2010 Ninth IEEE International Symposium on Network Computing and Applications

A Multi-Core Approach to Providing Fault Tolerance for Non-Deterministic Services

Balazs Gerofi* and Yutaka Ishikawa*†
* *Graduate School of Information Science and Technology*
The University of Tokyo
Tokyo, JAPAN
† *Information Technology Center*
The University of Tokyo
Tokyo, JAPAN
{bgerofi@il.is.s, hfujita@cc, ishikawa@is.s}.u-tokyo.ac.jp

***Abstract*—With the advent of multi- and many-core architectures, new opportunities in fault-tolerant computing have become available. In this paper we propose a novel process replication method that provides transparent failover of non-deterministic TCP services by utilizing spare CPU cores. Our method does not require any changes to the TCP protocol, does not require any changes to the client software, and unlike existing solutions, it does not require any changes to the server applications either. We measure performance overhead on two real-world applications, a multimedia streaming service and an Internet Relay Chat daemon and show that the imposed overhead is minimal as the price of seamless failover. Our prototype implementation consists of a kernel module for Linux 2.6 without any changes to the existing kernel code.**

I. Introduction

TCP is currently the most prevalent transport-layer communication protocol over the Internet, with several high-level protocols and a diverse set of distributed applications built on top of it. However, when a server machine fails, all TCP connections break and the clients get disconnected. Unfortunately, with the increasing number of components in recent computing systems, hardware faults have become common place rather than exceptional [1]. Such outages may cause large impact on an Internet based service, moreover, they force clients that had state associated to the connection to loose their state and redo possibly significant amount of work. Thus, transparent masking of server failures is a major concern.

Replication can be used to increase the availability of network services in the presence of hardware failures. There are two main approaches for software replicating network services. In the *primary-backup* [2] approach, every replica performs the processing of client requests, but only one of them, the primary replies. If the primary machine fails the connection is taken over by one of the replicas. In case of *message logging* [3], only one replica, the primary is processing the client requests and all messages are saved to a stable storage, which is guaranteed to survive server crashes. Should the primary machine fail, one of the replicas replays the log and takes over the connection. Message logging can be combined with checkpointing in order to reduce the size of the log and hence, the time necessary to replay it [3].

What is common in both scenarios is that before sending a reply to the client, the system ensures that the state from which the message is sent will be recovered despite any future failure. This is commonly referred to as the *output commit* problem [4] and the introduced latency in transmitting the reply is called the *output commit stall* [5].

Moreover, in both cases, the execution path of the primary and the backup processes must match, otherwise the backup machine may never reach the state of the primary and will not be able to take over the connection. This implies that the server process is either entirely deterministic on a per connection basis or that all non-deterministic events that a process executes can be identified and the information necessary to replay each event during recovery can be logged. This is called the piecewise deterministic assumption (PWD)[6]. *Leader-follower* models, which are often referred to as a solution for overcoming non-determinism in case of primary-backup replication [7], are also built on top of PWD, because the leader process needs to inform its followers regarding the result of each non-deterministic step during its execution [8].

Various sources of non-determinism may be return values of system calls, such as *select* and *poll*, thread scheduling and signal delivery, order of acquisition and release operations on locks in multithreaded environments, random values that are used in the communication, etc [3].

Previous research has shown several approaches for building fault-tolerant TCP architectures based on replication [7], [9], [10], [11], [12]. Nevertheless, these solutions all rely on the assumption that the server process is entirely deterministic or that the PWD assumption can be easily fulfilled. Despite the availability of solutions for overcoming certain types of non-determinism, such as random numbers [13], a recent paper [5] showed that identifying most sources can be cumbersome. Furthermore, synchronizing such events between the primary and the backup machines often requires modifications to the source code of the server application, which may not be always available [5].

The main motivation behind message logging and primary-backup, even with their difficulties of tackling with non-deterministic execution, is the fact that the output commit stall

978-0-7695-4118-1/10 $26.00 © 2010 IEEE
DOI 10.1109/NCA.2010.42

is short enough to impose neglectable performance overhead to the application.

