory.

or AS subsystems, will not necessarily result in a change of state in the MS service, but it is likely that it will influence the handling of immediately preceding or succeeding failures affecting the service. Let $X_i(t)$ denote the state of the MS service at time t in the i^{th} failure trajectory,

$$X_i(t) = \begin{cases} X_{i_j} & t_{i_j} < t \leq t_{i_{j+1}},\ j = 1, \ldots, m_i \\ X0 & \text{Otherwise} \end{cases}$$

were m_i is the last event of the i^{th} trajectory before all concurrent failure activities have concluded, and all subsystems are fully replicated. During the measurements a trajectory sample is recorded as the list

$$\underline{X}_i = \left\{X0, t_{i_1}, X_{i_1}, t_{i_2}, X_{i_2}, t_{i_3}, \ldots, t_{i_{m_i}}, X0\right\}.$$

Note that we record also trajectories for which the MS service does not leave the $X0$ state.

Characteristics Obtained from a Failure Trajectory. The unknown probability of failure trajectory i is p_i. For brevity we denote the duration of trajectory i by $T_i = t_{i_{m_i}} - t_{i_1}$, and its expectation $\Theta = E(T) = \sum_{\forall i} p_i T_i$.

In the following, let Y_i denote a sample from the experiment. The sample may be obtained from the trajectory by some function g, i.e., $Y_i = g(\underline{X}_i)$. The duration of a trajectory presented above may serve as an example. Determining the dependability attributes of the system are other possible samples that can be extracted from the experiment data. To determine these, it is assumed that the failure rate in the $X0$ state is $n\lambda$, that the expected sojourn time in this state is much longer than the expected trajectory duration, and that a particular trajectory is independent of the previous trajectory.

Unavailability. The time spent in a down state during a trajectory is given by

$$Y_i^d = g(\underline{X}_i) = \sum_{j=1}^{m_i} I(X_{i_j} \in \mathfrak{F})(t_{i_{j+1}} - t_{i_j}), \tag{1}$$

where $I(\cdots)$ is the indicator function and $\mathfrak{F}$ is the set of down states (the squared states in Fig. 3). Given that the periods in state $X0$ (the OK-periods) alternate with the failure trajectories, and are independent and much longer than the failure trajectory periods, we can obtain a measure for the service unavailability

$$\hat{U} = \frac{E(Y^d)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^d) n\lambda.$$

Note that the collective failure intensity of all processors when there are no faults in the system, is only marginally different from the intensity of trajectories. The difference is due to the restoration of failed processors during a trajectory, and is negligible.

Probability of failure, reliability. In this case, let $Y_i^f = 1$ if trajectory i visits one or more down states, otherwise let $Y_i^f = 0$.

$$Y_i^f = g(\underline{X}_i) = I(\exists X_{i_j} \in \mathfrak{F})_{j=1,\ldots,m_i}. \tag{2}$$

Disregarding multiple down periods in the same trajectory and assuming that system failures are rare, it is found that the system failure intensity is approximately

$$\hat{\Lambda} = \frac{1}{\text{MTBF}} \approx \frac{E(Y^f)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^f)n\lambda.$$

In addition, the predicted reliability function $R(t) = \exp(-\hat{\Lambda}t)$ as well as the mean down time $\text{MDT} = \hat{U}/\hat{\Lambda}$ may be obtained. MDT and the down time distribution may of course also be measured directly from the trajectories visiting the set of down states.

The above examples are chosen for illustration and the assumptions made for simplicity. By introducing rewards associated with the states and transitions, we may obtain predictions of far more comprehensive performability measures of the system.

4.2 The Experiment Engine

The *experiment engine* is used to perform repeated experiment runs. At each run, numerous tasks are executed; (1) bootstrap the object factories onto the processors in the target system, (2) bootstrap the ARM infrastructure, (3) deploy the initial MS and AS replicas, (4) perform fault injections, according to the scheme described in Section 4.3, (5) shutdown the experiment run, and finally, (6) collect and remove log files from the target system.

