SYSTEM SUPPORT FOR SERVICE AVAILABILITY, REMOTE HEALING AND FAULT TOLERANCE USING LAZY STATE PROPAGATION

BY FLORIN SULTAN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Liviu Iftode

and approved by

New Brunswick, New Jersey

October, 2004

ABSTRACT OF THE DISSERTATION

System Support for Service Availability, Remote Healing and Fault Tolerance Using Lazy State Propagation

by Florin Sultan

Dissertation Director: Liviu Iftode

Our thesis is that lazy state propagation can be successfully used to implement efficient support for service availability, remote healing and fault tolerance.

The end-to-end availability of an Internet service is currently constrained by the static client-server binding imposed by the TCP/IP protocol. To overcome this problem, we propose *lazy* migration of live client service sessions between equivalent servers. We have designed and implemented Service Continuations, an OS mechanism for session state migration between multi-process servers, along with Migratory TCP, a connection migration protocol that enables lazy session migration, and present experimental results with real Internet servers that validate the approach.

Failure or damage to the state of the OS can lead to loss of critical application and OS state residing in system memory. As a solution to this problem, we propose remote healing through *lazy* recovery/repair actions on the in-memory software state of a computer system. To enable remote healing, we have designed and implemented Backdoors, a novel system architecture based on remote memory communication that allows access to resources of a machine even after an OS failure renders it unavailable. We present experimental results showing the Backdoors achieves efficient monitoring and fast recovery and repair.

Distributed shared memory (DSM) systems used to run parallel applications on large commodity clusters are sensitive to individual node failures that compromise the whole computation. We have designed and implemented an efficient fault-tolerant DSM system for which we have developed two *lazy* algorithms for garbage collection of recovery state. We demonstrate through experiments with benchmark applications that our recovery support is light-weight and that lazy garbage collection effectively limits the amount of recovery state retained in the system.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Liviu Iftode, whose guidance, support and encouragement made it possible to bring this research work to its logical conclusion. I thank the members of my dissertation committee, Professor Badri Nath, Professor Ricardo Bianchini and Dr. Fred Douglis, for their very helpful comments on the contents of this thesis. I thank Professor Thu Nguyen, who provided guidance during my work on fault-tolerant DSM, and useful feedback on other projects.

I would like to thank all my colleagues from Rutgers, Maryland and INRIA/IRISA who, at one point or another, have worked on the projects that comprise this thesis, helping me mold ideas into real, tangible, working systems. Without them, this thesis, of which I am only the imperfect writer, would not exist. They contributed design ideas, wrote code, ran thousands and thousands of experiments. Their invaluable input in innumerable discussions when I was wrong and they were right was decisive in shaping ideas and validating them. Their unbending high spirits sustained the writer and helped him survive through and recover from countless kernel crashes, often when everything seemed without hope. They are: Kiran Srinivasan, Deepa Yier, Aniruddha Bohra, Pascal Gallard, Iulian Neamtiu, Stephen Smaldone and Yufei Pan. To all of them, I am deeply grateful - and you made a great team to work with!

I thank all other members of the Disco Lab at Rutgers, especially to Murali Rangarajan, Zoltan Jarai, Cristian Borcea, Suresh Gopalakrishnan and Vivek Pathak, for their help and for being so close to me during my years at Rutgers.

Finally, I thank my mother and my brother for believing in me. To Georgiana, for her love through years made mostly of days and nights spent in a lab, away from her. To my friends in Romania and elsewhere, for just being - across distances and time. To all of you: your warm hearts have been always with me, and kept my soul alive.

This work has been supported in part by the National Science Foundation under grants CAREER CCR-0133366 and ITR ANI-0121416.

Dedication

For my parents, Sanda and Ion Sultan.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SC	Service Continuations
M-TCP	Migratory Transmission Control Protocol
BD	Backdoors
I-NIC	Intelligent Network Interface Contoller
SB	Sensor Box
CB	Continuation Box
DSM	Distributed Shared Memory
HLRC	Home-based Lazy Release Consistency
CGC	Checkpoint Garbage Collection
LLT	Lazy Log Trimming

Chapter 1

Introduction

1.1 Thesis

The thesis of this dissertation is that lazy state propagation can be successfully used to implement efficient support for service availability, remote healing and fault tolerance.

1.2 Lazy versus Eager State Propagation in Distributed Systems

Distributed systems are collections of nodes separated by a network that cooperate and communicate towards performing a common task. In cases where the task involves maintaining consistent replica of the state of a node which is changing over time, the node has to propagate information describing the changes in its state. One obvious approach is *eager propagation*, which means that information is transferred from the source node as soon as a change in its state takes place. This minimizes the interval during which replica are inconsistent but consumes resources and may negatively impact performance. In addition, eager propagation misses on opportunities to perform optimizations like buffering or coalescing state updates. The alternative is *lazy propagation*, which delays the transfer until "the last moment" that is required to maintain correct semantics of the distributed/replicated state, depending on the particular system.

We further exemplify the concept of lazy versus eager state propagation in two areas that are the focus of this dissertation: availability and fault tolerance.