However, with the current hardware trend toward multi- and many-core architectures, availability of spare CPU cores is becoming widespread. This opens opportunity to utilize a spare CPU core for tracking changes of the server application (including all non-determinism), in a way that overlaps the application's execution and therefore it introduces modest performance degradation to the server process itself.

In this paper we revisit the idea of checkpointing the application before each externally visible event [14], and propose a mechanism, that:

- utilizes a spare CPU core for continuously tracking and synchronizing changes of the server process (between the primary and the backup nodes) in order to minimize the duration of the output commit stall;
- takes a snapshot of the execution context before each externally visible event (ensuring that all non-deterministic steps are reflected at the backup machine);
- and introduces an interposition layer between the application and the operating system that aggregates the application's output when possible to decrease the frequency of the externally visible events.

We make the following contribution: a novel replication method is proposed that is capable of overcoming all sources of non-determinism, while providing transparent failover of arbitrary network services. Our method does not require any changes to the TCP protocol, does not require any changes to the client software, and contrary to existing solutions, it does not require any changes to the server applications either. We measure the overhead on network throughput and provide evaluation on two real-world services: a multimedia streaming server, and an Internet Relay Chat (IRC) daemon. Our experiments suggest that the proposed method imposes acceptable overhead compared to the regular execution.

II. Architecture and Assumptions

In our single IP address cluster each server node is equipped with a public and a local network interface. The same IP address is assigned to the public interfaces and the local ones are used for in-cluster communication. The router simply broadcasts each incoming packet to all server nodes [15].

Contrary to network address translation (NAT) based single IP address clusters [16], there are two important benefits of this configuration. First, the backup machine can capture the same incoming packets that are received by the primary machine. Second, should the failover happen, since both machines have the same public IP address, the backup machine can simply take over the connection without any extra effort on the router.

In this paper we assume that the router does not fail delivering incoming packets to both nodes, although software solutions for ensuring this property have been proposed in the literature [7], [12]. We also assume that the application only reads from files and those files are available on both nodes. Nevertheless, writing files could be handled similarly with network transmissions and it is left for future work.

III. Failure-free Operation of the Primary Machine

A. Initialization

As discussed earlier, there is no need for any modifications to the server application in order to ensure fault tolerant operation. The only requirement is to start it via a special tool that enforces a dynamically loadable library to be attached during execution. The library installs a special signal handler and initializes the write aggregation layer. For more details of the write aggregation layer refer to Section III-B2.

Once the application is running, the replication can be triggered by another tool that establishes a connection to the backup machine and signals the threads of the application. In the signal handler one of the application threads clones a new thread that we call the *tracker* thread. The tracker thread's main responsibility, which will be described in detail in Section III-B1, is to track and transfer state changes of the server application to the backup node asynchronously. In order to ensure that the tracker thread and the application will not compete for the same CPU cores, their CPU affinities are set up to be different. The tracker thread initially transfers the memory mapping descriptors and non-zero pages of the address space.

B. Normal Execution

1) Tracker Thread: The tracker thread remains in kernel space during its whole life cycle and executes a loop in which it incrementally dumps memory changes (i.e. tracks memory geometry changes and dirty pages), and tracks socket data structures that represent the application's network connections. At present, TCP and UDP sockets are supported, where TCP sockets may be in established or listening states.

In between each subsequent loop the tracker thread is suspended for a user defined interval (in our current configuration this interval is 20 milliseconds). Figure 1 demonstrates the activity of primary machine's components including the tracker thread.

2) Write Buffering Layer: The output commit problem postulates that a fault tolerant system must ensure that any state from which a message is sent will be recovered despite a future failure [4]. Since the operating system's TCP stack is deterministic by means of producing the same TCP payload for the same input byte stream submitted to the same TCP state, from the application's point of view, a state from which a message is sent is where a *write*, a *send* or a *sendto* system call is invoked. This implies that a checkpoint of the execution context would have to be taken before any of these calls.