To be able to compute a trajectory of states and the time spent in each state, Jgroup/ARM and the MS service has been instrumented with a simple event logging mechanism to be able to generate a local trace of events occurring at each of the processors in the target system. The events in a trace correspond to the events of the state diagram in Fig. 3. Each event trace contains the set of events and their occurrence time (in milliseconds), in addition to various details associated with the event. This level of accuracy is sufficient for our evaluation, as the time values considered (t_{SR}) are in the range 7-30 seconds. The occurrence time of an event correspond to the local clock of the processor at which the event occurred. The processor clocks in the target system are synchronized using NTP [18].

After each experiment, the log files generated are collected from the target system and stored at the experiment engine machine, for the offline analysis. In this analysis, the independent event traces collected from different machines are merged into a single global timeline of events, that correspond to an approximation of the actual state transitions of the whole system. Given this global event trace, we can compute the trajectory of visited states and the time spent in each of the states. These trajectories allow us to classify the experiments, and to predict a number of dependability attributes for the monitored service, as discussed previously.

4.3 Experimental Strategy

The experimental strategy is based on a *post stratified random sampling* approach. For an introduction to stratified sampling see for instance [14]. This section elaborates on how the experiments are classified in different strata, and how the sampling is performed.

Stratification. Only some of the events along a failure trajectory will actually be failure events. The first event of each trajectory will always be a failure, and in a typical operational environment usually the only one. However, in the experiments we consider also multiple near-coincident failures which may require concurrent failure handling. In considering such failure scenarios, our experimental strategy is based upon subdividing the trajectories into strata S_k based on the number of failure events k in each of the trajectories. Each of the strata are sampled separately, and the

number of samples in each stratum are random variables determined a posteriori. This is different from previous work [9] in which the number of samples in each stratum is fixed in advance.

An example failure trajectory reaching stratum S_3 drawn from the experiment data is shown in Fig. 6. Three near-coincident fault injections were performed in this particular experiment. The first and last failure affect the MS service, while the second affect the RM service. The RM failure and its related events, as indicated on the curve, do not cause state transitions in the state diagram (Fig. 3) of the MS service.

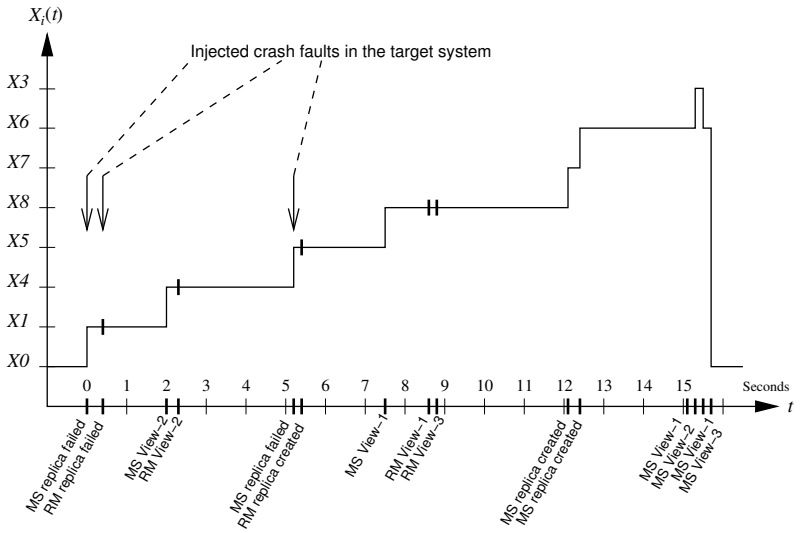


Fig. 6. Sample failure trajectory reaching stratum S_3 plotted on an approximate time scale. Only two of three injected faults affect the MS service. The second fault injection affect the RM service.

