

Group Communication Protocols under Errors

Claudio Basile, Long Wang, Zbigniew Kalbarczyk, Ravi Iyer

Center for Reliable and High-Performance Computing

University of Illinois at Urbana-Champaign, IL 61081

{basilecl, longwang, kalbar, iyer}@crhc.uiuc.edu

Abstract

Group communication protocols constitute a basic building block for highly dependable distributed applications. Designing and correctly implementing a group communication system (GCS) is a difficult task. While many theoretical algorithms have been formalized and proved for correctness, only few research projects have experimentally assessed the dependability of GCS implementations under complex error scenarios.

This paper describes a thorough error-injection experimental campaign conducted on Ensemble, a popular GCS. By employing synthetic benchmark applications, we stress selected components of the GCS—the group membership service, the FIFO-ordered reliable multicast, and the sequencer-based, total-ordered reliable multicast—under various error models, including errors in the memory (text and heap segments) and in the network messages.

The data show that about 5–6% of the failures are due to an error escaping Ensemble's error-containment mechanism and manifesting as a fail silence violation. This constitutes an impediment to achieving high dependability, the natural objective of GCSs. Our results are derived for a particular system (Ensemble), and more investigation involving other GCSs is required to generalize the conclusions. Nevertheless, through an accurate analysis of the failure causes and the error propagation patterns, this paper offers insights into the design and the implementation of robust GCSs.

1. Introduction

Reliable broadcasting and consistency of information are two core services in providing fault tolerance in a distributed networked environment. In-depth studies of these issues resulted in development of many group communication systems (GCSs).

Designing and correctly implementing GCSs is notoriously difficult. While in principle it is possible to formalize and prove the correctness of theoretical algorithms [1], it is very difficult to transform these idealizations into implementations that can be safely used in real systems. For a sound validation strategy, one should combine formal verification with model-based evaluation and error injection. In particular, error injection (carried out on the actual system or on a prototype) complements the other validation approaches by providing means for (1) removal of residual deficiencies in the GCS mechanisms, (2) assessing the validity of the assumptions made by the formal verification task, and (3) assessing a GCS's overall behavior in the presence of faults, in particular by estimating coverage and latency figures for the built-in error-detection mechanisms.

Thus far, few research projects have experimentally assessed

the dependability of GCS implementations [2], and often under relatively simple error scenarios [3–6]. For instance, [6] provides an experimental evaluation of a group communication protocol by injecting clean crash failures (i.e., by terminating the application process through a kill signal). In contrast, this paper provides a systematic, experimental study of GCS protocols under a variety of error models. The targeted system is Ensemble [7], which is a popular GCS developed at Cornell University. Ensemble was written in the OCAML dialect of the ML language so that the code would be amenable to automated proof checking. An automated construction of Ensemble's formal specifications from the source code is presented in [8]. To the best of our knowledge, no previous work has pursued a thorough characterization of Ensemble's behavior under real errors; notwithstanding, the system has been widely used.

The major findings of this paper include the following:

- A majority (94–95%) of memory errors (in text and heap segments) in the reliable communication layer result in crash/hang of the application process without incorrect data being sent out. More importantly, the remaining 5–6% of the errors do propagate and lead to multiple, correlated failures (and possibly to an entire system's failure).
- Errors in the messages exchanged among the processes lead, in many cases, to a crash of the receiving process(es). In addition, they can cause invalid data to be delivered to the application or application-level omission failures (up to 20% for atomic multicast). The frequency of the different failure categories depends on the position of the error in the corrupted message. The modeled errors originate in one process and propagate (as malformed messages) to other processes, causing them to fail.

It is important to note that although the percentage of the observed fail silence violations is relatively small (e.g., 5–6%), such errors do constitute an impediment to achieving high dependability because recovery from these failures can involve significant downtime in system operation. (For instance, in a replicated system based on GCS, a single replica failure can be masked transparently to the clients; in contrast, a crash of the entire system makes the service unavailable to all clients.) If high dependability is not a stringent requirement, then solutions less costly than group communication (e.g., checkpointing) should be considered.

Our results and analysis are not reported as an indictment of Ensemble, which is a well-engineered product, but to point out that the observed fail silence violations are outside the scope of the usual assumptions of crash/omission failures. Importantly, simply using protocols capable of handling application value

errors (e.g., Byzantine agreement) will not help cope with a significant portion of such failures. These failures are due to errors originating and propagating in the communication layer (i.e., Ensemble) and leading, for instance, to a corrupted header in the messages exchanged. To limit or prevent error propagation across the network, the middleware on which fault-tolerant techniques are based (in our case, the GCS), must itself be fault-tolerant [9].

The experiments presented in the paper target a particular GCS (Ensemble), and more investigation involving other GCSs is required to fully generalize the conclusions. Nevertheless, this study indicates that the analysis of the failure causes and error-propagation patterns offers valuable insights into the design and implementation of robust GCSs.

2. Related Work

Group communication has been extensively studied through formal analysis [1], but few papers have provided a detailed performance and dependability analysis of the algorithms. In [3], discrete event simulation is employed in a centralized approach to test the safety and timeliness properties of synchronous and asynchronous group membership services in the absence of failures. A number of studies apply stochastic modeling to characterize the failure behavior of fault-tolerance algorithms. Stochastic activity networks are used in [10] to evaluate the performability of a group-oriented multicast protocol and in [4] to analyze the latency of the Chandra-Toueg consensus algorithm.

For a sound validation strategy, formal verification and analytical modeling should be combined with error injection. The Messaline tool is used in [2] to evaluate the fail-silence properties of the Delta-4 fault-tolerant architecture through hardware-level error injection. In [11], an approach combining program analysis and fault injection is proposed to test fault-tolerant protocols. Doctor [12] uses a central process to control error injection experiments in a distributed environment under various error models. To support network injections, the Orchestra tool [13] inserts an additional layer below the layer under test by reimplementing the standard socket interface. It can thus simulate communication failures by filtering and altering the messages that pass through the test layer. Orchestra has been used to inject errors in a group membership protocol.

More recently, the unavailability introduced by the view change mechanism of a group communication protocol is evaluated in [6] by injecting clean crash failures. A Byzantine atomic broadcast protocol is experimentally evaluated in [5] through injection of crash failures and mutant messages, and by dropping messages. In [14], simulated crashes are injected in a simulation environment to compare the failure behavior of the Chandra-Toueg atomic broadcast (based on unreliable failure detectors) with that of a sequencer-based atomic broadcast (built on top of a group membership service).

