A Framework for Experimental Validation and Performance Evaluation in Fault Tolerant Distributed System

Hein Meling

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

Abstract

Performing experimental evaluation of fault tolerant distributed systems is a complex and tedious task, and automating as much as possible of the execution and evaluation of experiments is often necessary to test a broad spectrum of possible executions of the system to obtain good coverage. The confidence of the results obtained from an experimental evaluation depends on the degree of control over the environment in which experiments are being executed. Typically, an uncontrolled environment is exposed to numerous sources of external influence that can affect the obtained results. Automated and repeated executions can be used to reduce the impact of such influences.

In this paper, a framework for experimental validation and performance evaluation of fault management in a fault tolerant distributed system is presented. The framework provides a facility to execute experiments in a configured target system. It is based on injecting faults or other events needed to test the fault handling capability of the system. Relevant events are logged and collected for post-processing and analysis, e.g. to construct a single global timeline of events occurring at different nodes in the target system. This timeline of events can then be used to validate the behavior a system, and to evaluate its performance.

1. Introduction

Testing the validity of fault tolerant distributed systems and measuring the performance impact of faults is a challenging task. A common technique is to apply *fault injection* (see for instance [1, 2, 10]) as a means to accelerate the occurrences of faults in the system. The main purpose of fault injection is to evaluate and debug the error detection and recovery mechanisms of the distributed systems.

This paper presents a framework for experimental validation and performance evaluation of fault management in a fault tolerant distributed system. The framework enables us to execute experiments in a configured target system, where faults may be injected for the purpose of testing the fault handling capabilities of the system undergoing testing. Relevant fault and system events are logged and collected for post-processing and analysis, e.g. to construct a single global timeline of events occurring at different nodes in the target system. This timeline can then be used to validate the behavior a system, and to evaluate its performance.

Numerous systems [1, 4, 7, 20] have been developed to provide generic fault injection tools aimed at testing the fault handling capability of systems. The most relevant ones are discussed briefly below. Loki [4] is a global state-driven fault injector for distributed systems. Faults are injected based on a partial view of the global state of a system, i.e. faults injected on one node of the system can depend on the state of other nodes. Loki has been used to inject correlated network partitions to evaluate the robustness of the Coda distributed filesystem [12]. Orchestra [7] is a script-driven probing and fault injection tool designed to test distributed protocols. It is based on inserting a fault injection protocol layer below the target protocol that will filter and manipulate messages exchanged between protocol participants. Since a separate layer is used, the source code of the tested application does not need to be modified. NFTAPE [20] is a software infrastructure for assessing the dependability attributes of networked configurations. The main feature of NFTAPE is extensibility, and is so in two ways: (i) a suite of tools to support specifying injection scenarios, and (ii) a library of injection strategies and a lightweight API to customize injection strategies or develop new ones. Each machine in the target system is associated with a process manager which communicates with a centralized

1-4244-0910-1/07/$20.00 ©2007 IEEE.

controller. The centralized controller injects faults according to a specified fault scenario by sending commands to the process managers. Neko [21] is a framework for designing, tuning, and analyzing the performance of distributed algorithms. Algorithms can be either simulated or executed on a real network.

The experiment framework presented in this paper distinguishes itself from these other frameworks in its support for testing fault treatment mechanisms of systems. It offers a facility for execution of experiments in a configured target system, and supports the injection faults to emulate realistic failure scenarios. Both crash and network failures are supported; fault injections are generally based on random selection, rather than being triggered by the global state of the system as in Loki [4]. A particular focus has been to test system behavior when exposed to a series of near-coincident failures [11, 16]. The framework is intentionally simple, modular and lightweight to avoid any overhead. The framework has been used for testing Jgroup/ARM (see Section 2). A significant portion of the Jgroup/ARM API is reused by the framework to ensure consistency between the various tunable parameters of the target system.