The collected samples for each stratum are used to obtain statistics for the system in that stratum, e.g., the expectation $E(Y|S_k)$. The expectation and the variance of the length of the trajectory within a stratum S_k are denoted $\Theta_k = E(T|S_k)$ and $\sigma_k = Var(T|S_k)$, respectively. Estimates may then be obtained by

$$E(Y) = \sum_{k=1}^{\infty} E(Y|S_k)\pi_k \approx \sum_{k=1}^{3} E(Y|S_k)\pi_k, \tag{3}$$

where $\pi_k = \sum_{\forall i \in S_k} p_i$ is the probability of a trajectory in stratum S_k. Recall that k represents the number of possible concurrent failure events, and in (3), we replace ∞ in the summation with 3, since we only consider up to 3 concurrent failure events. Expressions for π_k are derived in Section 4.4.

If upper and lower bounds for Y exist, and we are able to determine $\pi_k, k > 3$, we may also determine bounds for $E(Y)$ without sampling the higher-order strata, i.e.,

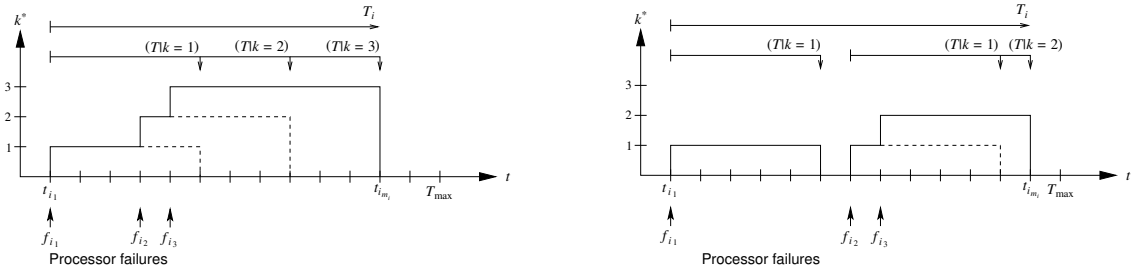
$$\sum_{k=1}^{3} E(Y|S_k)\pi_k + \inf(Y)\sum_{k>3}\pi_k \le E(Y) \le \sum_{k=1}^{3} E(Y|S_k)\pi_k + \sup(Y)\sum_{k>3}\pi_k.$$

Since the probability of k concurrent failures is much greater than $k+1$ failures, $\pi_k \gg \pi_{k+1}$, the bounds will be tight, and for the estimated quantities the effect of estimation errors are expected to be far larger than these bounds. The effect of estimation errors is discussed in Section 4.4.

Sampling Scheme. Under the assumption of a homogeneous Poisson fault process with intensity λ per processor, it is known that if we have $k-1$ faults (after the first failure starting a trajectory) of n processors during a fixed interval $[0, T_{\max}\rangle$, these will occur

– uniformly distributed over the set of processors, and
– each of the faults will occur uniformly over the interval $[0, T_{\max}\rangle$.

Note that, all injected faults *will* manifest itself as a failure, and thus the two terms are used interchangeably. In performing experiments, the value $T_{\max}$ is chosen to be longer than any foreseen trajectory of stratum S_k. However, it should not be chosen excessively long, since this may result in too rare observations of higher-order strata.



(a) Fault injections causing the failure trajectory into higher-order strata.

(b) Failure trajectory that completes before reaching higher-order strata.

Fig. 7. Sample failure trajectories with different fault injection times.

In the following, let $(T|k = l)$ denote the duration of a trajectory if it completes in stratum S_l, as illustrated in Fig. 7(a), and let f_{i_l} denote time of the l^{th} failure, relative to the first failure event in the i^{th} failure trajectory. That is, we assume the first failure occur at $f_{i_1} = 0$ and that $f_{i_l} > f_{i_1}, l > 1$. To obtain dependability characteristics for the system, we inject k failures over the interval $[0, T_{\max}\rangle$. This leads to the following failure injection scheme for trajectory i, which *may* reach stratum S_k. However, not all trajectories obtained for experiments with $k > 1$ failure injections will reach stratum S_k, since a trajectory may reach $(T|k = 1)$ before the second failure (f_{i_2}) is injected. That is, recovery from the first failure may complete before the second failure injection, as illustrated in Fig. 7(b). Such experiments contain multiple trajectories, however, only the first trajectory is considered in the analysis to avoid introducing a bias in the results.