Our study differs from previous work by focusing on the fail silence violations that are due to errors in the internals of a GCS and by providing a detailed analysis of the root causes of the observed failures, including error-propagation patterns.

3. Ensemble Overview

Ensemble provides a library of group communication protocols for building reliable distributed applications. The archi-

ture of the system is highly modular and is based on stacking micro-protocol layers, which can be combined to match the needs of a particular application. Layers implement several message-ordering properties, group membership properties, and virtual synchrony. The Ensemble group membership service supports member addition and removal as well as partition merging, failure detection, and suspicion sharing. The ordering properties for message delivery include sender-based FIFO, causal delivery, and total order delivery. In our experiments, we use a stack that provides membership service, virtual synchrony, and either FIFO ordering or sequencer-based total ordering, depending on the experiments. More detailed information on the Ensemble system can be found in [7].

4. Benchmarks

Depending on the user application, different Ensemble functions are activated with varying frequency. In order to determine the relative importance of the different subsystems and the most frequently used functions, we profile Ensemble with three synthetic benchmark applications. The benchmark programs serve two purposes: (1) to allow profiling the system to determine targets (e.g., most used functions) for the error injection campaigns and (2) to create system activity during the error injection campaigns to maximize chances for error activation.

Each benchmark application consists of three processes creating a reliable multicast group. The benchmarks are described below.

- The `group` benchmark is intended to exercise the membership service of the GCS, which provides a consistent view of non-failed processes in the system. The importance of this component stems from the fact that all other services offered by a GCS rely on the membership service. In the `group` benchmark, three processes join the same multicast group and remain inactive (from the application standpoint, there is no message exchange among the processes in the group). Notwithstanding its simplicity, we will show that this configuration does enable error propagation.
- The `fifo` benchmark is intended to exercise the FIFO-ordered reliable multicast, which is the basic multicast primitive offered by a GCS. In this benchmark, each of the three processes executes the following algorithm: (1) multicast a message to the group and (2) wait for receiving a message from each of the other group members. This sequence is repeated 1000 times. The goal of this benchmark is to study the behavior of the FIFO-ordered reliable multicast in presence of errors and then to compare it with the atomic multicast.
- The `atomic` benchmark implements an agreement algorithm on top of a total-ordered reliable multicast (also referred to as atomic multicast). Each of the three processes executes the following steps: (1) propose a binary value by multicasting one of two predefined messages to all group members and (2) on receiving a value/message from each group member (including itself), determine by a majority vote the final outcome at each process. As all processes receive messages in total order, in absence of failures they agree on the same value. This sequence is repeated 1000 times.

Atomic multicast is a fundamental block for implementing software-based replication. In our benchmark, the actual message content is immaterial, as no further computation is performed on the decided values. Nevertheless, the benchmark enables us to study the failures originating in the atomic multicast without the interference of failures due to application errors.

The Ensemble implementation of atomic multicast is based on a sequencer algorithm. A sender who wishes to initiate a totally ordered multicast, multicasts the message immediately (via FIFO-ordered reliable multicast). All receivers buffer the message until they receive a special accept message from the sequencer (by default, this is the group member with lowest id). The sequencer itself listens for all multicast messages sent on the network. As soon as the sequencer receives a multicast message m , it delivers m and multicasts the accept message for m . When a group member receives an accept message, it delivers the corresponding message. The overall effect is that all group members deliver messages in the order in which they are received/delivered by the sequencer.

5. Ensemble Profiling

All Ensemble’s functions are contained in a monolithic library of approximately 2.5 MB, which must be linked by an application that wishes to use Ensemble’s services. This library contains approximately 6000 functions, which correspond to Ensemble’s micro-protocols, Ensemble’s infrastructure management, and the run-time support of OCAML, the dialect of the ML language in which most of Ensemble is coded.

In general, an Ensemble-based application does not use all 6000 functions with the same frequency. By profiling the Ensemble library under the workload generated by the three benchmarks described in § 4, we can identify the most frequently used functions and determine the relative importance of the different subsystems of Ensemble. Profiling is done by using a ptrace-based tool we developed,¹ which does not require application recompilation. The user specifies an application binary and a list of function addresses (i.e., function entry points) to profile. Then, the tool executes the application and reports for each profiled function a *profiling value*, i.e., the number of times the application invoked the function.

Figure 1 shows the distribution of profiled functions among the Ensemble subsystems. Interestingly, a large portion (about 50%) of run-time function invocations are for the run-time support of OCAML. The second important portion (about 20%) corresponds to utility functions belonging to the Ensemble source code (*infr*, *util*, *type*, which are in charge of micro-protocol management, array and queue management, etc.). Only about 5% of function invocations are for the Ensemble micro-protocols (*layers*), which make up the part of a GCS that is usually formally specified and verified (e.g., in [8]). Among these micro-protocols, we determined that the most-used ones are MNAK (providing FIFO-ordered reliable multicast), BOTTOM (providing core networking primitives), SEQBB (used by *atomic* for the sequencer-based atomic multicast), and FRAG (providing message fragmentation services), in order of decreasing usage.

¹Ptrace is a process-tracing facility offered by UNIX operating systems.

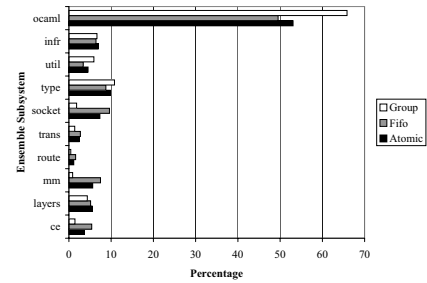


Figure 1. Run-time function invocations to Ensemble subsystems.

An important observation is that a large majority of function invocations are to components that typically are not objects of formal analysis. Therefore, we argue that formal analysis (such as model checking and theorem proving) and error-injection experimental evaluation must be considered as two complementary techniques in analyzing the failure behavior of a system.

6. Experimental Setup

The experimental testbed employed in this study consists of three nodes interconnected by an Ethernet 100 Mbps LAN. The nodes are Pentium III 500 MHz-based machines running Linux 2.4. Ensemble 1.40 [7] is used as the group communication system (GCS). In particular, we use Ensemble’s CE interface, which exposes Ensemble services to C/C++ applications. NF-TAPE [15], a software framework for conducting automated fault/error injection experiments, is used to conduct the tests.