In order to prevent taking context checkpoints before each write call, the write aggregation layer buffers frequent writes and submits them together to the operating system. Write buffering is implemented by hijacking system calls in the dynamically loaded library. Note that the write aggregation layer resides in the application's address space (since it is part of the library), which ensures that all buffered writes will be also reflected at the backup machine.

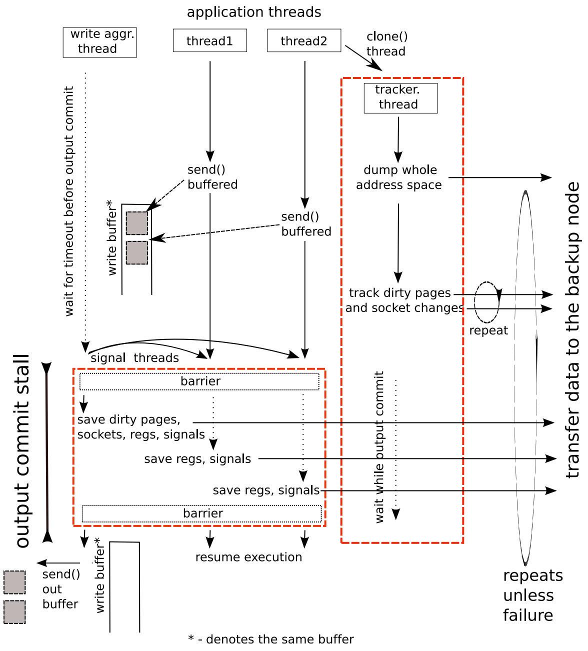


Fig. 1: Failure-free operation of the primary machine (Dashed frames denote kernel space).

The write aggregation layer is implemented with the help of a thread (not to be confused with the tracker thread) that periodically checks if there were any writes buffered, signals the application threads for checkpointing their execution context and submits the write calls to the operating system. The period is a user supplied parameter of the system. We use two values, 50 and 100 milliseconds, depending on the interactivity of the application.

3) Output commit stall: Preceding the invocation of the actual write system calls, the application's execution context has to be synchronized with the backup machine in order to ensure that the state will be recovered despite any future failure.

Taking a snapshot of the execution context is initiated by the write aggregation layer's thread. First it sends a special checkpoint signal to all application threads, including itself. As the result of signaling, they all enter kernel space from the signal handler and a leader thread is chosen, based on setting an atomic variable. The leader thread synchronizes with the tracker thread and dumps any remaining dirty pages. It checkpoints (incrementally) network sockets, dumps thread relations, and finally it saves its registers and signal handlers. The rest of the threads dump only their registers and signal handlers. These steps are necessary in order to ensure consistency at the backup machine.

We anticipate that due to the tracker thread's activity, i.e. keeping the backup machine almost up-to-date with the primary, the output commit stall will take reasonably short time. In Section V we measure the output commit stall latency for real-world applications and show that the introduced overhead is acceptable.

IV. Backup Machine

A. Initialization

Initially, the backup machine creates a new process and clones one more thread than the number of threads the server application contains. Note that the write aggregator thread also counts as part of the application. Once the threads are spawned, they all enter kernel space and a leader thread is chosen, we call this thread the *updater*. All threads, besides the updater, will wait for the failover notification. The updater first re-creates the server application's memory mappings and applies the initial non-zero pages sent by the primary machine.

B. Failure-free Operation

During failure-free operation, the backup machine receives and manages updates from the primary server. It is always prepared to perform an immediate failover. Incremental changes in the address space and the network socket representation are simply stored first. Only when the primary machine dumps execution context (during output commit stall) are the changes applied in order to ensure that the backup machine always maintains the last consistent state of the application's execution.