Terminology (partially borrowed from [4]): To experimentally evaluate a system, one or more *studies* may be defined, e.g. a crash failure study or a network instability tolerance study. For each study, one or more *configurations* are defined; a configuration typically specify the target system and deployment parameters such as the number of replicas for each service. However, in the following only a single configuration per study is considered. To obtain statistically significant measures, several runs of each study are performed. Each of these runs is called an *experiment*. Each study (and configuration) is evaluated separately. After each experiment, the system is reset to its original configuration before beginning the next experiment.

During an experiment, events are logged. The set of events to be logged are defined *a priori*, and the code is instrumented with logging code. After the completion of an experiment the log files are collected for analysis. Experiment analysis is specific to each study and typically involves the construction of a single global timeline of events occurring at the different nodes in the target system. This global timeline of events can then be used to validate the behavior of the system, and to evaluate its performance. For instance, a predefined state machine for the system behavior may be used to validate the behavior of the system by projecting the event trace onto the state machine.

Paper organization: Section 2 gives an overview of the Jgroup/ARM middleware platform for which the experiment framework is designed. Section 3 gives an architectural overview of the experiment framework, while Section 4 describes the organization of experiment scripts. Section 5 discuss two fault injectors used in previous experiments. Section 6 presents the organization of the analysis modules, and finally in Section 7 the impact on the tested system and the accuracy of the instrumentation is discussed.

2. Jgroup/ARM Overview

Jgroup/ARM is a novel middleware platform based on object groups for developing, deploying and operating distributed applications with strict dependability requirements. *Jgroup* [18] is a group communication service that integrates the Java RMI distributed object model with object groups. Apart from "standard" group communication facilities, Jgroup includes several features that make it suitable for developing modern networked applications. Firstly, Jgroup supports *partition-awareness*: replicas in disjoint network partitions are informed about the current state of the system, and may take appropriate actions to ensure the availability of the provided service in spite of the partitioning. A network partition occur when failures render communication between subsets of nodes impossible. By supporting partitioned operation, Jgroup trades consistency for availability, whereas other systems takes a *primary partition* approach [5], ensuring consistency by allowing only a single partition to make progress. A *state merging service* is provided to simplify the re-establishment of a consistent global state when partitions merge. Jgroup is unique in providing a uniform object-oriented programming interface (based on RMI) to govern *all* object interactions both within an object group as well as interactions with clients.

The *Autonomous Replication Management* (ARM) framework [14, 16, 13] extends Jgroup with automated mechanisms for performing management activities such as distributing replicas among nodes and recovering from replica failures, thus reducing the need for human interactions. These mechanisms are essential to operate a system with strict dependability requirements, and are largely missing from existing group communication systems [8, 3]. The ARM framework supports seamless deployment and operation of dependable services. The set of nodes that may host applications and ARM-specific services is called the *target environment*; within it, issues related to service deployment, replica distribution and recovery from failures are autonomically managed by ARM, following the rules of user-specified distribution and replication policies. Maintaining a fixed redundancy level is a typical requirement specified in the replication policy. Failure scenarios are discovered and handled through recovery actions with the objective to minimize the period of reduced failure resilience, and objects may be relocated/removed to adapt to uncontrolled changes such as failure/merge scenarios, or controlled changes such as scheduled maintenance (e.g. OS upgrades), as well as software upgrade management [19]. These features enable self-healing and self-configuring services.

3. Architectural Overview

The experiment framework is designed to perform repeated experiments of a study to obtain statistically significant measures. Fig. 1 shows the components of the framework, where the *experiment executor* is the main component. Its purpose is to execute scripts defining a study. In each experiment numerous tasks are executed, e.g.:

1. Reset and initialize the nodes in the target system
2. Bootstrap the factories onto nodes in the target system
3. Bootstrap the ARM infrastructure
4. Deploy the replicas
5. Inject faults
6. Shutdown the experiment
7. Collect log files from the target system nodes

Each component in the architecture is defined by a set of tasks that it performs. Tasks are building blocks for constructing study scripts, and each script is comprised of a set of common/specialized tasks. For example, the specialized fault injector and analysis tasks used for the two experiments reported in [11] and [16] are completely different. The experiment executor interacts with the other components to activate tasks according to the study script.