Two major sets of error injection experiments are conducted: (1) memory injections, to assess the impact of errors in a process text- and heap-memory segments,² and (2) network injections, to analyze the impact of errors in the contents of messages exchanged in support of the communication protocols.

7. Error Injection into Process Memory

The experimental setup described in § 6 is used to study the impact of errors in the text- and heap-memory segments of the benchmark applications. An experiment consists of executing three copies of a selected benchmark (on three different machines) and injecting a single-bit error, at a random time, in a targeted process. The experiment concludes when all processes terminate.³ The system is reset between experiments.

Error Models and Outcome Categories. The error models considered are summarized in Table 1 and represent a combination of those used in several past experimental studies [17, 18]. By injecting single bits in the targeted process, we emulate errors in the main memory, the cache, the processor execution buffer, and the processor execution core, as well as errors occurring during transmission over a bus. Previous research on microprocessors [19] has shown that most (90–99%) device-level

²Errors are not injected in the static data memory segment, as our previous study [16] shows that such errors have very low activation rate for Ensemble.

³For the *group* benchmark, a correct process terminates when 15 seconds have passed from the time the multicast group is formed. This configuration allows us to analyze the effect of errors for a sufficient amount of time after an injection is performed and to run a large number of experiments in a reasonable amount of time.

transients can be modeled as logic-level, single-bit errors. Data on operational errors also shows that a large number of errors in the field are single-bit errors.

Table 1. Error models.

Error Model	Description
TEXT	A single bit in the text segment of the target process is flipped.
HEAP	A bit in allocated regions of the heap memory of the target process is flipped periodically, until the process terminates or crashes. Note that more than one error may be injected during a single experiment.

Manifested errors are divided in two major outcome categories: (1) *crash failures*, in which the injected process stops executing and no incorrect state transition is performed before the failure, and (2) *fail silence violations*, in which the injected process performs incorrect state transitions.⁴ The two categories and their corresponding subcategories are reported in Table 2.

Table 2. Outcome categories.

<i>Activated</i>	The corrupted instruction is executed.
<i>Manifested</i>	The corrupted instruction/datum is executed/used and does cause a visible abnormal impact on the system.
<i>Crash Failure</i>	<i>SIGNAL</i> , the operating system terminates the target process by sending a signal (e.g., SIGSEGV, SIGILL, SIGBUS, SIGFPE). <i>ASSERT</i> , the target process shuts itself down owing to an internal check violation detected in Ensemble. <i>HANG</i> , the target process does not terminate and does not make progress.
<i>Fail Silence Violation</i>	The target process misbehaves and possibly sends corrupted message to other processes, causing them to fail.

7.1. Memory Injection Results

Table 3 reports the results from error injection experiments for the three benchmark applications and for the text- and heap-error models listed in Table 1. For the `atomic` benchmark, we distinguish between injection in the sequencer process, *Atomic-SEQ*, and injection in a non-sequencer process, *Atomic-NSEQ*.

Text Injections. In this set of experiments, random bit errors are injected uniformly in the Ensemble functions that are executed more than once (i.e., presenting a profiling value greater than one), according to the profiling procedure delineated in § 5.⁵ A single error is injected per experiment, and only after the initial multicast group is formed. Key findings are summarized below.

- A large number of injected errors are not activated, i.e., the corrupted instruction is not executed. For instance, the activation rate (i.e., ratio between activated and injected errors) for `atomic` is 29–35%.
- Given that a corrupted instruction is executed (i.e., the error is activated), the data show that about 30% of the activated errors have no observable effect (i.e., do not manifest). This result is consistent with previous studies [20] and can be partially attributed to an inherent redundancy in the code.
- Table 3 shows that fail silence violations are rare for `group` (0.5%) but not absent, as the lack of application-level communication would suggest. The reason can be found in the underlying heartbeat occurring periodically

⁴This failure type covers cases such as corrupted data saved on persistent storage or a corrupted message sent to other nodes.

⁵The percentage of run-time function invocations covered by this criterion is 99.7% for `group` and (practically) 100.0% for both `fifo` and `atomic`.

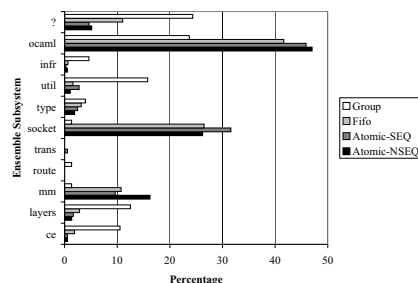


Figure 2. Crash location for heap-error injections.

between group members, which is used to detect process crashes and maintain a consistent group membership view. Note that for a quick failure detection, heartbeat messages should be sent often enough (e.g., the default setting for Ensemble is every second). We argue that the impact of the silent communication induced by the membership service should not be neglected when studying a GCS’s failure behavior, especially with respect to an application executing for a long period of time.

- The addition of communication among application processes significantly increases the chances of fail silence violations: 4% and 3–6% for `fifo` and `atomic` benchmark programs, respectively. This result shows that the frequency of fail silence violations is directly related to the number of messages exchanged over the network. Fail silence violations are further discussed in § 7.2.

Heap Injections. In this set of experiments, after the initial multicast group is formed, random bit errors are injected periodically in the allocated regions of the heap memory of a target process.

The heap-injection results of Table 3 show no occurrence of fail silence violations for the `group` benchmark. For the other two benchmarks, the presence of application-level communication makes fail silence violations account for 5% of the manifested errors, a number similar to that for the text-error injections (shown in the same table). Note the larger ratio of manifested errors to injected errors for the sequencer process. This result indicates a more intensive use of dynamic memory.

Figure 2 shows the distribution, in terms of Ensemble subsystems, of the crash locations (i.e., locations of the instruction causing the crash) for the heap-error injections. A crash occurring on an attempt to access an invalid memory location (e.g., due to a jump to an invalid page) is denoted by ‘?’’. One can see that for `fifo` and `atomic`, a large number of crash locations are within subsystems providing network services. For instance, the subsystem `socket`, which deals with socket abstraction, and the subsystem `mm`, which deals with memory management (e.g., allocation/deallocation of message buffers), jointly account for about 35% of the crash locations. On the other hand, most of the crashes for `group` are located in other Ensemble subsystems (e.g., `util`, `layers`, `ce`); this is because the communication required by this benchmark is minimal (primarily due to process heartbeating). Note that for all benchmarks, about 25–45% of the crashes are located in the OCAML run-time support, `ocaml`.