1) Capturing incoming packets: Besides ensuring that the execution context is carefully synchronized with the backup machine before each output event, another important task is to log incoming traffic between two context snapshots.

Exploiting the router's broadcast configuration, the logging mechanism captures network packets, which match any of the connections that the application maintains. Matching is based on remote IP address, remote port and local port numbers.

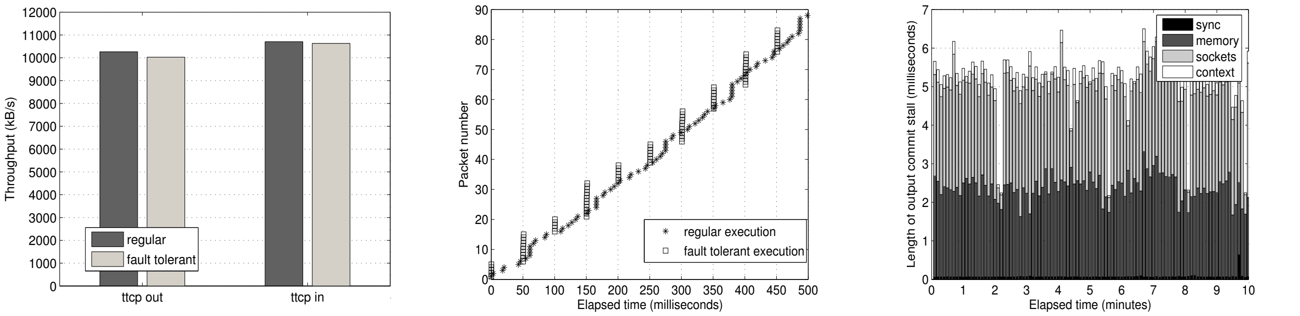
When the backup machine receives an execution context dump, packets that were already acknowledged by the primary machine are dropped. Note that acknowledged packets can be easily identified based on the sequence numbers of the TCP state machine.

C. Failover

The backup machine periodically exchanges heartbeat messages with the primary in order to monitor its health. Should the primary machine go down, the backup recognizes the failure. At this point the application threads, which have been suspended since the initialization of the backup process, get notified and restore the latest execution snapshot received from the primary machine.

Before resuming the application's execution however, packets that were captured after the last context update are injected into the network stack. Once all sockets are ready for communication, each thread restores its registers, signal handlers and resume their execution. Note that along with the application's original threads the write aggregation thread is also resumed. It flushes any pending writes from the aggregation buffer and finally, it disables checkpointing.

It is also worth noting that the recovered process might retransmit several packets that were already transmitted by the primary machine before it failed. The backup machine has no knowledge on when exactly the failure occurred (before or



(a) Network throughput, ttcp.

(b) Packet level effect of the write buffering, Live555 (25 clients).

(c) Output commit stall, Live555 (25 clients, 50ms output commit period).

Fig. 2: Evaluation on network throughput and the Live555 streaming server.

after replying the client), which is a fundamental limitation of any fault-tolerant system [5]. However, this is not a problem in the case of TCP, because the protocol itself handles retransmissions naturally.

V. Experimental Evaluation

We evaluate our system's performance on a broadcast based Single IP Address cluster. Both the primary and the backup machines are equipped with a 2.4GHz Dual-Core AMD Opteron processor and two gigabytes of RAM. The nodes are interconnected with Gigabit Ethernet network for in-cluster communication and each machine has a 100Mbps Ethernet public interface to which the router broadcasts incoming packets.

A. Network Throughput

For studying the effect of our replication mechanism on network throughput we used the *ttcp* utility [17], a simple bandwidth testing tool available online. All data in ttcp are generated by the sender and discarded by the receiver.

Figure 2a shows the results we obtained from running ttcp under regular and fault tolerant operation. Besides showing the obtained transmission rates, we grouped the measurements for *in* and *out* traffic, where the direction is meant from the server's point of view.