Nodes in the target system must host a fault injector through which faults can be injected. Depending on the type of faults being injected the fault injector code may have to be instrumented into the system code on the target node.

The events of interest must be logged for use in the analysis phase, and typically requires additions to the code. The log files are collected from each node in the target system and stored in a repository for post-processing.

During a study, the CPU/IO activity of the nodes in the target system is checked before and after each experiment. Experiments whose load exceeds some configurable threshold may then be marked for further analysis. This is particularly useful to detect artifacts due to external influence when the study is performed in an uncontrolled environment.

The analysis component is organized in two separate modules; one module to process each experiment individually and another module to process all experiments in the study collectively and produce statistics. Typically, the latter module will use the former to obtain measurements from each individual experiment. Note that each experiment may be analyzed after its completion, and the results of the analysis can be used by the experiment executor to make decisions; hence the dashed arrow between the experiment executor and the analysis component in Fig. 1. The analysis component is discussed further in Section 6.

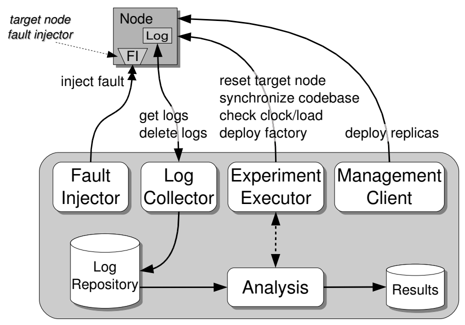

Figure 1. Experiment framework architecture.

4. Experiment Scripting

The initial version of the experiment framework used XML based scripts to specify the experiment tasks to be executed; the experiment tasks themselves were implemented in Java. The crash failure study presented in [11] was performed using the XML based framework. However, lack of support for control flow mechanisms in XML made it difficult and unnatural to write advanced study scripts. Therefore the experiment framework was ported [22] to the Groovy [9] scripting language, making it much easier to prepare complex study scripts. Groovy allows close integration with the Java language, thus enabling reuse of several Jgroup/ARM APIs. Study scripts are organized in four phases, each executing various tasks:

1. **Initialization:** The static configuration of the study is initialized. Dynamically adjustable parameters of the study are embedded within the experiment tasks below.
2. **Pre-study tasks:** Tasks performed only one time before the actual study begins. This typically involves the creation of a log repository on the experiment machine and synchronizing the codebase of all the nodes in the target system.
3. **Experiment tasks:** The main tasks needed to perform the study; these are repeated for each experiment. These tasks typically include: deploying the factories and replicas on the target system nodes, and injecting faults into the nodes in the target system. After the execution of an experiment, the logs are collected from the nodes in the target system and the nodes are reset, e.g. by killing any remaining experiment processes and deleting log files.
4. **Post-study tasks:** Tasks performed only one time after the completion of the study. For example, to remove log files from the target system nodes.

5. Code Instrumentation

Instrumenting code for our experiments is done by inserting logging statements and other code directly inside the actual source code of the system under study.

5.1. The Logging Facility

To simplify the logging of various system and failure events a logging facility is provided. A particular *event* is recorded by logging calls inserted at appropriate locations in the source code. Each recorded event includes:

- Time of event; obtained from the local processor clock.
- Machine name on which the event was recorded.
- Event type and a brief description.

The recorded events are Java objects and support is provided for ordering the events into a single global timeline independent of the node on which the events occurred. Such ordering requires that the processor clocks of all the nodes in the target system are synchronized using NTP [17]. The granularity of the clock is one millisecond. Nanosecond granularity is also possible for relative time between events occurring on the same node. The precision obtained using NTP is in the range 1-5 ms, according to the offset values obtained from the `ntpdate` command. This level of accuracy makes it very unlikely that events recorded on different nodes are ordered incorrectly in the global trace. Note that the clock offset values of each node are checked before and after each experiment to detect deviations above some threshold. Experiments with too large a clock deviation may be marked and excluded from further consideration.