Full replication of the state of a computation has long been used to provide tolerance to failures or improve the availability of a service. The most prevalent approaches to state replication in distributed systems are primary-backup [16] and state-machine replication [77], two paradigms originally developed for building fault-tolerant and highly available services. In the state-machine replication approach, the state of a service is replicated at several nodes and operations that access (and possibly modify) it are delivered in the same total order, thus keeping the replica consistent. In the primary-backup approach, one node designated as primary executes operations that alter its state, then updates the state of one or more backup nodes.

Practical implementations of fault-tolerant operating systems use the primary-backup approach to maintain hot backups of the state of a process (e.g., in Tandem [9] and TARGON 32 [14]) or of a virtual machine (e.g, in Hypervisor-based fault-tolerance [15]). In case of a failure of the primary, a backup with similar state takes over. In all cases, replication is performed eagerly and requires spare machines or processors dedicated to mirroring the state of one system or process of interest. The focus of research in these systems was on how to keep the replica fully synchronized down to the interrupt level. For example, in the TARGON 32 system, asynchronous events (UNIX signals) are transformed into synchronous messages delivered to the destination process and its backup. In Hypervisor, the virtual machine monitor tracks and replicates the state of a virtual machine at the granularity of epochs delimited by hardware interrupts. As a result, because they eagerly propagate synchronization information to a replica at a very low granularity of events and rely on hardware redundancy, systems that achieve full-state replication are costly to implement and/or exact high performance penalties.

At a smaller scale, primary-backup schemes have been used to replicate only part of a system, e.g., to build fault-tolerant TCP servers by mirroring the communication and computation state on another machine through active remote logging of TCP segments [101] or passive network traffic tapping at the link-layer [60, 51]. These schemes suffer from the same core limitations: they require fully-dedicated nodes as backups and interpose on the critical path, affecting the performance of failure-free execution.

The widely-used alternative to full-state replication in building fault-tolerant systems is rollback recovery, which periodically checkpoints the state of a computation and uses a checkpoint to restart it. Checkpoints commonly include only the user-level state of a process and are saved on stable storage (usually disk) which is orders of magnitudes slower than the main memory. Because of the high overhead of checkpointing (large amounts of unstructured state are saved to a slow device), Zhou et al. have proposed efficient mirroring of the address space of a process in the memory of other nodes in a computer cluster using virtual memory-mapped communication [103]. This system also uses eager propagation of recovery state, but takes advantage of specialized hardware to automatically perform mirroring while reducing the interference with application execution. However, since it manages only user-level state and does not replicate the communication state maintained by the OS, it does not support recovery of communicating processes.

The common feature of all the above and similar approaches is that they proactively and continuously propagate state from the source machine to other machines. This eager propagation of the state of a system or subsystem has two important advantages: *(i)* it creates an exact replica of the state at a given moment in time, always available if the original system fails or becomes unavailable, and *(ii)* it can guarantee perfect recovery after a failure since the replica is self-contained. However, these advantages come at the expense of added communication traffic during normal execution of the system, costly dedicated nodes and decreased performance.

In this dissertation, we propose solutions for service availability, remote healing and fault tolerance that rely only on *lazy propagation* or *extraction* of computation and communication state from a computer system. Laziness implies that we will wait until the last moment, when action for moving or extracting the state from the computer of interest becomes absolutely necessary for maintaining availability or to survive a failure. Lazy propagation/extraction of state has the advantage that it does not consume resources of other nodes until the moment they are actually needed and that it does not impact the performance of the source node during normal execution.

Using lazy state propagation raises several important research issues: *(i)* the transferred state must be light-weight: since the transfer is deferred until when necessary, it usually lies on a critical path for the performance or responsiveness of the system, therefore it must be efficiently implemented; *(ii)* the state transfer may need to take place after a failure, crash, DoS attack, etc. has crippled or made unavailable the OS of the machine that holds the state; *(iii)* lazily propagated state that is used in resource management decisions, e.g., in garbage collection of recovery state in a fault-tolerant system, may lead to suboptimal resource usage in the system, so careful designs are needed in such cases.

1.3 End-to-End Internet Service Availability

The vast majority of today's Internet services are built over TCP, the standard reliable transport protocol of the Internet. The connection-oriented nature of TCP, along with its endpoint naming scheme based on network layer (IP) addresses, creates an implicit *binding* between a service and the IP address of a server providing it, throughout the lifetime of a client connection. This makes the client's session with the service sensitive to all adverse conditions that may affect the server endpoint or the internetwork *after* the connection is established: congestion or failure in the network, server overload, failure or DoS attack.

Studies that quantify the effects of network stability and route availability [54, 28] demonstrate that they can significantly reduce the end-to-end availability of Internet services. Dahlin et al. [28] have found that network failures lead to disconnections between a typical client and a typical server on the order of 15 minutes per day. One of their key findings is that the distribution of unavailability intervals is heavy-tailed, which means network disconnections are of long duration, and thus are likely to affect ongoing client sessions. Labovitz and Ahuja [54] show that 40% of the inter-domain BGP routing failures take 10 minutes to repair, and 60% are resolved in 30 minutes. All these result in decreased end-to-end service availability. As Dahlin et al. [28] point out, "providing highly available servers is not sufficient for providing a highly-available service because it is not and end-to-end approach: other types of failures can prevent users from accessing services."