Compared to its clean operation ttcp achieves 99% of the incoming and 97% of the outgoing network throughput in the case when replication is enabled. We explain the higher overhead for outgoing traffic with the buffering behavior of the write aggregation layer, which dirties extra pages that need to be reflected at the backup machine. However in both cases, ttcp spends most of its execution time on waiting for results of the actual write and read calls, leaving enough space for occasional context checkpoints.

B. Live555 Media Server

The Live555 Media Server [18] is an open-source media server application that supports on-demand streaming of various file formats over the Real-Time Streaming Protocol (RTSP). The default setup for streaming a video file utilizes a TCP connection for obtaining the media's meta-file and for exchanging streaming commands, while the actual content is transmitted over UDP.

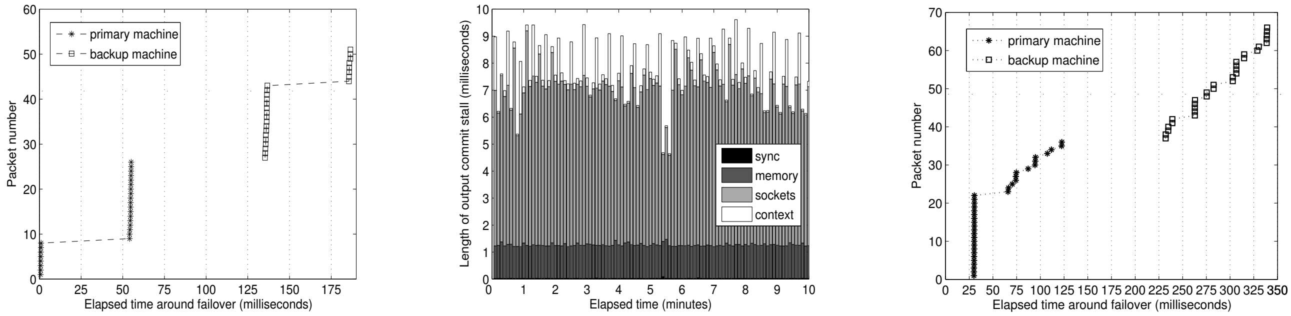
We apply our fault tolerant mechanism to the Live555 streaming server without any modifications to its source code and measure several aspects of the execution. In each experiment we use a 352x240 MPEG4 video file for streaming.

The *tcpdump* tool has been used for logging packets transmitted by the server in order to assess the effect of the write aggregation. We use 50 milliseconds as the output commit period and 25 clients are involved in the experiment. Figure 2b shows the distribution of packets for a half second time interval with and without fault tolerance enabled. Note that by looking at the video, the fault tolerant execution has no distinguishable effect compared to regular execution. The key observation in Figure 2b is the 50 milliseconds grouping of outgoing packets, which is the result of the write aggregation layer forcing one output commit stall in every 50 ms. Figure 2c depicts the length of the output commit stall periods. Samples from a 10 minutes long time interval are provided, 10 values for each minute. A single column details the distribution of the major activities during context checkpoint, which are: synchronization with the tracker thread, dirty page tracking, incrementally dumping network sockets and saving registers and signal handlers.

TABLE I: Worst case (longest) output commit stall, Live555.

Action	Proportion	Length
Synchronizing with tracker thread	1.2%	0.08 ms
Dumping dirty pages	46.2%	3.14 ms
Dumping socket changes	48.3%	3.28 ms
Dumping execution context	4.2%	0.28 ms
Total		6.8 ms

The worst case (longest) output commit stall is given in Table I. Although the output commit stall lasts for almost 7



(a) Failover time, Live555 (25 clients).

(b) Output commit stall, ircd (1024 clients, 100ms output commit period).

(c) Failover time, ircd (1024 clients).

Fig. 3: Failover time of Live555, output commit stall and failover time of ircd.

milliseconds in the worst case, the average length remains less than 5 milliseconds. Since it is executed once approximately in every 50 milliseconds, effectively the application spends less than 10% of its execution time on context checkpointing.