The event class used by the logging facility may be subclassed to include event-specific details. For instance the *view event* subclass includes the view object generated by the Jgroup membership service [18]. Event classes may also provide methods that can be used in the analysis phase to extract various properties from the event, for instance to check if a view event represents a fully replicated view.

To reduce the processing overhead of event logging, events are first stored in memory and periodically flushed to disk. However, to avoid loss of events in response to fault injections, the flush mechanism can also be triggered immediately before a fault injection.

5.2. Fault Injectors

The experiment framework currently supports two distinct randomized fault injectors; both implemented by means of code instrumentation:

- Crash failure injection
- Reachability change injections

The former kind of fault injector was used to perform an evaluation of dependability attributes by prediction through a *stratified sampling* approach as reported in [11, 13]. A series of experiments were performed; in each experiment, one or more faults were 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 completely handled. Three strata were considered, i.e. single failures, and double and triple near-coincident failures. The nodes of the system under study is assumed to follow the crash failure semantics. For the duration of an experiment, the events of interest are monitored, and post-experiment analysis constructs a single global timeline of fault injections and other relevant events. The timeline is used to compute trajectories on a predefined state machine. Given data from the experiments, we were able to predict several dependability attributes of the system under study, e.g. its availability.

The second fault injector was used to test the ability of Jgroup/ARM to tolerate network instability and partitioning due to network failures [16, 13]. Network instability and partition failures may arise for a number of reasons, e.g. router crashes or power outages, physical link damage, buffer overflows in routers, router configuration errors and so on. In the experiments, four reachability patterns were injected at random times, the last one returning to the fully connected reachability pattern. Multiple near-coincident reachability changes may occur before the system stabilizes, i.e. a new reachability pattern may be injected before the previous has been completely handled. In the study analysis density estimates for the various delays involved in fault detection and recovery were computed.

The Crash Failure Emulator. A crash failure occurs when a unit (e.g. object) halts, losing its volatile data. To support crash failure emulation, the factory has been instrumented with a `shutdown()` method. The `shutdown()` method simply sends a terminate signal to the replicas associated with the factory, forcing each replica to halt its execution. Fig. 2 illustrates the crash failure injector. When injecting multiple crash failures in a single experiment, all injections are sent to their respective nodes at the start of the experiment. A timer is then used to trigger the injections at the specified activation times. This way the communication step has a very low impact on the injection time accuracy.

Inaccuracy discussion: Injections are performed using the fastest possible way to stop a Java virtual machine from within itself, namely using the `Runtime.halt()` method. This means that no shutdown hooks or finalizers are executed during the shutdown sequence, as would be the case if we used the `System.exit()` method. However, measuring the accuracy of crash injections is difficult since it is not easy to

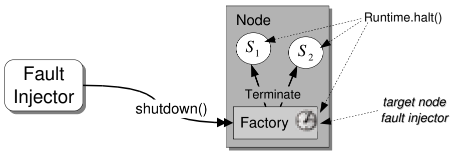

Figure 2. The crash failure injector.

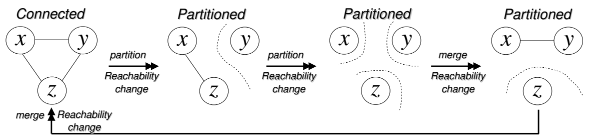

Figure 3. A sequence of reachability patterns.

Table 1. Statistics for crash injection (ms)

Mean	StdDev	Max	Min
13.833	4.772	32	8

accurately detect the time when the process ceases to exist. Hence the statistics on the injection time presented in Table 1 only accounts for the time taken from activating a crash failure at the specified time in the factory until immediately before the halt method is invoked in the replica JVMs. The results were obtained from 100 crash failure experiments. The results indicate that crash fault injections are quite fast, and hence do not contribute to any significant inaccuracy in the measurements reported in [11].