This problem is even more obvious if we consider that in virtually all instances of Internet services, there exist identical servers that can provide the same service. This redundancy is typically used by service providers *(i)* in cluster-based Internet services, to increase the availability of the service in face of individual server failures, and *(ii)* over the wide area, to distribute servers across geographical regions for better response latency. However, the increased *server availability* does not improve the *end-to-end service availability* in case of connectivity failures on the client-server path or overload on the server once the client session has started. The static binding of the client to one server, established at the start of the service session, prohibits existing redundant servers from being used to sustain continuous service to the client. In this dissertation, we present Service Continuations, a solution to this problem based on OS and protocol support for lazy and dynamic migration of a service session between equivalent servers to cope with availability problems after the start of the session.

1.4 Self-Healing Systems

Operating system hangs, crashes, deadlocks or panics are system failures from which the only option for recovery is a reboot. A reboot regains control over the machine but discards all live application and OS state still present in system memory, while this state might be critical to users or clients of the system.

Traditionally, two approaches have been used to preserve the state of a system across failures: process or virtual machine checkpointing (in which the state of a process/VM is periodically saved to stable storage) and hot backups (in which the whole state of a system is mirrored on a dedicated machine, e.g., using a primary-backup replication scheme). These approaches have the advantage that they exploit well-understood and easy to manipulate units of state encapsulation, i.e., a process or a (virtual) machine. However, this also makes them extremely heavy-weight since it involves handling large amounts of state, either by saving it to stable storage or mirroring it on a dedicated machine. In addition, state mirroring on a hot backup is intrusive and costly as it must interpose on all system interactions and requires extra hardware, while checkpointing requires stable storage which is still available after a failure.

Common to both checkpointing and hot backup approaches is that the current in-memory state of the system, even if still available, is discarded in the event of a failure. With checkpointing, the system rollbacks execution to a previous (checkpointed) state, which destroys its current state. With hot backups, the backup machine takes over the functionality of the primary and its state is simply ignored.

In many cases, however, a failed system is frozen, crashed, panic'ed or under a DoS attack that makes it unresponsive, while its memory still holds "good" application and OS state, e.g., data in TCP buffers of live connections in a network server, data in dirty buffer cache blocks not synced to a file server, application-specific data describing the point an application has reached in its computation, etc. In such cases, in contrast to approaches like checkpointing and replication, it would be appealing to reuse (rather than discard) this fine-grained state still present in the memory of the failed system, *after* the failure has occurred. This would enable fast recovery (similar to hot backup replication, but with no dedicated machine) and building self-healing support into the system with low failure-free overhead. The main obstacle against such an approach is that access to the useful state of a system is mediated by its operating system and processors, which may no longer be available or responsive. In this dissertation, we propose Backdoors, a system architecture and OS support that enable automated healing operations to be lazily performed from another system, even after a failure of the OS of the target system.

1.5 Fault-Tolerant Distributed Shared Memory

Clusters of commodity machines achieve scaling of parallelizable applications in two dimensions: time (execution time) and space (problem size). In the time dimension, for a fixed problem size, a cluster provides increased computing power to speed-up application execution compared to a single node. In the space dimension, the aggregate memory of nodes in the cluster can accommodates larger problem sizes.

Distributed shared memory (DSM) has emerged as a simple abstraction for programming on clusters that conveniently covers both of the above dimensions. The idea behind DSM is to extend a simple programming abstraction provided by the OS, the virtual memory, across machine boundaries in a cluster. A DSM system uses virtual memory mechanisms along with message passing to provide parallel applications with the view of a coherent virtual address space shared across nodes.

In DSM systems used to run long-running applications on large commodity clusters, tolerance to individual node failures becomes critical since a single node failure can compromise the whole computation. At the same time, any system support introduced for fault tolerance must be scalable and must not impact failure-free execution. Large DSM clusters practically prohibit the use of costly synchronization to achieve a consistent recoverable state in case of individual node failures. For these reasons, fault-tolerant DSM approaches based on coordinated checkpointing like [21, 44, 50, 17, 27], in spite of being easy to implement, incur high performance penalties in the more common case of failure-free execution.

The alternative is to use independent checkpointing and maintain logs of communication events that affect the DSM state. However, existing systems that adopt this approach [73, 93, 27, 64, 53, 52] differ widely in the memory consistency models used, the frequency of logging and/or checkpointing needed to maintain the recovery state, the logging support used (disk or volatile memory), etc. They either *(i)* use memory consistency models and coherency protocols that do not allow efficient implementations in a DSM system, which affects their base performance, or *(ii)* use expensive techniques, often interposed on the critical communication/execution path, to manage the recovery state during execution. From the memory model viewpoint, [73] is based on sequential consistency, an inefficient model for DSM, and uses logging of all memory accesses, while [64] uses an entry-consistent protocol limited to a single-writer, object-based DSM system. From the performance viewpoint, even when they use efficient consistency models, these systems require global coordination for their operation [27], perform frequent log flushing to stable storage [73, 52, 93], or use expensive (global) checkpointing as surrogate for garbage collection operations [93, 27] when not ignoring them altogether [52]. In this dissertation, we describe a scalable fault-tolerant DSM based on independent checkpointing using lazy garbage collection of recovery state.