Table 3. Text- and heap-error injection in all Ensemble subsystems.

Group	Error Model	Total Injected Errors	Total Activated Errors	Total [†] Manifested Errors	Manifested Errors [‡]			
					Crash Failures			Fail Silence Violations
					SIGNAL	ASSERT	HANG	
Group	TEXT	6792	3223	2286 (71%)	2019 (88%)	223 (10%)	33 (1.5%)	11 (0.5%)
	HEAP	20390	N/A	177 (0.87%)	151 (85%)	26 (15%)	0	0
Fifo	TEXT	6882	2337	1671 (72%)	1425 (85%)	127 (8%)	46 (3%)	73 (4%)
	HEAP	14930	N/A	387 (2.6%)	309 (80%)	48 (12%)	9 (2%)	21 (5%)
Atomic-SEQ	TEXT	7401	2583	1878 (73%)	1604 (85%)	142 (8%)	27 (1%)	105 (6%)
	HEAP	11278	N/A	412 (3.7%)	352 (85%)	26 (6%)	13 (3%)	21 (5%)
Atomic-NSEQ	TEXT	7267	2106	1503 (71%)	1272 (85%)	170 (11%)	21 (1%)	40 (3%)
	HEAP	15673	N/A	414 (2.6%)	367 (89%)	22 (5%)	3 (1%)	22 (5%)

[†] The ratios manifested/activated and manifested/injected are shown in parentheses for text and heap injections, respectively.

[‡] The percentage of the particular manifestation type with respect to the total number of manifested errors is shown in parentheses.

7.2. Fail Silence Violations

This section discusses the fail silence violations observed in the text- and heap-error injections. A majority of the fail silence violations due to heap errors are due to a corrupted application message being sent/received.⁶ In contrast, a detailed analysis of the text-error injections indicates that about 40–80% of the fail silence violations for `atomic` and `fifo` are due to application-level omission failures, i.e., cases in which a process omits sending/receiving an application message that it was supposed to send/receive. The application does not usually cope with such failures, since reliable communication is supposed to be provided by the underlying GCS. Note that these omission failures are different from the omission failures experienced by the GCS, which are detected and recovered from transparently to the application by means of sequence numbers and retransmissions.

A run of the `atomic` and `fifo` benchmarks involves the execution of a sequence of 1000 iterations (see § 4), during each of which a process expects a message from the other group members. If an application-level omission failure occurs, then the process waits forever for the message to arrive and never completes the iteration. Interestingly, most of the observed omission failures occur many iterations after the iteration in which the error is injected (e.g., 100 iterations, which corresponds to 300 application messages being exchanged). An example of such a failure is discussed at the end of this section.

The data also show that in 15 cases of fail silence violations due to text errors, the injected process does not fail (an example of such a failure is discussed at the end of this section) but causes other processes to fail (e.g., by raising an exception or due to segmentation violation).

Figure 3 depicts the probability of a fail silence violation due to an activated error in the text segment of the various Ensemble subsystems (e.g., an error in `ocaml` is likely to activate a fail silence violation with similar probability for all benchmarks). The figure provides useful feedback to a GCS designer as to which subsystems are most error-sensitive and, thus, require careful implementation and extensive verification.

The following are two actual failure scenarios of fail silence violations selected from the text-error injection campaign:

Scenario 1. In the experiments employing the `atomic` benchmark program, a fault is injected in the function `Mflow_up_hdr_354` of the MFLOW layer, which implements window-based flow control for multicast messages and

⁶These failures are detected by dedicated assertions we added to the benchmark code.

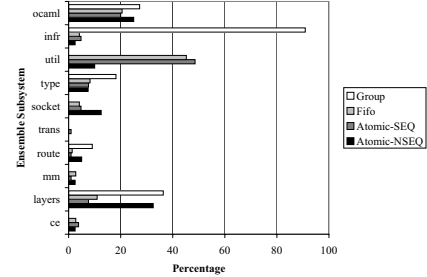


Figure 3. Distribution of the origin of fail silence violations.

is based on the notion of *credit*. Each process is allowed to send multicast messages only if the number of its send credit is greater than zero. The send credit is decreased when a message is sent; whenever the amount of send credit drops to zero, messages are buffered without being sent. The mechanism is such that a receiver sends an acknowledgment credit to a sender only after receiving a fixed number of bytes of data (e.g., 50KB) from the sender. This number is called the *acknowledgment threshold*. On receipt of an acknowledgment credit, the amount of send credit is increased, and buffered messages are sent based on the new credit.

The source code of the MFLOW layer’s function `up_hdr` is listed below. This function corresponds to the behavior of a process receiving a message and deciding whether to send an acknowledgment credit to the sender.

```

1 let up_hdr ev abv hdr = match getType ev, hdr with
2 | ECast iovl, NoHdr ->
3   let origin = getPeer ev in
4   if origin <>| ls.rank then (
5     let len = msg_len s iovl in
6     let current_credit = Mcredit.get_msg s.credit origin len in
7     if current_credit >=| s.ack_thresh then (
8       let nacks = current_credit / s.ack_thresh in
9       let remainder = current_credit - (nacks*s.ack_thresh) in
10      dnln (sendPeer name origin) (Ack nacks);
11      Mcredit.set_credit s.credit origin remainder
12    )
13  );
14 up ev abv
15
16 | ECast _, NoFlow -> up ev abv
17 | _, NoHdr -> up ev abv
18 | - -> failwith bad_header

```

The injected instruction “`mov 0xffffffff(%ecx, %ebx, 2), %eax`” is used at line 6 in the process of assigning `current_credit`, a variable maintaining the credit of the message sender, `origin`. On flipping one of the bits, the instruction changes to “`mov 0xffffffffbe(%ecx, %ebx,`

2), %eax”. Consequently, the resulting `current_credit` is assigned to an invalid value—which happens to be less than the acknowledgement threshold, `s.ack_thresh`—and does not reflect any increase of the sender credit. Eventually the sender consumes all its credit, which may involve sending several messages. Since the error will prevent lines 8–11 from executing, the receiver will not send more acknowledgment credits, and once the sender has consumed its sender credit, the entire multicast group hangs. Importantly, the group membership service does not detect process failures because its messages are not subject to the MFLOW protocol.