The Network Partition Emulator. At any given time, the connectivity state of the target environment is called the current *reachability pattern*. The reachability pattern may be *connected* (all nodes are in the same partition), or *partitioned* (failures render communication between subsets of nodes impossible). The reachability pattern may change over time, with partitions forming and merging as illustrated in Fig. 3. The letters x, y and z each denote a different site in the target environment. Injections causing a transition from one reachability pattern to another is called a *reachability change*. A reachability change is due to either a *partition* or a *merge* event. In the network instability tolerance study reported in [16, 13], the injected reachability patterns are *symmetric*. However, *asymmetric* reachability patterns are easily supported by the partition emulator.

Injecting and measuring real network partitions in a wide area network is difficult for a number of reasons: (i) lack of physical access and permissions to disconnect cables from switches/routers, (ii) it is difficult to measure the exact time of disconnection, and (iii) performing a large number of disconnections would be very time consuming. For these practical reasons, network partition scenarios are instead emulated. To accomplish this, Jgroup/ARM has been instrumented with code to emulate network partition scenarios.

The partition emulator allows us to remotely configure and inject the reachability changes to be seen by the various nodes in the target system. Each node in the target system has a local partition emulator module through which injections are managed by the experiment executor (see Fig. 1). The node local partition emulator is implemented by intercepting and discarding packets according to the configured reachability pattern. However, to avoid complicated changes to Jgroup/ARM, packet discarding must be done at the receiver, rather than the sender side. Hence, packets from "disconnected" nodes are also received and do require some minor processing. However, in our experiments on network instability tolerance [16] this processing overhead is negligible, since there are no clients generating traffic.

The injection of a new reachability pattern is organized in a *setup phase* and a *commit phase*. The former configures the reachability change to be injected, while the latter activates it. The setup phase also serves to establish TCP connections to be reused in the commit phase. The setup phase must be performed before the injection time. Fig. 4 shows the interactions needed to inject a new reachability pattern. The inaccuracy of the measured injection time is very small (65 ms on average) and does not contribute to any detectable effects in our measurements. Details of the inaccuracy and other limitations of our emulated reachability patterns are discussed below.

Inaccuracy discussion: Let I_i denote the injection time of the i^{th} injection event. Let δ_s denote the setup latency, which is the time from beginning a setup phase and until all nodes have been configured. Let δ_c be the commit latency, which is the time from the injection time I_i until all nodes have activated the new reachability pattern. These latencies limits the accuracy that can be obtained, as illustrated in Fig. 5. Two consecutive injections are shown, I_1 followed by I_2, which serves to illustrates the smallest possible delay between a pair of injections. That is, $\delta_s + \delta_c$ is the minimum time between two consecutive injections. Furthermore, δ_c limits the accuracy in detection of a newly injected reachability pattern. This is since each node may perceive the new reachability at different times, at most separated by δ_c.

Table 2 provides statistics for these two limiting factors. Note that these statistics do not show the complete picture, since there is an apparent bimodality in the setup latency, as illustrated in Fig. 6. The peak around 900 ms stems from the first setup phase shown before I_1 in Fig. 5 and is due to connection establishment between the experiment executor

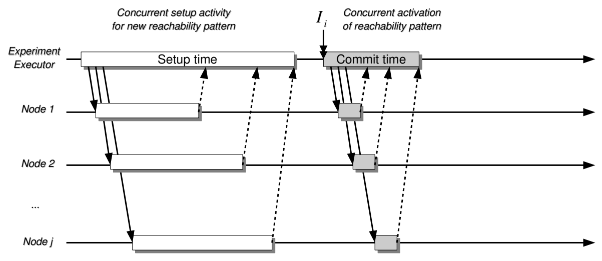
Concurrent setup activity for new reachability pattern
I_i
Concurrent activation of reachability pattern
Experiment Executor
Setup time
Commit time
Node 1
Node 2
...
Node j

Figure 4. The setup and commit phases.