1. The first failure, starting a failure trajectory i, is at $f_{i_1} = 0$. The following $k - 1$ failure instants are drawn uniformly distributed over the interval $[0, T_{\max}\rangle$ and sorted such that $f_{i_q} \leq f_{i_{q+1}}$ yielding the set $\{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\}$. Let k^* denote the reached stratum, and l is the index denoting the number of failures injected so far. Initially, set $k^* := 0$ and $l := 1$.
2. Fault $l \leq k$ is (tentatively) injected at f_{i_l} in processor $z_l \in [1, n]$ with probability $1/n$.
   (a) If trajectory i has not yet completed, i.e., $f_{i_l} < T_i$, then set $l := l + 1$ and
      i. If the selected processor has not already failed $z_l \notin \{z_w | w < l\}$: *Inject fault* at f_{i_l} and set $k^* := k^* + 1$
      ii. Prepare for next fault injection, i.e., goto 2.
   (b) Otherwise the experiment ended "prematurely".
3. Conclude and classify as a stratum S_{k^*} measurement.

The already failed processors are kept in the set to maintain the time and space uniformity corresponding to the constant rate Poisson process. Although k failures are not injected in a trajectory, the pattern of injected failures will be as if they came from a Poisson process with a specific number (k^*) of failures during $T_{\max}$. Hence, the failure injections will be representative for a trajectory lasting only a fraction of this time.

4.4 Estimators

Strata Probabilities. In a real system, the failure intensity λ will be very low, i.e., $\lambda^{-1} \gg T_{\max}$. Hence, we may assume the probability of a failure occurring while the system is on trajectory $i \in S_1$ is $T_i(n-1)\lambda$. Hence, the probability that a trajectory (sample) belonging to a stratum $S_k, k > 1$ occurs, given that a stratum S_1 cycle has started is

$$\frac{\sum_{\forall i \in S_1} p_i T_i (n-1)\lambda}{\sum_{\forall i \in S_1} p_i} = \frac{\sum_{k>1} \pi_k}{\pi_1}.$$

Due to the small failure intensity, we have that $\sum_{k>1} \pi_k \approx \pi_2$ and the unconditional probability of a sample in stratum S_2 is approximately

$$\pi_2 = (n-1)\lambda \Theta_1 \pi_1. \tag{4}$$

This line of argument also applies for the probability of trajectories in stratum S_3. However, in this case we must take into account the first failure occurrence. Let $i \in S_k \wedge X_i(t_x) \bowtie \mathfrak{f}$ denote a trajectory of stratum S_k, where a failure occurs at t_x. The probability that a trajectory belonging to stratum $S_k, k > 2$ occurs, given that a stratum S_2 cycle has started is, cf. Fig. 7(a):

$$\frac{\int \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i (T_i - t_x)(n-2)\lambda dt_x}{\sum_{\forall i \in S_2} p_i} = \frac{\sum_{k>2} \pi_k}{\pi_2}. \tag{5}$$

Ignoring the constant part of (5) for now; the first term on the left hand side of (5) is not depending on t_x and may be reduced as follows:

$$\frac{\int \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i T_i dt_x}{\sum_{\forall i \in S_2} p_i} = \frac{\sum_{\forall i \in S_2} p_i T_i}{\sum_{\forall i \in S_2} p_i} = \Theta_2.$$

For the second term we have, slightly rearranged:

$$\int t_x \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i dt_x.$$

The probability of having a stratum S_2 trajectory experiencing its third failure at t_x is the probability that the first (and second) failure has not been dealt with by t_x, i.e., the duration $T_j > t_x, j \in S_1$ and that a new failure occurs at t_x. These two events are independent. Up to the failure time t_x, the trajectories of strata S_1 and S_2 passing this point are identical. Hence, $\sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i = \Pr\{T_j > t_x\} \pi_1 (n-1)\lambda$ and by partial integration,

$$\int t_x \Pr\{T_j > t_x\} dt_x = \frac{1}{2} E(T_j^2 | j \in S_1) = \frac{1}{2}(\Theta_1^2 + \sigma_1).$$