1.6 Dissertation Contributions

In summary, the main contributions of this dissertation are:

- We design and implement Service Continuations (SC), a novel OS mechanism for increased end-to-end Internet service availability through lazy session migration. We design and implement Migratory TCP (M-TCP), a connection migration protocol with lazy state transfer based on the SC model.

  SC was presented in a paper published at the *22nd Symposium on Reliable Distributed Systems (SRDS 2003)*. M-TCP was described in a position summary at *HotOS-VIII 2001*, a short paper published in the *22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, and a detailed description of its implementation was published as *Department of Computer Science Technical Report DCS-TR-459*.

- We propose the concept of remote healing, in which actions to restore the functionality of a system are performed lazily from another healthy system, after a failure or damage to the state of the target system has occurred. We design and implement Backdoors (BD), a novel system architecture for remote healing based on remote memory communication. We design and implement OS support for monitoring the state of a computer system, recovery of in-memory state, and repair of damaged OS state.

  The Backdoors architecture was first described in a paper at the *1st Workshop on Algorithms and Architectures for Self-Managing Systems (2003)*. Remote repair with Backdoors was presented in a paper published in the *International Conference on Autonomic Computing (ICAC 2004)*. Recovery from OS failures using Backdoors is described in the *Department of Computer Science Technical Report DCS-TR-543* and in a paper currently under review by the *Internet Computing* journal.

- We design and implement system support for efficient fault tolerance in DSM clusters based only on independent checkpointing and volatile memory logging.

In our system, garbage collection of recovery state relies on lazy propagation of control information and operates lazily with respect to checkpointing and logging operations. We develop, prove and evaluate two novel algorithms for lazy garbage collection of logs and checkpoints.

This work was presented in a paper published in the *Supercomputing Conference 2000*, and in a paper published in *IEEE Transactions on Parallel and Distributed Systems, October 2002.*

1.7 Contributors to Dissertation

The following is a list of all my colleagues who co-authored papers from which I used material in this dissertation, along with their contributions: Kiran Srinivasan and Deepa Yier have contributed to the M-TCP implementation and applications. Aniruddha Bohra has worked on the SC implementation and applications, on support for remote recovery in Backdoors, and applications of Backdoors in remote recovery and repair. Pascal Gallard (INRIA/IRISA) has contributed to the implementation of remote memory access support in Backdoors. Iulian Neamtiu (University of Maryland) has contributed to the implementation of monitoring and remote repair support in Backdoors. Stephen Smaldone has implemented real applications using SC and Backdoors. Yufei Pan has contributed to the evaluation of the Backdoors architecture.

1.8 Outline of Dissertation

This dissertation is organized as follows. Chapter 2 describes Service Continuations, an OS mechanism for lazy dynamic migration of Internet service sessions, along with Migratory TCP, a connection migration protocol. Chapter 3 describes the remote healing approach and its enabling Backdoors system architecture. Chapter 4 describes a fault-tolerant DSM system based on independent checkpointing and lazy garbage collection. Finally, Chapter 5 concludes the dissertation.

Chapter 2

Service Continuations

2.1 Problem Statement

The growth of the Internet has led to increased demands by its users with respect to both availability and quality of the services delivered over an internetwork where best-effort service is the norm. Critical applications, as well as applications requiring long-term connectivity run over the Internet. In addition, increased user expectations conflict with increasing load on popular servers that become overloaded, fail, or may fall under DoS attacks. From the clients' perspective, all these result in poor *end-to-end* service availability.

A vast majority of today's Internet services are built over TCP [69], the standard reliable transport protocol of the Internet. The connection-oriented nature of TCP, along with its endpoint naming scheme based on network layer (IP) addresses, creates an implicit *binding* between a service and the IP address of a server providing it, throughout the lifetime of a client connection. This makes the client prone to all adverse conditions that may affect the server endpoint or the internetwork after the connection is established: congestion or failure in the network, server overload, failure or DoS attack. Although highly available *servers* can be deployed, providing highly available *services* remains a problem due to connectivity failures. With TCP/IP, availability of a service is constrained not only by the availability of a given server, but also by that of the routing path(s) to the server. Studies that quantify the effects of network stability and route availability [54, 28] demonstrate that they can significantly reduce the end-to-end availability of Internet services.

This problem is even more obvious if we note that in all practical instances of Internet services there always exist identical servers that provide the same service. This redundancy is typically used by service providers for two reasons. First, in cluster-based Internet services, to increase the availability of the service in face of individual server failures. Second, over the wide area, to distribute servers (or cluster of servers) across geographical regions for better response latency. However, despite of the existence of alternate servers, these cannot help improve the end-to-end service availability in case of connectivity failures on the client-server path or overload on the server. The static binding of the client to one server, established at the start of the service session, prohibits these servers from being used to sustain continuous service to the client.

2.2 Background and Related Work

Our Service Continuations (SC) are closely related to work in process migration, fault-tolerant operating systems, and in providing high availability for Internet services through protocol support.

2.2.1 Process Migration

Process migration [59, 97, 8, 29] is a heavy-weight, generic mechanism that enables seamless execution of an application process at multiple nodes during its lifetime, targeting load sharing and balancing in clusters. SC differs from classical process migration in that it trades off transparency for finer migration granularity, by application-level control of migrated state. While the goal of process migration is to transparently move and restore *whole execution contexts*, SC requires the application to cooperate in defining fine-grained specific state distributed across multiple processes to be moved and restored in *different execution contexts*.

