Group Communication Protocols under Errors

Claudio Basile
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign, IL 61081
basilecl@crhc.uiuc.edu

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 (GCS) [1].

Designing and correctly implementing GCSs is notoriously difficult. While in principle it is possible to formalize and prove the correctness of theoretical algorithms, 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 works 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 it would be amenable to automated proof checking—an automated construction of its 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). Although the percentage is small, 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; on the contrary, 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.

It should be noted that our results and analysis are not reported as an indictment of Ensemble, which is a well-engineered product. Our intention is to point out that these failures are due to fail silence violations and error propagations, which are outside the scope of the usual assumptions of crash/omission failures. In addition, simply using protocols capable of handling application value errors (e.g., interactive consistency) will not help cope with the observed fail silence violations; these failures are due to errors originating 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 reliable communication layer), must itself be fault-tolerant.

2. Experimental Setup

Ensemble [7] provides a library of group communication protocols whose architecture is based on stacking micro-protocol layers. 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.

Benchmarks. 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: `group`, in which 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); `fifo`, in which three processes exchange messages though the FIFO-ordered reliable multicast; `atomic`, which implements an agreement algorithm on the top of a total-ordered reliable multicast.[1] The benchmark applications also serve the following purposes: (1) 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.

Profiling. All Ensemble's functions are contained in a monolithic library of approximately 2.5 MB, containing approximately 6000 functions, which correspond to Ensemble's micro-protocols, Ensemble's infrastructure management, and the runtime support of OCAML, the dialect of the ML language in which most of Ensemble is coded.

By profiling the Ensemble library under the workload generated by our three benchmarks, we identify the most frequently

[1] Total-ordered reliable multicast is also referred to as atomic multicast.

Table 1. Text- and heap-error injection in all Ensemble subsystems.

	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.

used functions. Interestingly, 50% of run-time function invocations are for the run-time support of OCAML. About 20% of invocations corresponds to utility functions belonging to the Ensemble source code, which are in charge of micro-protocol management, array and queue management, etc. Only about 5% of invocations are for the Ensemble micro-protocols, which make up the part of a GCS that is usually formally specified and verified (e.g., in [8]).

3. Error Injection into Process Memory

Table 1 reports the results from our error injection experiments.[2] In the text-error experiments, random bit errors are injected uniformly in the Ensemble functions that are executed more than once.[3] A single text-error is injected per experiment, and only after the initial multicast group is formed. In the heap-error experiments, a bit in allocated regions of the heap memory of a target process is flipped periodically.

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]

Key findings from data in Table 1 are summarized below:

- Fail silence violations are rare for `group` but not absent, as the lack of application-level communication would suggest. The reason can be found in the underlying heartbeat occurring periodically between group members, which is used to detect process crashes and maintain a consistent group membership view.
- The addition of communication among application processes significantly increases the chances of fail silence violations. This result shows that the frequency of fail silence violations is directly related to the number of messages exchanged over the network.

4. Fail Silence Violations

Fail silence violations include cases in which one or more non-injected processes fail while executing Ensemble code either by raising an exception or due to segmentation violation.

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. These omission failures are different from the omission failures of the underlying GCS, which are detected and recovered transparently to the application by means of sequence numbers and retransmissions. The application does not usually cope with such omission failures, since reliable communication is supposed to be provided by the underlying GCS. Interestingly, most of the observed omission failures occur only after many application messages (e.g., 300) have been exchanged since the error is injected .

For the heap-error injections, a majority of fail silence violations are due to a corrupted application message being sent/received.[5] In addition, 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).

Figure 1 depicts the probability of a fail silence violation due to an activated error in the text segment of the various Ensemble subsystems (e.g., `ocaml`, the run-time support of 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.

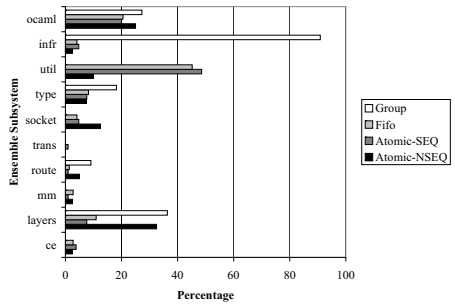

Figure 1. Origin of fail silence violations.

5. Error Latency

Despite its vast acceptance, experimental studies have shown that assuming a benign failure model for the underlying communication layer is an impediment to achieving high dependability [9]. Understanding the error-propagation patterns between

[2] For the `atomic` benchmark, we distinguish between injection in the sequencer process, *Atomic-SEQ*, and injection in a non-sequencer process, *Atomic-NSEQ*.

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

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

[5] These failures are detected by dedicated assertions we added to the benchmark code.

the location of an error and that of a process failure is vital to maintaining system integrity.

Figure 2 shows the error location distribution, with respect to the injected function, for the text-error injections of Table 1. [6] The figure 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.

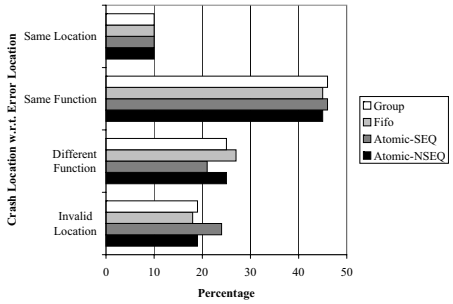

Figure 2. Text-error latency.

Figure 3 shows probabilities for a text error occurring in one subsystem to propagate and manifest as a crash in another subsystem.[7] The left node on each graph refers to the subsystem where an error is injected. The outgoing arcs indicate error-propagation paths (including the self loop). The destination node of an arc corresponds to the subsystem where the crash occurred.[8] The major findings include:

1. The overall percentage of error propagation to different subsystems is about 20%.

2. Several critical error-propagation paths can be identified, e.g., from virtually any subsystem to *ocaml*, the run-time support of OCAML. It is feasible to identify strategic locations for embedding additional 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 [10].

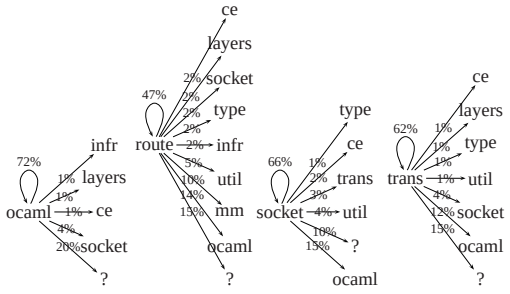

Figure 3. Text-error propagation.

[6] *Invalid location* denotes an injected process crashing due to an attempt to access an invalid memory location (e.g., due to a jump to an invalid page).

[7] Due to space limitations, the data is reported only for some Ensemble subsystems and for the `atomic` benchmark.

[8] A crash occurring due to an access to an invalid memory location is denoted by '?'.

6. Conclusions

This paper describes a thorough error-injection experimental campaign conducted on Ensemble, a popular group communication system (GCS). 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 due to segmentation violation. Although the actual percentage is small, such errors do constitute an impediment to achieving high dependability, the natural objective of GCSs.

The experiments presented in the paper target a particular GCS (i.e., 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.

Acknowledgments

This work is supported in part by NSF grants CCR 00-86096 ITR and CCR 99-02026.

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 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. 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. 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 European Dependable Computing Conference*, 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] C. Basile, Z. Kalbarczyk, and R. Iyer. Preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. Int'l Conf. on Dependable Systems and Networks*, 2003.

[10] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proc. Int'l Conf. on Dependable Systems and Networks*, pages 135–144, 2002.