Combining the above, inserting it into (5), using that $\sum_{\forall i \in S_2} p_i = \pi_2$ and that due to the small failure intensity $\sum_{k>2} \pi_k \approx \pi_3$, the unconditional probability of a trajectory in stratum S_3 approximately becomes:

$$\begin{aligned} \pi_3 &= (n-2)\lambda(\Theta_2 \pi_2 - \frac{1}{2}(\Theta_1^2 + \sigma_1)\pi_1 (n-1)\lambda) \\ &= (n-1)(n-2)\lambda^2(\Theta_2 \Theta_1 - \frac{1}{2}(\Theta_1^2 + \sigma_1))\pi_1. \end{aligned} \tag{6}$$

Since we have that $1 > \pi_1 > 1 - \pi_2 - \pi_3$ and as argued above, a sufficiently accurate estimate for π_1 may be obtained from the lower bound since $1 \approx \pi_1 \approx 1 - \pi_2 - \pi_3$, or slightly more accurately by solving π_i from (4), (6) and $1 = \pi_1 + \pi_2 + \pi_3$.

Estimation Errors. The estimation errors or the uncertainty in the obtained result is computed using the sectioning approach [14]. The experiments are subdivided into $N \sim 10$ independent runs of the same size. Let $\hat{E}_l(Y)$ be the estimate from the l^{th} of these; then:

$$\hat{E}(Y) = \frac{1}{N}\sum_{l=1}^{N}\hat{E}_l(Y), \quad \hat{\mathrm{Var}}(Y) = \frac{1}{(N-1)}\sum_{l=1}^{N}(\hat{E}_l(Y^2) - \hat{E}^2(Y)).$$

5 Experimental Results

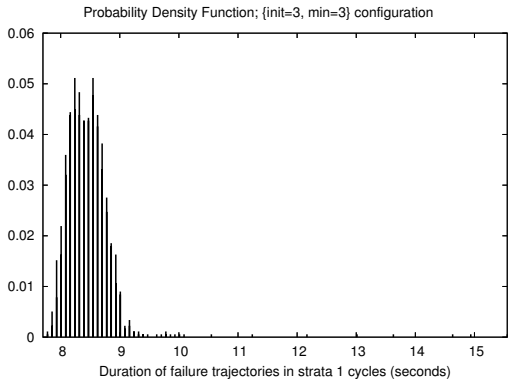
This section presents experimental results of fault injections on the target system. A total of 3000 experiment runs were performed, aiming at 1000 per stratum. Each experiment is classified as being of stratum S_k, if exactly k fault injections occur before the experiment completes (all services are fully recovered). The results of the experiments are presented in Table 1. Some runs "trying to achieve higher order strata" (S_3 and S_2) fall into lower order due to injections being far apart, cf. Fig. 7(b), or addressing the same processor.

Table 1. Results obtained from the experiments (in milliseconds).

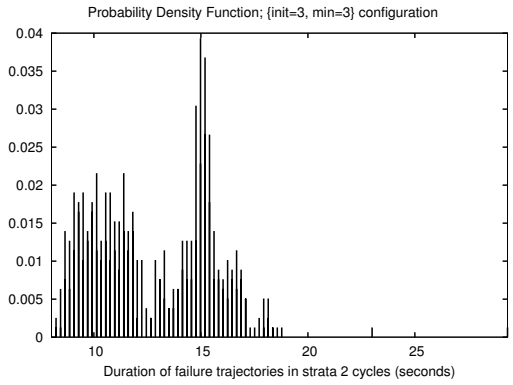
Classification	Count	$\Theta_k = E(T\|S_k)$	sd=$\sqrt{\sigma_k}$	Θ_k, 95% conf.int.
Stratum S_1	1781	8461.77	185.64	(8328.98, 8594.56)
Stratum S_2	793	12783.91	1002.22	(12067.01, 13500.80)
Stratum S_3	407	17396.55	924.90	(16734.96, 18058.13)