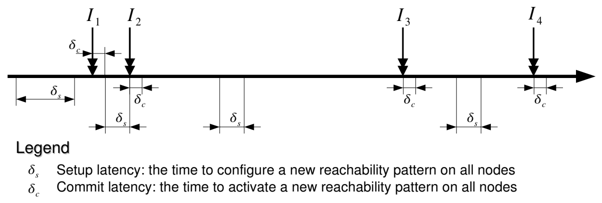
I_1
I_2
I_3
I_4
δ_c
δ_s
Legend
δ_s Setup latency: the time to configure a new reachability pattern on all nodes
δ_c Commit latency: the time to activate a new reachability pattern on all nodes

Figure 5. Example injection timeline

and the node-local fault injector modules. The peak around 450 ms is the latency typically seen between injections I_2, I_3 and I_4. Thus, taking also the commit latency (65 ms on average) into account, these observations seem to indicate that a pair of consecutive injections that arrive within an interval shorter than 500-600 ms cannot be reliably tested using our fault injector. However, such near injections are very rare, and in most cases would not be detected by the failure detector as a network partition in the first place. The density of the commit latency (δ_c) is shown in Fig. 7. The variations in the commit latency are rather small, and are most likely due to correlation between the commit invocations and the garbage collection mechanism of the various JVMs in the target system. It is the commit latency that limits the accuracy of partition detection. Hence, the results of our network instability tolerance study presented in [16] may have an inaccuracy of approximately 65 ms on average.

5.3. Avoiding Code Modification

Modifying the original source code to insert instrumentation logic is a common approach to evaluate systems, and is also used in the framework presented herein. The drawback is that it makes the code harder to understand – it is difficult to determine what is evaluation logic (fault injector) and what is actual system algorithms. Moreover, the evaluation logic must be removed or disabled when a real system is deployed.

In recent years a programming technique called aspect-oriented programming (AOP) [6] has become popular. AOP

Table 2. Statistics for setup and commit latencies for injections (ms).