As for the duration of the actual failover Figure 3a depicts the time interval elapsed between the last packet of the primary and the first packet of the backup machine. As it demonstrates the backup recovers in less than 100 milliseconds, rendering the fault completely transparent for the clients.

C. Internet Relay Chat daemon

Our second application is the de-facto Internet Relay Chat (IRC) daemon, ircd [19]. IRC is a real-time Internet text messaging protocol which is mainly used for group communication and is the basis of several other protocols, including Microsoft's popular MSN Messenger network. Clients of the IRC network associate state to their connection, such as privileges on channels. When the server machine fails, clients are disconnected and their state is lost.

TABLE II: Worst case (longest) output commit stall, ircd.

Action	Proportion	Length
Synchronizing with tracker thread	1.2%	0.11 ms
Dumping dirty pages	27.3%	2.64 ms
Dumping socket changes	62.3%	6.04 ms
Dumping execution context	9%	0.87 ms
Total		9.7 ms

We evaluate our replication mechanism on ircd without changing its source code. The output commit period is set to 100 milliseconds and there are 1024 clients involved. Most of the clients are IRC robots that resemble human communication on multiple channels, which in turn we observe with a regular IRC client. We run the same set of experiments in order to assess the length of the output commit stall, the number of dirty pages transferred by the tracker thread as well as during the output commit and the failover time based on logs collected by tcpdump.

The duration of output commit stalls is depicted in Figure 3b, sampled the same way it was for Live555. Table II reveals the worst case scenario, in which the output commit stall takes 9.7 milliseconds. However, in average it lasts for less than 8 milliseconds, which means that the application spends around 8% of its execution time on context checkpointing, because there is one output commit scheduled approximately in every 100 milliseconds.

The failover time is shown in Figure 3c. The time difference between the last packet from the primary machine and the first from the backup is approximately 100 milliseconds, which renders the recovery fully transparent for the clients.

VI. Related Work

Several projects have explored the idea of TCP level recovery, where the failure remains hidden from the clients. They can be categorized into two main groups, assuming completely deterministic applications or building upon the piecewise deterministic assumption.

Fetzer et al. proposed a library extension that replicates a server application in a semi-active manner [9]. When the primary server fails the client opens a new connection to the backup server and continues communication. Besides that the solution needs client side library modification, nondeterminism is simply assumed to be solved by a leader-follower protocol, which in turn requires the PWD assumption to be satisfied.

Zagorodnov et al. describes a system, FT-TCP, in which all client-server TCP communication as well as socket read calls are logged and transferred to a backup machine [10]. In case of a failure the backup machine replays all network activity and it takes over the role of the server. Extensions of this work [11], [5] investigate exploiting PWD by means of wrapping the TCP stack and logging not only packets and socket read calls, but also other system calls that may form sources of non-determinism. The improved FT-TCP comes in two flavors, hot and cold backups. In case of cold backup when a failure occurs the backup machine first replays the steps and than takes over the communication. Hot backup performs the same steps on both machines. It is reportedly necessary to modify most applications in order to identify and synchronize all non-determinism [5].

Systems, where the TCP communication at the network layer is replicated to a secondary server have been also proposed. ST-TCP [7] provides a backup server which taps the TCP traffic at the Ethernet level and executes the exact same steps on the backup machine with the ones on the primary. Replies from the backup server are simply dropped during failure-free execution. Reception of the same data is synchronized through an additional communication channel between the primary and backup machines. Melliar-Smith et al. proposes a similar approach with having the backup machine's network interface in promiscuous mode [12]. Data reception is synchronized between the machines and replies are compared. In both cases mentioned above, the server processes are assumed to be completely deterministic.

The idea of checkpointing each externally visible event has been explored for recovering general-purpose applications, built on reliable main memory and high-speed transactions [14]. This work however cannot be applied directly for hardware failures.