SC addresses the problem of transparently restoring the state of open communication channels of an application when the execution site changes. Unlike most process migration schemes, SC does not freeze execution, and does not require inter-process or client-server synchronization. This comes at the expense of potential re-execution after a migration. Most existing systems supporting process migration are custom operating systems designed from scratch with built-in migration support [97, 8, 29]. In contrast, we provide a fairly straightforward implementation of SC in a general-purpose OS.

2.2.2 Fault-Tolerant Operating Systems

The idea of using logs at the OS level for deterministic execution replay can be traced back to early fault-tolerant operating systems like NonStop [9] and Auros [13, 14]. In the NonStop kernel, a message logging scheme was used to provide single-failure fault tolerance with primary-backup process pairs. This was probably the first system to use logging on inter-process communication channels to ensure deterministic replay at the OS level during fault recovery. SC uses a similar technique to synchronize the application and communication state, thereby ensuring deterministic behavior after a migration.

The SC synchronization scheme is similar to log-based rollback recovery techniques used in fault-tolerant distributed systems [33]. Unlike these, an SC does not use full checkpoints of process state, and limits the rollback and replay to restore only a small part of process state, specific to exactly one client.

2.2.3 Protocol Support for Highly-Available Internet Services

Providing high availability for Internet services through protocol support has been approached in several ways: using connection migration in application-specific solutions [80, 100], fault tolerance for TCP [2], and new transport protocols [85]. None of these approaches provides the OS support required for migrating and resuming complex, multi-process service instances.

Snoeren at al. describe a scheme that enables HTTP connection endpoints to migrate within a pool of support servers [80]. Migration is supported by broadcasting per-connection HTTP and TCP state within the server pool. The scheme adds an HTTP aware module at the transport layer that extracts information from the application data stream to be used for connection resumption. While this achieves transparency, it creates a strong dependency on the content of the application data stream and on HTTP specifics. Connection migration is limited to HTTP, for which it can only migrate static transfers. In contrast, our SC abstraction provides support for fine-grained connection migration, through a generic mechanism that can be used with any application, and which is not limited to single-process servers. With SC, a server application must change to assist migration. However, no knowledge of application specifics is required at the OS/protocol level for resuming the service after migration.

Luo and Yang describe a technique for fault-resilience in a cluster-based HTTP server using a front-end dispatcher to monitor client connections serviced by back-end nodes [57]. In case of failure, the dispatcher restores the serviced connections on another node using a scheme similar to [80]. The dispatcher is involved in correct CGI execution by caching dynamic content. The scheme is application-specific, limited to clusters, and makes the dispatcher a single point of failure and a potential bottleneck. In contrast, SC is a generic solution, does not use centralized control, can support multiple processes and works over wide area.

Alvisi et al. propose Fault-tolerant TCP (FT-TCP), a system for masked recovery of a crashed server process with open TCP connections [2]. A wrapper around the TCP layer intercepts and logs reads by the process for replay during recovery, and shields the remote client endpoints from the failure. The scheme uses full process context checkpoints to recover from a crash and works only for single process servers. Compared to FT-TCP, SC allows dynamic and fine-grained connection migration of client sessions with state distributed over multiple communicating processes.

Aron et al. use TCP connection handoff for load balancing in clustered HTTP servers by distributing incoming client requests from a front-end host to back-end server nodes [4]. Migration is limited to the initial connect. Multiple handoffs of persistent HTTP/1.1 connections at request granularity are mentioned, but no design or implementation are described. In contrast, SC allows dynamic migration between multi-process servers that can be distributed across a WAN. SC can be used in load-balancing schemes where load may be monitored at a finer granularity than a fixed, application-specific unit. For HTTP servers, SC supports migration with dynamic content execution and persistent connections.

2.3 Fine-Grained Session Migration

2.3.1 The Service Continuation Idea

To address the problem of service continuity, we propose to exploit the server redundancy *after* the session has started by lazily and dynamically migrating live client sessions between servers in the Internet. Migration of a session is *lazy* in that state pertaining to the session is not transferred or otherwise replicated on other nodes until the moment of migration. Migration of a session is *dynamic* in the sense that it may occur multiple times, at any point during session lifetime, transparently to the client and asynchronously with respect to server execution. To achieve these goals, we propose *service continuations* (SC), a novel OS mechanism to support dynamic migration of live client sessions between multi-process cooperative servers. The SC concept is rooted in the (most often valid) assumption that a service maintains well-defined *fine-grained* state for a client session and that client sessions are independent of each other.

To the server application, an SC represents an abstraction of *discrete* application-level state associated with a client session, spanning multiple process contexts, which is guaranteed to be restored at a new server upon migration. At the OS level, an SC is an ordered sequence of fine-grained state components associated with processes involved in servicing the client.

For each server process, and for a given client, an SC stores client session state and OS state of communication channels. The first component in the SC sequence corresponds to the front-end process that accepts the client connection for service. Subsequent components correspond to other back-end processes (if any) that participate in servicing the client. To resume the service at a new server, the SC is migrated and used to reinstate the service state and the communication state for all processes involved in the service. We emphasize that migrating an SC does not involve whole process contexts, but only "small" state components associated with a client session.