Scenario 2. A fault is injected in the function `writencode16` while running the `group` benchmark. This function is provided by the OCAML run-time support for marshaling the ML objects included in the Ensemble messages. The source code of `writencode16` is listed below.

```

1 static void writencode16(int code, long int val)
2 {
3   if (extern_ptr + 3 > extern_limit) resize_extern_block(3);
4   extern_ptr[0] = code;
5   extern_ptr[1] = val >> 8;
6   extern_ptr[2] = val;
7   extern_ptr += 3;
8 }

```

The corrupted instruction is “`mov 0x10(%esp, 1), %eax`”, which corresponds to the assignment statement at line 4. Note that `code` corresponds to the stack location `0x10(%esp, 1)`, and `%eax` is a register later assigned to `extern_ptr[0]`. On flipping one of the bits, the instruction becomes “`mov 0x12(%esp, 1), %eax`”. Consequently, an invalid value is assigned to `extern_ptr[0]`. Note that the code information is crucial for the marshaling/unmarshaling of ML objects because it includes the object type and the object size (as described in § 8.1).

When a recipient of the resulting message unmarshals the corrupted ML object, a buffer overflow occurs. This is because the OCAML unmarshal function, `intern_rec`, does not perform any boundary check on the buffer used for supporting unmarshaling while the unmarshaling process is in progress. (The size of the unmarshaling buffer is determined when an Ensemble message is received and is based on a field included in the message header.)

As a result of the buffer overflow, `intern_rec` can overwrite unallocated as well as perfectly valid memory regions, leading to corruption of the process’s memory. In this experiment, all group members except the injected process eventually crash in the function `chunk_free`, which is called by `free`, the standard memory deallocation function.

7.3. Error Latency

Much work in reliable distributed computing is based on benign failure semantics (i.e., crash/omission failures). Despite its vast acceptance, experimental studies have shown that assuming such a failure model for the underlying communication layer is an impediment to achieving high dependability. For example, [16] shows that in a significant number of cases, errors originating in the communication layer can propagate and lead to catastrophic failures of an entire replicated system. Understanding the error-propagation patterns between the location of

an error and that of a process failure is vital to maintaining system integrity. This section discusses error propagation in Ensemble. Initially, we consider only errors whose propagation is limited to the injected process; error propagation involving the network is considered separately in § 8.

Figure 4 shows the error latency distribution for the text-error injections of Table 3. Four levels of latency are considered: (1) *Same Location*, the injected process crashes at the time of executing the injected instruction; (2) *Same Function*, the injected process crashes while executing code from the injected function (excluding the previous case); (3) *Different Function*, the injected process crashes while executing a function that is not the injected one; (4) *Invalid Location*, the injected process crashes due to an attempt to access an invalid memory location (e.g., due to a jump to an invalid page).

Figure 4 clearly indicates that failure propagation does occur. In about 20–25% of the cases, the process crashes in a different function, which contributes to the probability of performing invalid computation before crashing, while in 55% of the cases, the process crashes immediately or almost immediately after executing the corrupted instruction. (In [20] it is shown that 40% of the crashes occur within 10 CPU cycles of the corrupted instruction and that about 20% of the crashes have latency longer than 100,000 cycles.)

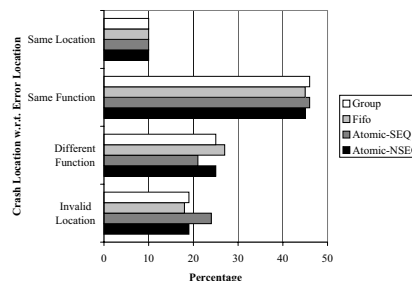


Figure 4. Text-error latency.

To refine our analysis, we show in Figure 5 the probabilities for a text error occurring in one subsystem to propagate and manifest as a crash in another subsystem.⁷ The data are from the experiments corresponding to Table 3. The left node on each graph refers to the faulted subsystem (i.e., the subsystem where an error is injected). The outgoing arcs (transitions) indicate error-propagation paths (including the self loop). The destination node of each transition corresponds to the subsystem where the crash occurred. The major findings from the error-propagation analysis are as follows:

1. The overall percentage of error propagation to different subsystems is substantial, about 20%, with most subsystems having a large percentage. The result heavily depends on how subsystems are defined. In this paper, we have chosen to divide Ensemble into subsystems according to its source tree structure. Although arbitrary, our choice aims to delimit portions of code serving different purposes.
2. Several critical error-propagation paths can be identified,

⁷The analysis of error propagation has been conducted for other Ensemble subsystems and for all benchmarks used in this study. Due to space limitations, we only report data for the selected subsystems while running the Atomic-SEQ benchmark.

e.g., from virtually any subsystem to *ocaml*. A closer analysis of the propagation patterns indicates that it is feasible to identify strategic locations for embedding additional internal checks (i.e., assertions) to detect the errors and, hence, prevent them from propagating, e.g., through the network. By making the application process fail-fast, one can limit the possibility of corruption of the global system state. (Embedding assertions based on error-propagation analysis has been suggested in [21].)

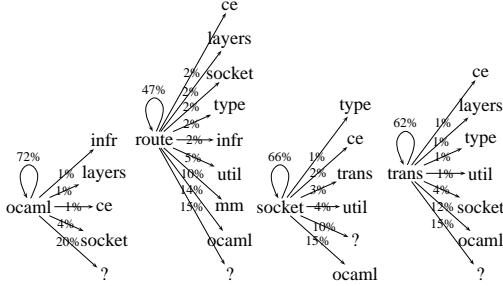


Figure 5. Text-error propagation across subsystems for Atomic-SEQ benchmark.

8. Network Injection

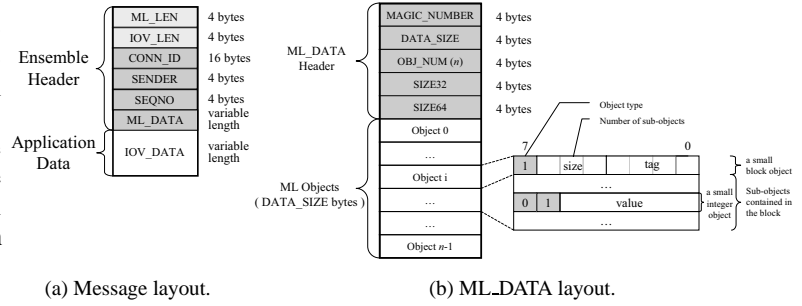
This section discusses the impact of errors in the messages exchanged by Ensemble processes. A single-bit error is injected in a randomly selected message sent by a target process; consequently, the corrupted message propagates to other processes and can possibly cause a failure at the receiving end. In this way, we model failures occurring in the Ensemble system leading to malformed messages being sent/received. Note that the modeled errors occur before/after any encoding, e.g., checksum,⁸ is applied/removed to protect messages against transmission errors. The goals are: (1) to test the robustness of Ensemble with respect to erroneous input data (e.g., corrupted network messages) and (2) to provide further insight into the error propagation mechanisms leading to the fail silence violations observed in the memory injection experiments.