Of the 3000 runs performed, 19 (0.63%) were classified as inadequate. In these runs one or more of the services failed to recover (16 runs), or they behaved in an otherwise unintended manner. In the latter three runs, the services did actually recover successfully, but the runs were classified as inadequate, because an additional (not intended) failure occurred. The inadequate runs are dispersed with respect to experiments seeking to obtain the various strata as follows; two for S_1, 6 for S_2, and 11 for stratum S_3. One experiment resulted in a complete failure of the ARM infrastructure, caused by three fault injections occurring within 4.2 seconds leaving no time for ARM to perform self-recovery. Of the remaining, 13 were due to problems with synchronizing the states between the RM replicas, and 2 were due to problems with the Jgroup membership service. Even though none of the inadequate runs reached the down state, $D0$, for the MS service, it is likely that additional failures would have caused a transition to $D0$. To be conservative in the predictions below, all the inadequate runs are considered to have trajectories visiting down states, and causing a fixed down time of 5 minutes.

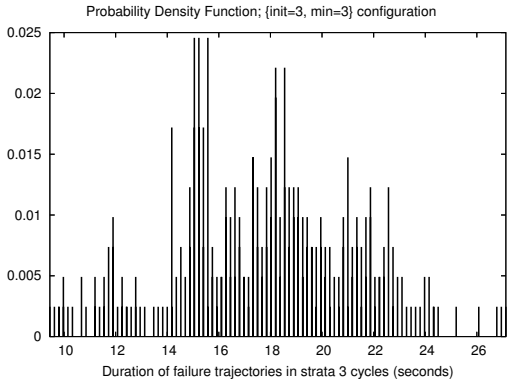
Fig. 8 shows the probability density function (pdf) of the recovery periods for each of the strata. The data for stratum S_1 cycles indicate that it has a small variance. However, 7 runs have a duration above 10 seconds. These durations are likely due to external influence (CPU/IO starvation) on the machines in the target system. This was confirmed by examining the cron job scheduling times, and the running time of those particular runs. Similar observations can be identified in stratum S_2 cycles, while it is difficult to identify such observations in S_3 cycles. The pdf for stratum S_2 in Fig. 8(b) is bimodal, with a top at approx. 10 and another at approx. 15. The density of the left-most part is due to runs with injections that are close, while the right-most part 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. Those injections that are close tend to be recovered almost simultaneously. The pdf for stratum S_3 has indications of being multimodal. However, the distinctions are not as clear in this case.



(a) Stratum S_1



(b) Stratum S_2



(c) Stratum S_3

Fig. 8. Probability density function of trajectory durations for the various strata.

Given the results of the experiments, we are able to compute the expected trajectory durations, Θ_1, Θ_2 and the variance σ_1 as shown in Table 1, and the unconditional probabilities π_2 and π_3 given in (4) and (6) for various processor mean time between failures (MTBF=λ^{-1}), as shown in Table 2. The low probabilities of a second and third near-coincident failure is due to the relatively short recovery time (trajectory durations) for strata S_1 and S_2. Table 2 compares these values with a typical processor recovery (reboot) time of 5 minutes and manual recovery time of 2 hours.

Table 2. Computed probabilities, unavailability metric and the system MTBF.