SC *do not require client applications to change*, while imposing minimal changes on existing server applications. SC *do not require server processes to synchronize* between themselves or with the client. However, the server OS must include support for

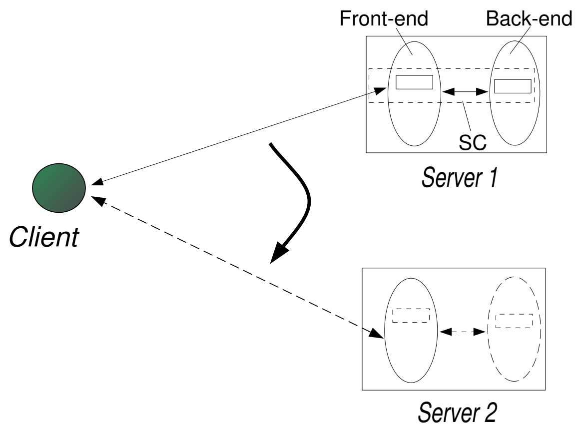


Figure 2.1: *Cooperative 2-process servers use a service continuation (SC) to migrate a client session.*

transparent SC migration. When servers are distributed over wide-area, an efficient implementation of connection migration may require changes to the client-side TCP to make it aware of server endpoint relocation.

2.3.2 An Example

Consider a web-based e-commerce service structured in a two-tier architecture (Figure 2.1). A front-end FE (web server) receives a client request, and a back-end BE translates it into a sequence of one or more database transactions. The two processes communicate over IPC channels (e.g., pipes): FE parses the request and passes its arguments to BE, which executes each transaction in the sequence and sends the results back to FE. The FE forwards the data to the client.

Most of the time, a client may complete its transactions on the server to which it initially connected. However, in case of network congestion or server overload, a client may experience large delays. Instead of cancelling the requests of such clients, their sessions can be dynamically migrated to another server. The migration must be transparent to the client application, e.g., a web browser, so that the user does not have to restart the whole request.

Note, however, that the client request cannot be re-executed from the beginning at the new server for two reasons. First, the request string is no longer available to the new front-end. Second, restarting execution at the new back-end (if possible) would be wrong, as it may cause already committed transactions to execute twice, compromising correctness.

With SC, the session can migrate at any time, without synchronizing the FE and BE processes. For this to happen: *(i)* FE saves the original request in an SC; *(ii)* BE saves the sequence number of the last executed transaction in the SC; *(iii)* the OS includes the state of the pipes connecting FE and BE in the SC.

To continue execution after the connection migrates to the new server, the system migrates the SC, passes the migrated connection to the destination FE, and reinstates the state of the pipes connecting the new process pair now servicing the client. Using the migrated SC, the new FE and BE processes restore state pertaining to the migrated client session (the request and the sequence number of the last executed transaction, respectively), and resume execution. The FE passes the request arguments to BE, and BE resumes service by executing the next transaction in the sequence.

The SC must guarantee exactly-once communication semantics across migration for the new pair of processes and must assist it in providing consistent service to client. SC migration requires a transport protocol that supports dynamic migration of the server endpoint of a live connection.

2.4 The Service Continuation Design

2.4.1 System Model

Essential to the service continuation idea is the assumption that the state of a server application can be logically partitioned among the clients it services, such that there exists a well-defined, fine-grained state associated with each client. Any other non-specific state needed to resume service on a migrated session is deemed accessible at the new server. To migrate a live client session, we assume that there exists a *connection migration protocol* that can be used to transfer the associated state between the kernels of the origin (old) and destination (new) server hosts. The design of such a protocol, itself modeled on the SC idea, is described in Section 2.5.

The state that the server application maintains for a client may span multiple communicating processes in a *process service set* performing work for the client. Processes in a process service set communicate via IPC channels. We assume a uniform abstraction of communication channels (inter-process and client-server) as reliable byte streams.

This computation model is described by Figure 2.2, where the process service set $\{P_1, P_2, P_3\}$ handles the service session of one client. Only a select process in the set, called the root or accepting process (P_1 in Figure 2.2), communicates directly with the client over TCP. The other processes can be either workers (processes spawned on demand and dedicated to servicing one client) or servers (processes that service multiple clients concurrently). The split-state assumption applies to any server process in the service set. In particular, the state of the root process can be logically split among its incoming connections. Every process has a well-defined and reproducible initial state with respect to a client.

Execution of each process in the process service set *with respect to a client* is modeled as a sequence of state intervals. State intervals are delimited by well-defined states reached by a process while servicing a client. Process execution within an interval can be either deterministic or nondeterministic. In a deterministic interval, changes of per-session state in a process are determined only by the data it receives over its incoming communication channels. In a nondeterministic interval, the execution (and the output generated) may depend on the results of nondeterministic system calls made by the process. When a process executes in a deterministic interval, it will always produce the same stream of data on the outgoing channels, if it receives the same stream of data on its incoming channels.