8.1. Message Format

Before presenting the results from error injection experiments, we introduce the Ensemble message format. This discussion is essential in analyzing and explaining the observed system behavior, e.g., error propagation. Figure 6(a) shows the layout of an Ensemble message. `ML_LEN` is the total length of the Ensemble-specific data, ranging from `CONN_ID` to `ML_DATA`. `IOV_LEN` is the length of `IOV_DATA`, the application data (i.e., the actual data exchanged between the application and the Ensemble’s send/receive primitives). `CONN_ID` identifies the Ensemble connection to which the message is addressed and is an MD5 hash value.⁹ `SENDER` is the sender’s rank in the multi-

⁸Ensemble supports a security mode in which all group messages are signed and (possibly) encrypted to secure them from tampering and eavesdropping. In this mode, message corruption occurring over the network is detected at the receiving end as an invalid message signature.

⁹The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit *fingerprint* or *message digest* of the input. The MD5 algorithm is intended for digital signature applications, where a large file must be compressed in a secure manner before being encrypted with a private key under a public-key cryptosystem such as RSA.



(a) Message layout.

(b) ML_DATA layout.

Figure 6. Ensemble message format.

cast group (ranging from zero to the group size minus one), and `SEQNO` is a sequence number set by the sender.

The internal format of `ML_DATA` is shown in Figure 6(b). This field contains several ML objects piggybacked by Ensemble, including message headers, header parameters, arrays, etc. To send out and read in ML objects, Ensemble uses the standard marshal/unmarshal mechanism provided by the run-time support of OCAML.¹⁰

The first byte of a marshaled ML object is named *code* and includes the object’s type. Possible types include: block, integer, string, float, etc. (Integers with different sizes are considered to be different types and, hence, use different code.) Depending on the type, an object can have various attributes, such as *size*, *tag* (used by the marshaling/unmarshaling mechanism), or *value*. Figure 6(b) includes, as an example, a marshaled small block object in which the *size* attribute indicates the number of sub-objects contained in the block. Note that the OCAML’s marshal/unmarshal mechanism is quite fragile to errors. For Ensemble messages, only the bit pattern used to determine the object type is actually checked.¹¹ If a corrupted ML object is found, Ensemble drops the message. Otherwise, that object plus all the following ones are incorrectly reconstructed by the unmarshal mechanism.

8.2. Error Models and Outcome Categories

The network injection campaign is performed by flipping a random bit in a message sent by the target process. The target message is selected randomly from the (application and control) messages exchanged during a single experiment. A separate set of experiments is performed to corrupt each message field shown in Figure 6(a), except for `IOV_DATA`.¹²

Table 4 reports outcome categories for network-error injection campaigns. Since a network injection occurs when a target process sends a message to other group member(s), all failure scenarios covered in the Table 4 consider the failure of a recipient of the corrupted message. Note also that failures corresponding to `INVALID BEHAVIOR` and `APPLICATION-LEVEL OMISSION` categories are detected, respectively, by means of dedicated assertion checks we embedded in the benchmark code and by noting that the application cannot complete an experiment (although the processes do not crash) because of

¹⁰The following description is based on the source code of the OCAML 3.06 compiler.

¹¹Among the 256 possible bit patterns for an object type, only 13 constitute invalid combinations.

¹²The effect of corrupted application data is application-specific and is beyond the scope of this study.

the omission.¹³

To perform the network injection experiments, we have extended the NFTAPE framework [15] to include a network-error injector. The injection mechanism used is similar to that proposed in [13]. By inserting an additional layer between Ensemble and the standard socket interface of the operating system, we can alter the messages that pass through the layer.¹⁴ This allows us to monitor messages exchanged by Ensemble processes with minimal intrusiveness.

Table 4. Outcome categories for network injections.

<i>Not Manifested</i>	The injected error does not cause a visible abnormal impact on the system.
<i>Manifested</i>	<i>SIGNAL</i> , the operating system terminates the recipient process by sending a signal (e.g., SIGSEGV, SIGBUS, SIGFPE). <i>ASSERT</i> , the recipient process shuts itself down owing to an internal check violation detected in Ensemble. <i>INVALID BEHAVIOR</i> , the recipient process detects invalid behavior of some process: it receives (i) an invalid message or (ii) more than two messages from the same process in the same round. <i>APPLICATION-LEVEL OMISSION</i> , a recipient process omits sending/receiving a subsequent application message.

8.3. Network Injection Results

The results of our network injection experiments are reported in Table 5 and are discussed below.

1. *Corruption of the ML_LEN and IOV_LEN fields.* On receiving a new message, Ensemble always checks the validity of the ML_LEN and IOV_LEN fields. Example validity criteria include consistency with the actual message length (as returned by the socket receive function), consistency with the maximum message length (8KB), and consistency with the DATA_SIZE field included in ML_DATA (only for ML_LEN).

If a corrupted ML_LEN/IOV_LEN field is negative, then Ensemble raises an exception and the process terminates. If a corrupted IOV_LEN indicates less application data than it should, an invalid application message is detected by the assertion checks embedded in the benchmark code (see § 4). In all the other cases, which constitute a majority in our experiments, Ensemble simply drops the message. This explains the small number of manifested errors in the ML_LEN and IOV_LEN experiments.

2. *Corruption of the CONN_ID field.* All cases of corrupting CONN_ID cause a message to be dropped. Indeed, since a connection id is a 16-byte MD5 hash value, it is very unlikely for a single-bit error to change a valid CONN_ID to another valid one.
3. *Corruption of the SENDER field.* The vast majority (97–99%) of corruptions in the sender field cause a process to crash. This is because Ensemble never checks the validity of this field, although its value is often used to index various arrays in micro-protocols (e.g., MNAK and BOTTOM). A detailed example is provided in § 8.4.