	Experiment Recovery Period		Processor Recovery (5 min.)		Manual Processor Recovery (2 hrs.)	
	Processor Mean Time Between Failure (MTBF=λ^{-1}) (in days)					
	100	200	100	200	100	200
π_1	0.99999314	0.99999657	0.99975688	0.99987845	0.99412200	0.99707216
π_2	$6.855602 \cdot 10^{-6}$	$3.427801 \cdot 10^{-6}$	$2.430555 \cdot 10^{-4}$	$1.215278 \cdot 10^{-4}$	$5.833333 \cdot 10^{-3}$	$2.916667 \cdot 10^{-3}$
π_3	$4.072921 \cdot 10^{-11}$	$1.018230 \cdot 10^{-11}$	$5.595341 \cdot 10^{-8}$	$1.398835 \cdot 10^{-8}$	$4.466146 \cdot 10^{-5}$	$1.116536 \cdot 10^{-5}$
$\hat{U}$	$4.671318 \cdot 10^{-7}$	$2.335617 \cdot 10^{-7}$	$2.777102 \cdot 10^{-4}$	$1.388720 \cdot 10^{-4}$	$6.627480 \cdot 10^{-3}$	$3.323574 \cdot 10^{-3}$
$\hat{\Lambda}^{-1}$	20.3367 yrs	40.6741 yrs	-	-	-	-

Of the 407 stratum S_3 runs, only 3 reached a down state. However, we include also the 19 inadequate runs as reaching a down state. Thus, Table 2 provides only indicative results of the unavailability ($\hat{U}$) and MTBF ($\hat{\Lambda}^{-1}$) of the MS service, and hence confidence intervals for these estimates are omitted. The results show as expected, that the two inadequate runs from stratum S_1 included with a service down time of 5 minutes, completely dominates the unavailability of the service. However, accounting for near-coincident failures may still prove important once the remaining deficiencies in the platform have be resolved. Although the results are indicative, it seems that very high availability and MTBF may be obtained for services deployed with Jgroup/ARM.

6 Conclusions

This paper has presented an approach for the estimation of dependability attributes based on the combined use of fault injection and a novel post stratified sampling scheme. The approach has been used to assess and evaluate a service deployed with the Jgroup/ARM framework. The results of the experimental evaluation indicate that services deployed with Jgroup/ARM can obtain very high availability and MTBF.

Thus far, our automated fault injection tool has proved exceptionally useful in uncovering at least a dozen subtle bugs, allowing systematic stress and regression testing. In future work, we intend to improve the Jgroup/ARM framework further to reduce the number of service failures due to platform deficiencies. The approach may also be extended to provide unbiased estimators, allowing us to determine confidence intervals also for dependability attributes given enough samples visiting the down states.

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. J. Arlat, M. Aguera, , L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb. 1990.

3. J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell. Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System. *IEEE Transactions on Reliability*, 39(4):455–467, Oct. 1990.
4. D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, Sept. 1996.
5. Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, Apr. 2001.
6. B. Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University, July 1998.
7. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Commun. ACM*, 36(12):36–53, Dec. 1993.
8. 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 Transactions on Parallel and Distributed Systems*, 15(7):593–605, July 2004.
9. M. Cukier, D. Powell, and J. Arlat. Coverage Estimation Methods for Stratified Fault-Injection. *IEEE Transactions on Computers*, 48(7):707–723, July 1999.
10. P. Felber. *The CORBA Object Group Service: a Service Approach to Object Groups in CORBA*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Jan. 1998.
11. U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. of the 19th Int. Symp. on Fault-Tolerant Computing*, pages 340–347, Chicago, IL, USA, June 1989.
12. K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental Evaluation of the Unavailability Induced by a Group Membership Protocol. In *Proc. of the 4th European Dependable Computing Conference*, pages 140–158, Toulouse, France, Oct. 2002.
13. C. Karamanolis and J. Magee. Client-Access Protocols for Replicated Services. *IEEE Transactions on Software Engineering*, 25(1), Jan. 1999.
14. 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.
15. H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.
16. 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 Communications Conf.*, Phoenix, Arizona, Apr. 2004.
17. H. Meling, A. Montresor, Ö. Babaoğlu, and B. E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS-2002-12, Dept. of Computer Science, University of Bologna, Oct. 2002.
18. D. L. Mills. Network Time Protocol (Version 3); Specification, Implementation and Analysis, Mar. 1992. RFC 1305.
19. A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.
20. P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, Dec. 1999.
21. D. Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, Feb. 1994.
22. Y. Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
23. 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 Transactions on Computers*, 52(1):31–50, Jan. 2003.