2.4.2 Per-Client Service Continuation

For the server application, a *service continuation* (SC) is a *cut* in the global execution state across all processes, mapped into the most recent well-defined states reached in servicing a given client session by each individual process in the service set. For example, in Figure 2.2, $\{S_1, S_2, S_3\}$ is an SC with three components, each of which describes the point reached by P_1, P_2 and P_3, respectively, while servicing the client.

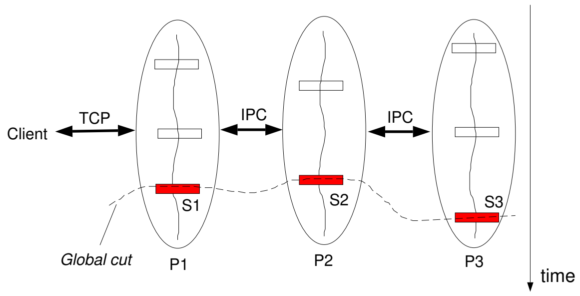


Figure 2.2: *The application view of an SC: a global cut* $\{S_1, S_2, S_3\}$ *through logical states of server processes corresponding to one client session.*

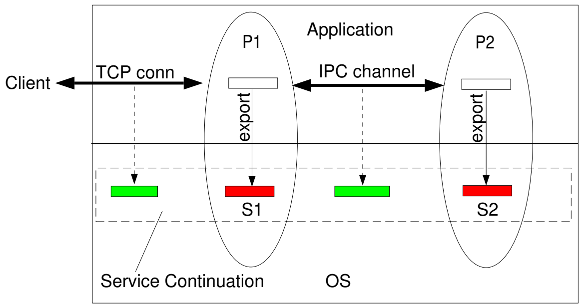


Figure 2.3: *The OS view of an SC: application saved continuations and the state of communication channels.*

Each state component can be individually and independently used by its respective process to resume service to the client. In the OS, an SC is reflected as an ordered set[1] of per-process fine-grained state snapshots, the contents of which is opaque to the system. A process in the service set would declare a continuation point by saving a state snapshot in the SC.

Resuming service from an SC involves more than the simple transfer of session state between two hosts. We identify two requirements from the OS support for service continuations. First, in the OS and at the transport protocol level, a user-level service continuation must be associated with the state of the communication channels (inter-process and client-server). An SC needs OS support for reinstating the state of communication channels at the destination host. Figure 2.3 describes the OS view of an SC, which includes the continuation points S_1 and S_2 saved by processes in the service

[1]For presentation purposes, the process service set is represented as a chain, although in practice the set might be organized as a tree. In this case, there exists a total order, e.g., given by a tree traversal algorithm on the set, that can be enforced by the system. The topology of the set can be specific to a given client.

set, along with the communication state.

Second, the OS must perform synchronization of SC state. Recall that the SC model: *(i)* does not impose any form of synchronization between the server processes, nor between these and the client; *(ii)* does not impose any restriction on when a migration may occur, i.e., migration is completely asynchronous with respect to server or client execution. This requires the OS to synchronize the two endpoints of a communication channel when resuming service from an SC.

2.4.3 Migration API

SC provides a minimal API that allows a server application to enable migration of a client session by establishing continuation points in any process in its service set. A process must export/import a *snapshot* of application state associated with a client to/from an SC object. The exported state is opaque to the OS and to the underlying migration protocol.

The SC service interface can be best described as a *contract* between an application process and the system. According to this contract, a process must execute the following actions: *(i)* upon starting service to a client, associate with an SC all the communication channels it needs to be restored after migration; *(ii)* export state snapshots during service; *(iii)* import the last state snapshot at the new server after a migration and resume service to the client. In exchange, the system: *(i)* transfers the SC state to the new server, and *(ii)* synchronizes the state of processes in the service set with the state of their associated communication channels.

The main primitives of the SC API are:

```
sc = create_sc(conn)
export_state(sc, state_snap)
import_state(sc, state_snap)
associate(sc, ch_set)
sc = open_sc(ch)
```

where `sc` is the SC object associated with a client (an OS-specific identifier), `conn` is

```
/* Front-end process (parent) */
while (conn = accept()) {
  sc = create_sc(conn)
  if (!import_state(sc, req)) {
    receive(conn, req)
    export_state(sc, req)
  }

  args = parse(req)
  pipe = create_pipe()
  associate(sc, pipe)
  if (fork() == 0)
    exec(backend, args)
  else
    while (read(pipe, buff) != EOF)
     send(conn, buff)
}
```

```
/* Back-end process (child) */
sc = open_sc(pipe)
if (!import_state(sc, {last, tid})){
   connectDB()
   last = 0
}
else {
   status = reconnectDB(tid)
   write(pipe, status)
}
goto last + 1
1:  tid = txn_begin() /* 1st */
    ...  do work for transaction 1
    export_state(sc, {++last, tid})
    status = txn_commit()
    write(pipe, status)
2:  tid = txn_begin() /* 2nd */
    ...  do work for transaction 2
```

Figure 2.4: *Pseudocode for the front-end (left) and back-end (right) server processes in the example given in Section 2.3.*

the client connection in the root process, `ch` identifies an IPC channel, and `state_snap` is an application memory buffer summarizing the client session state in a process.