¹³If a process omits sending/receiving a message during one of the 1000 iterations it goes through in each experiment (see § 4), then the process cannot complete that iteration and, consequently, the group hangs.

¹⁴In practice, the injector is in a shared library that overloads, at run-time, the operating system’s shared library that offers the socket interface.

4. *Corruption of the SEQNO field.* SEQNO is only used in `fifo` benchmark program, and hence, we observed errors only for this benchmark. To deliver messages in FIFO order, each process maintains an array (reorder buffer) in which the received messages are stored. SEQNOs are used to index the messages in the reorder buffer. Two variables, `lo` and `hi`, point to the next message to deliver and to the received message with the largest SEQNO, respectively.

On receiving a message, Ensemble checks whether the difference between the SEQNO of currently received message and `lo` is greater than a predefined constant ($2^{22} - 1$ in our experiments). If so, an exception is raised. If the SEQNOth entry is already occupied (on a receiving end) or SEQNO is less than `lo`, the message is dropped. Otherwise, the message is stored in the SEQNOth entry.

An interesting failure may occur when a message is received with an erroneous SEQNO pointing to an empty entry in the reorder buffer. In this case, the received message is placed in an incorrect location of the reorder buffer. Eventually, the receiver will ask for retransmission of the original message, since the corresponding spot in the reorder buffer remains empty. However, when the legitimate message (i.e., message with SEQNO pointing to the location occupied by the erroneously placed message) arrives, it is dropped because its entry has already been taken. Consequently, the erroneous message is delivered to the application in place of the correct one. We did not observe this scenario in our experiments, probably because of the potentially large latency for this error to manifest.

5. *Corruption of the ML_DATA field.* Unmarshaling a corrupted ML_DATA potentially results in a set of incorrect objects, i.e., objects different than those in the original message. As a consequence, the application process suffers from memory corruption and can eventually crash. Due to the fact that reconstructed objects are used by multiple functions, a process can crash while executing different functions. The actual error propagation path depends on the particular error and the injection time.

In many experiments, the process crashes in `chunk_free`, which indicates corruption of the process’ memory. This occurs when an error affects the type or size information of a marshaled object contained in ML_DATA. This case is similar to the second scenario described in § 7.2.

The analysis of failure causes and error-propagation paths presented above can be used to develop new mechanisms for enhancing robustness of the group communication system. Based on our study, example improvements would include: (1) introduction of range checks on the SENDER field (in the Ensemble messages) used in indexing an array and (2) employment of a more robust encoding for marshaled ML objects, e.g., by means of object delimiters. Furthermore, the data in Table 5 indicate that a majority of crashes occur in a small subset of the Ensemble functions; therefore, it is desirable (and practical) to harden the implementation of such functions.

8.4. Example Failure Scenarios

This section details two failure scenarios selected from the experiments of Table 5.

Table 5. Network-error injection results.

Targeted Message Field	Benchmark Application	Total Injected Errors	Total [†] Manifested Errors	Manifested Errors [‡]				Most Frequent Crash Locations ^{††}
				SIGNAL	ASSERT	INV BEH	APP-LEV OMS	
ML_LEN	Group	999	34 (3.4%)	0	34 (100%)	0	0	
	Fifo	1000	36 (3.6%)	0	36 (100%)	0	0	
	Atomic	999	33 (3.3%)	0	33 (100%)	0	0	
IOV_LEN	Group	1000	23 (2.3%)	0	23 (100%)	0	0	
	Fifo	1000	121 (12%)	0	27 (22%)	94 (78%)	0	
	Atomic	999	109 (11%)	0	23 (21%)	86 (79%)	0	
CONN_ID	Group	999	0	0	0	0	0	
	Fifo	1000	0	0	0	0	0	
	Atomic	1000	0	0	0	0	0	
SENDER	Group	999	419 (42%)	414 (98.8%)	5 (1.2%)	0	0	Bottom_up_hdr_279 (90%), Mnak_up_hdr_323 (5%), Pt2pt_handle_data (3%), ...
	Fifo	1000	741 (74%)	720 (97.2%)	21 (2.8%)	0	0	
	Atomic	999	654 (65%)	652 (99.7%)	2 (0.3%)	0	0	
SEQNO	Group	999	0	0	0	0	0	
	Fifo	1000	198 (20%)	0	198 (100%)	0	0	
	Atomic	997	0	0	0	0	0	
ML_DATA	Group	999	178 (18%)	169 (95%)	9 (5%)	0	0	chunk_free (46%), ? (15%), Array_to_list_146 (9%), ...
	Fifo	894	119 (13%)	104 (87.4%)	11 (9.2%)	0	4 (3.4%)	
	Atomic	999	189 (19%)	124 (66%)	11 (6%)	15 (8%)	39 (20%)	

[†] The ratio between manifested and injected errors is shown in parentheses. Note that in the network injections, each injected error is activated.

[‡] The percentage of the particular manifestation type with respect to the total number of manifested errors is shown in parentheses.

^{††} The percentage of the particular location with respect to the total number of crash errors is shown in parentheses.

Scenario 1. In an experiment for the `group` benchmark, an error is injected in the `sender` field of a message sent by the targeted group member. Because of the error, the `sender` field is changed from 2 to 8388610.

When the corrupted message is received by another group member, the message traverses unchecked through several functions. Then, the message is converted into an Ensemble event (in the file `config_trans.ml`) and is passed up to the Ensemble stack. Eventually, the process crashes at line 4 of the `BOTTOM` layer’s function `up_hdr`, shown below.

```

1 let up_hdr ev abv hdr = match getType ev, hdr with
2 | (ECast iov | ESend iov), NoHdr->
3   if s.all_alive
4     || not (Arrayf.get s.failed (getPeer ev))
5   then (* Common case: origin was alive
6         *)
7     up ev abv
8   else
9     Iovecl.free iov
10  | ECast iov, Unrel->
11  ...
12  | ESend iov, Unrel->
13  ...
14  | EAuth, _->
15  ...
16  | _-> eprintf "%s\n" (Event.to_string ev) ;
17      failwith bad_up_event

```

At line 4, the code “`getPeer ev`” extracts the `sender` field from the corrupted event `ev`. The sender information is then passed as an index to the function `Arrayf.get` to extract the corresponding element from the array `s.failed`. The array records whether group members are considered as faulty. Because in our experiments only three members join the multicast group, only three entries are allocated for `s.failed`. Therefore, when `Arrayf.get` attempts to access the 8388610th entry, a segmentation violation occurs. As a result of this error, all group members except the injected process member crash.