With recent advances in virtualization technologies, transparent failover at the VM level has been also proposed [20]. While it succeeds in dealing with non-determinism entirely, the imposed overhead is much bigger than at the process level, due to the large amount of state that needs to be synchronized between the primary and the backup machines.

VII. Conclusion and Future Work

We proposed and evaluated a novel replication method that provides fault-tolerant execution for non-deterministic Internet services without any modifications to their source code. Our solution exploits the availability of spare CPU cores, an expected benefit of the upcoming multi- and many-core hardware environments.

The proposed mechanism utilizes a separate CPU core for transferring changes of the application asynchronously to a backup node without affecting the performance of the server process itself. It ensures that before generating any outside visible event the execution context is carefully checkpointed so that recovery is always possible when a hardware failure occurs on the primary machine.

Evaluated on two real-world applications, we showed that the output commit stall can be reduced as far as 8% of the overall execution, rendering the failure-free operation statistically indistinguishable from the the applications' original execution. Furthermore, we presented measurements to demonstrate that recovery is on the 100 milliseconds scale.

In the future we intend to investigate replicating process groups, such as a web-server and its CGI scripts. We also intend to evaluate our mechanism on applications which exhibit higher interactivity, such as multiplayer online games.

Acknowledgment

This work has been supported by the CREST project of the Japan Science and Technology Agency (JST).

References

[1] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Primary-Backup Protocols: Lower Bounds and Optimal Implementations," in *In Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, 1992, pp. 187–198.

[3] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[4] R. E. Strom, S. A. Yemini, and D. F. Bacon, "A recoverable object store," in *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 215–221.

[5] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Practical and low-overhead masking of failures of TCP-based servers," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 1–39, 2009.

[6] R. Strom, D. Bacon, and S. Yemini, "Volatile logging in n-fault-tolerant distributed systems," Jun 1988, pp. 44 –49.

[7] M. Marwah, S. Mishra, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," *Dependable Systems and Networks, International Conference on*, vol. 0, p. 373, 2003.

[8] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs, "The Delta-4 extra performance architecture (XPA)," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, Jun 1990, pp. 481–488.

[9] M. Orgiyan and C. Fetzer, "Tapping TCP Streams," in *NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 278–289.

[10] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping server-side TCP to mask connection failures," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, 2001, pp. 329–337.

[11] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud, "Engineering fault-tolerant TCP/IP servers using FT-TCP," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, June 2003, pp. 393–402.

[12] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith, "Transparent TCP Connection Failover," *Dependable Systems and Networks, International Conference on*, pp. 383–392, 2003.

[13] D. Zagorodnov and K. Marzullo, "Managing self-inflicted nondeterminism," *1st Workshop on Hot Topics in System Dependability (HotDep)*, pp. 323–328, 2005.

[14] D. E. Lowell and P. M. Chen, "Discount checking: Transparent, low-overhead recovery for general applications," University of Michigan, Tech. Rep. CSE-TR-410-99, 1998.

[15] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang, "ONE-IP: techniques for hosting a service on a cluster of machines," in *Selected papers from the sixth international conference on World Wide Web*. Essex, UK: Elsevier Science Publishers, Ltd., 1997, pp. 1019–1027.

[16] W. Zhang, "Linux Virtual Servers for Scalable Network Services," *Linux Symposium*, 2000.

[17] Test TCP (TTCP): Benchmarking Tool and Simple Network Traffic Generator, http://www.pcausa.com/Utilities/pcattcp.htm, 2010.

[18] Live555.com Internet Streaming Media, http://www.live555.com, 2010.

[19] IRC DAEMON: IRC Server Software, http://irchelp.org/irchelp/ircd, 2010.

[20] Brendan Cully and Geoffrey Lefebvre and Dutch Meyer and Mike Feeley and Norm Hutchinson and Andrew Warfield, "Remus: High availability via asynchronous virtual machine replication," in *In Proc. NSDI*, 2008.