	Mean	StdDev	Max	Min
δ_s	661.45	217.98	1040	425
δ_c	64.88	27.92	144	35

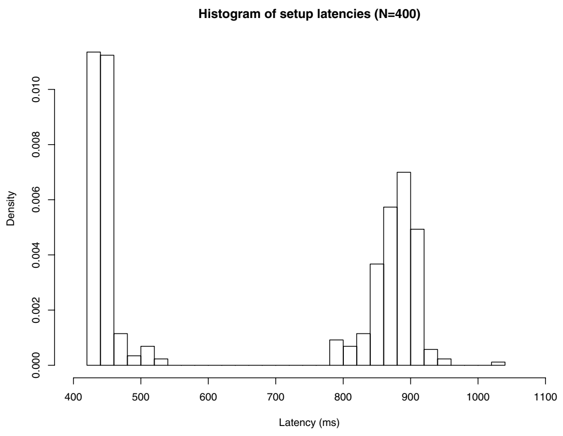

Figure 6. Histogram of setup latencies

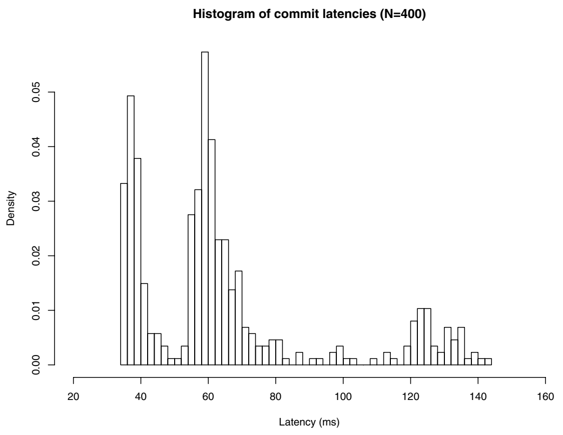

Figure 7. Histogram of commit latencies

extends object-oriented programming by introducing a new unit of modularity, called an *aspect*. Aspects are special modules that focus on *crosscutting concerns* in a system that are difficult to address in traditional object-oriented languages. In the future, code instrumentation could instead be handled using AOP techniques. That is, it is possible to define aspects that can ”insert“ logging statements or other interception logic at certain points in the code without having to modify the actual source code of the system. Such aspects are typically used only during testing, and can be easily be removed when deploying the system since they are provided by separate modules. An example could be an aspect to emulate network partition scenarios implemented through packet discarding. Such code is hardly useful in a deployed system, and if implemented as an aspect (a separate module), it is much easier to remove from the system than modifying the original source code. Note that using AOP techniques for inserting code and intercepting method calls may result in a slightly higher overhead compared to inline code instrumention as currently used.

6. Experiment Analysis

Experiment analysis is specific to the kind of study being performed, and is organized in two separate modules:

- Experiment analysis module (EAM)
- Study analysis module (SAM)

With EAM each experiment is processed individually; it can be used to extract measurement information from the log files (event traces) obtained after the completion of the experiment. Event traces from all nodes in the target system are collected and may be used to construct a single global event trace (timeline). This is done for the studies reported in [11] and [16].

The purpose of SAM is to aggregate the results obtained from the individual experiments to compute various statistical properties for the evaluation. Hence, SAM is used after the completion of all the experiments in the study. Typically, EAM is reused for data collection, by extracting relevant measures for which statistics should be computed. For example to extract detection and recovery delays, or to determine the system down time. SAM may also be used to construct a partial state machine given a collection of experiment traces. Given a sufficient number of traces, the constructed state machine may then be used to verify the correctness of future experiments.

In general, two kinds of studies are considered:

- Studies for system validation and error correction.
- Studies for performance and dependability evaluation.

The former aims to test the system functionality and allow the developer to obtain debug logs that can be used for debugging and error correction. In this case, EAM can be used online during a study execution to analyze each experiment individually after their completion. The results of the analysis can be used by the experiment executor to make decisions about the continued execution of the study. This is useful to determine whether a particular experiment should be repeated or if the study should be terminated. An experiment may be repeated by using the same fault injection data as in the original execution of the experiment; recall that fault injections may occur at randomized times. Repeating an experiment in this manner is useful to obtain additional debug logs from similar experiments, to better understand the incorrect system behavior and to be able to debug the problem. Furthermore, after a fix has been applied, the same fault injection scenario can be repeated to determine if the problem has been solved.

Note that repeating the same fault injection scenario does not offer any guarantee that the same bug is revealed again. However, if a particular bug is revealed in repeated experiments prior to applying a fix, and not after the fix, increased confidence is gained that the bug has in fact been fixed. After fixing the bug, a full randomized fault injection test should be performed again to determine if the fix has introduced new bugs. The EAM may also be use when debugging a particular experiment, e.g. by printing the global event trace for visual inspection.

In the past we have performed three studies focusing on performance and dependability evaluation of Jgroup/ARM as reported in [11, 16, 15].

7. Summary and Conclusions

The experiment framework presented in this paper has proved exceptionally useful in uncovering at least a dozen subtle bugs in the Jgroup/ARM platform, allowing systematic stress and regression testing. Below the impact of instrumentation and injection accuracy is discussed.