Scenario 2. In an experiment for the `group` benchmark, an error is injected in the `ML_DATA` field of a message sent by the targeted group member (i.e., the process where the error is injected). This field contains several marshaled ML objects. The injected error causes the size information of the 11th object in `ML_DATA` to be changed from 3 to 1.

When the corrupted message is received by another group member, the unmarshaling process gets out of sync because of the incorrect size of object 11 and creates invalid ML objects. In particular, while the original `ML_DATA` contained two ML vectors (object 11 and object 12) used by the `STABLE` layer, the reconstructed ML objects include only the first vector, which is shortened from 3 elements to 1.

In the `STABLE` layer, each group member p gossips to the multicast group a vector `remote` containing, for each group member q , the sequence number of the last message that p has acknowledged to q . Occasionally, p also includes in the gossip message a vector `failed`, whose elements indicate the group members that p considers as failed. An extract from the `STABLE` source code is shown below.

```

1 and uplm_hdr ev hdr = match getType ev, hdr with
2 (* Gossip Message: if from a live member, copy into
3 * the origins row in my acknowledgement matrix. *)
4 | (ECast _ | ESend _ | ECastUnrel _ | ESendUnrel _),
5 Gossip(remote, failed)->
6   let origin = getPeer ev in
7   if (not (Arrayf.get failed ls.rank))
8     && (not (Arrayf.get s.failed origin))
9   then (
10    let local = s.acks.(origin) in
11    for i = 0 to pred ls.nmembers do
12      let remote = Arrayf.get remote i in
13      if remote >| local.(i) then
14        local.(i) <- remote
15    done ;
16  ) ;
17  free name ev
18
19 | _-> failwith unknown_local

```

At line 5, `remote` and `failed` are extracted from the event `Gossip`. Note that when a group member receives a corrupted

Gossip event, remote has been shortened by the unmarshaling process and failed points to an invalid memory location. Since the program does not perform any check, it uses the two vectors as if the Gossip event were correct. At line 7, the process attempts to access the array failed, and hence, a segmentation violation occurs. As a result of this error, all group members except the targeted member crash.

9. Conclusions

This paper describes a thorough error-injection experimental campaign conducted on Ensemble, a popular group communication system (GCS). By employing synthetic benchmark applications, we stress selected components of the GCS: the group membership service, the FIFO-ordered reliable multicast, and the sequencer-based total-ordered reliable multicast. This is done under various error models, including errors in the memory (text and heap segments) and in the network messages.

A majority of failures (95%) can be modeled with clean crash failures, and about 5–10% of these failures are detected by the assertion checks embedded in Ensemble. Importantly, about 5% of the failures occur when an error escapes Ensemble’s error-containment mechanism and manifests as a fail silence violation. Examples include cases in which (1) the faulty node suffers from an application-level omission failure and (2) one or more non-injected processes fail while executing Ensemble code either by raising an exception or because of segmentation violation. Although the actual percentage is small, such errors do constitute an impediment to achieving high dependability, the natural objective of GCSs.

It is important to observe that simply using protocols capable of handling application value errors (e.g., Byzantine agreement) will not help cope with the large portion of the observed fail silence violations that originate and propagate in the communication layer. To limit or prevent such failures, the middleware on which fault-tolerant techniques are based (in our case, the reliable communication layer) must itself be fault-tolerant.

The results of our study are derived for a particular system (Ensemble), and more investigation involving other GCSs is required to generalize the conclusions. Nevertheless, through an accurate analysis of the failure causes and the error propagation patterns, this paper offers valuable insights for the design and the implementation of robust GCSs.

Acknowledgments

This work is supported in part by NSF grants CCR 00-86096 ITR and CCR 99-02026. We thank Fran Baker for insightful editing of our manuscript.

References

- [1] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.
- [2] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. on Computers*, 42(8):913–923, 1993.
- [3] G. A. Alvarez and F. Cristian. Simulation-based test of fault-tolerant group membership services. In *Proc. of the Annual Conf. on Computer Assurance*, 1997.

- [4] A. Coccoli, P. Urban, A. Bondavalli, and A. Schiper. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2002.
- [5] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2002.
- [6] K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental evaluation of the unavailability induced by a group membership protocol. In *Proc. of the European Dependable Computing Conf.*, pages 140–158, 2002.
- [7] M. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, USA, 1997.
- [8] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. *Lecture Notes in Computer Science*, 1421:317–332, 1998.
- [9] K. Whisnant, R. Iyer, P. Jones, R. Some, and D. Rennels. An experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, pages 585–595, 2002.
- [10] L. Malhis, W. Sanders, and R. Schlichting. Numerical evaluation of a group-oriented multicast protocol using stochastic activity networks. In *Proc. of the Int’l Workshop on Petri Nets and Performance Models*, pages 63–72, 1995.
- [11] K. Ehtle and Y. Chen. Evaluation of deterministic fault injection for fault-tolerant protocol testing. In *Proc. of the Symp. on Fault-Tolerant Computing*, pages 418–425, 1991.
- [12] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proc. of the Int’l Computer Performance and Dependability Symp.*, pages 204–213, 1995.
- [13] S. Dawson, F. Jahanian, T. Mitton, and T. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proc. of the Symp. on Fault-Tolerant Computing*, pages 404–414, 1996.
- [14] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2003.
- [15] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer. Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE. In *Proc. of the Int’l Computer Performance and Dependability Symp.*, 2000.
- [16] C. Basile, Z. Kalbarczyk, and R. Iyer. Preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2003.
- [17] E. Fuchs. Validating the fail-silence assumption of the MARS architecture. In *Proc. of the Dependable Computing for Critical Applications Conf.*, pages 225–247, 1998.
- [18] H. Madeira and J.G.Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *Proc. of the Int’l Symp. on Fault-Tolerant Computing*, pages 350–359, 1994.
- [19] M. Rimen, J. Ohlsson, and J. Torin. On microprocessor error behavior modeling. In *Proc. of the Int’l Symp. on Fault-Tolerant Computing*, 1994.
- [20] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux kernel behavior under errors. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2003.
- [21] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proc. of the Int’l Conf. on Dependable Systems and Networks*, pages 135–144, 2002.