Impact of the instrumentation code. The experiment framework relies on logging system events to memory during experiment execution. Generally, these events are infrequent and thus will not influence the overall system significantly. Periodically, events are flushed to disk and this may result in minor disturbances if disk access is congested. Crash failure injections are passive in that they are only activated at the injection time, thus there is no other impact on the system during an experiment. On the other hand, partition failure injections are implemented by discarding packets according to the configured reachability pattern. With this approach some minor processing at each node is required, even for packets from ”disconnected“ nodes. This processing is done for all packets, independent of the reachability pattern. The processing overhead for each packet is very low. However, given a high system load, this packet processing overhead may have an impact on system performance. The Loki fault injector used to evaluate correlated network partitions in the Coda filesystem [12] is different from our approach. Instead of inline packet discarding, a firewall mechanism was used to configure blocking on specific ports. This approach is likely to give slightly less overhead as packets are discarded by the operating system. However, the drawback with this approach is that configuring the firewall often requires administrator (root) access. Unexpected system behaviors due to the instrumentation code have not been observed in our measurements reported in [13, 11, 16, 15].

Injection accuracy. The accuracy obtained from partition failure injections is very good. In the worst case a delay of 144 ms (the max commit latency) may separate the activation of a particular reachability pattern at two nodes. Hence, nodes may perceive a different reachability pattern at the same time instance. The impact of such a small delay is insignificant, since the view agreement protocol takes much longer to complete in most cases. In the worst case, it could cause additional protocol runs. The fact that different nodes perceive a different reachability pattern at the same time instance may also occur in real disconnection scenarios, e.g. if routing tables have been incorrectly altered. Such errors should be tolerated by the middleware.

Future work. The goal for future work is to build a generic scripting based evaluation framework for distributed Java applications, and prepare evaluation scripts to test it on at least three different systems. This generic framework will take advantage of code instrumentation by means of aspect-oriented programming, but will also reuse large portions of the current codebase.

References

[1] 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 Trans. Software Eng.*, 16(2):166–182, Feb. 1990.

[2] 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 Trans. Rel.*, 39(4):455–467, Oct. 1990.

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

[4] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7):593–605, July 2004.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, Dec. 2001.

[6] A. Colyer, A. Clement, G. Harley, and M. Webster. *eclipse AspectJ*. Addison-Wesley, 2004.

[7] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Technical Report CSE-TR-318-96, University of Michigan, EECS Department, 1996.

[8] 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.

[9] Groovy. http://groovy.codehaus.org/. Last visited May 2006.

[10] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. 19th Int. Symp. on Fault-Tolerant Computing*, pages 340–347, Chicago, IL, USA, June 1989.

[11] B. E. Helvik, H. Meling, and A. Montresor. An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In *Proc. Fifth European Dependable Computing Conference*, LNCS, pages 179–198. Springer-Verlag, Apr. 2005.

[12] R. M. Lefever, M. Cukier, and W. H. Sanders. An Experimental Evaluation of Correlated Network Partitions in the Coda Distributed File System. In *Proc. 22nd Symp. on Reliable Distributed Systems*, pages 273–282, Florence, Italy, Oct. 2003. IEEE Computer Society.

[13] 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.

[14] H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.

[15] H. Meling and B. E. Helvik. Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In *Proc. 23rd Int. Performance, Computing, and Comm. Conf.*, Phoenix, Arizona, Apr. 2004.

[16] H. Meling, A. Montresor, B. E. Helvik, and Ö. Babaoğlu. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management. Technical Report No. 11, University of Stavanger, Jan. 2006.

[17] D. L. Mills. Network Time Protocol (Version 3); Specification, Implementation and Analysis, Mar. 1992. RFC 1305.

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

[19] M. Solarski and H. Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *Proc. Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC*, Oxford, England, Aug. 2002.

[20] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proc. 4th Int. Computer Performance and Dependability Symp.*, 2000.

[21] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. 15th Int'l Conf. on Information Networking (ICOIN)*, pages 503–511, Beppu City, Japan, Feb. 2001.

[22] R. Vestvik. Paalitelighetsvurdering og integrasjonstesting av distribuerte applikasjoner. Master's thesis, Dept. of Electrical and Computer Engineering, University of Stavanger, June 2005. In Norwegian.