The accepting process of a process service set creates an SC using `create_sc` on the accepted client connection. Processes in the service set use `export_state` to save state snapshots to an SC and `import_state` to retrieve them after a migration. The `associate` primitive is used by a process in the service set to enable service continuation support in the OS for a selected set `ch_set` of its communication channels. The `open_sc` primitive returns the SC object to which channel `ch` was previously associated.

The `associate` and `open_sc` calls transitively propagate channel membership in the SC, starting from the root process down the chain, and allocate system resources for stateful channels that must be restored after a migration. A process must associate channels with an SC before passing or connecting them to other processes and starting communication. The SC object can be either passed between processes, or a process may query an existing channel using `open_sc` to retrieve the SC object it was associated with, then associate its own channels with that SC.

Figure 2.4 shows the pseudocode for the example in Section 2.3, structured as a parent-child process pair, with SC API calls highlighted. The session state consists of the request string `req` (at FE) along with the sequence number `last` and transaction identifier `tid` of the transaction about to commit (at BE). The FE exports `req` to an SC to preserve it across migration. The BE exports {`last, tid`} to the same SC to ensure correct resumption of execution after migration. If the connection is with the original server, `import_state` returns `NULL` and the session state is initialized locally. If the connection is with a different server (after migration), `import_state` retrieves the state exported to the SC at the previous server.

Note that the same code runs at all servers. All SC and IPC calls may fail after a migration, returning an `EMIGRATED` error code; error checking is omitted for clarity. For convenience, we assume that the status of a transaction in the DB system can be obtained at any server by calling reconnectDB() with the transaction identifier `tid`.

If the application follows the terms of the SC API contract, the system supports service resumption in all processes in a process service set and guarantees integrity and consistency of their data streams. The migration API decouples the server application logic from the actual migration time by enabling asynchronous migration, makes the scheme light-weight and yields good performance.

2.4.4 Service Continuation Synchronization

One of the salient features of the SC abstraction is that it does not require synchronization between server processes, nor between client and server. This means that server processes may export their state snapshots at different moments in time and that data may be in transit (in the communication channels) at the time of snapshot. To support seamless and consistent communication after migration, the OS needs to perform the synchronization of independent snapshots and communication channels.

In this section, we describe the mechanisms used by the operating systems of the origin and destination machines, during and after the migration, to synchronize the per-session state of a process and the state of its communication channels. For this we assume, without loss of generality, that a communication channel can be modeled as a unidirectional reliable byte stream between a writer W and a reader R. The client-server connection itself can be viewed as a pair of such streams, hiding the TCP reliability mechanisms.

In order to resume service to a client at a new server, R and W import state snapshots from an SC. Since the corresponding processes at the old server did not synchronize when exporting their snapshots, and because migration is asynchronous with respect to process execution, R and W will not be synchronized when resuming service. Similarly, while executing, the in-kernel data buffering and the active read/write communication cause the state of a channel at a given moment in time to go out of sync with respect to the state reached by R and W as a result of reading/writing data from/to the channel.

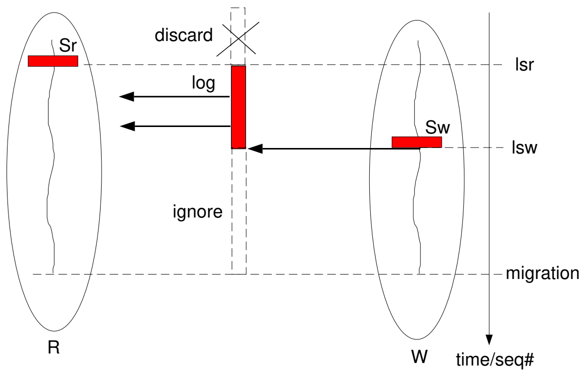


Figure 2.5: *Synchronization with SC. Reader behind writer.*

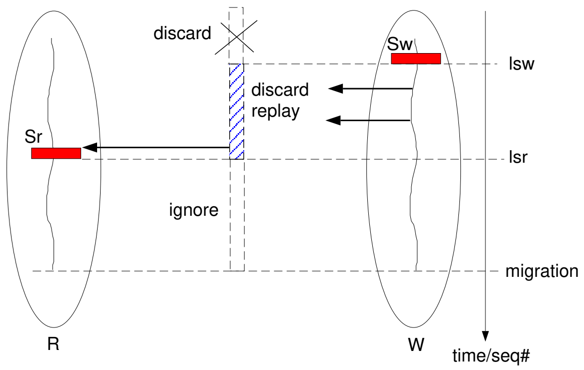


Figure 2.6: *Synchronization with SC. Reader ahead of writer.*

Figure 2.5 shows an example of execution where reader R is behind the writer W in taking its state snapshot. The vertical bar represents a virtual infinite buffer abstracting the channel over which the two processes communicate. For convenience, we use the sequence number of the next byte of interest in the buffer to define a timeline for events in a process. For example, the moment at which a process takes its snapshot is marked by the sequence number of the first byte it will read/write after the snapshot. For R, lsr denotes the first byte to read after taking its last snapshot S_r. Similarly, lsw denotes the first byte to be written by W after its last snapshot S_w. In Figure 2.5, since $lsr < lsw$, upon resuming service at the new server, R will issue reads for data that can no longer be supplied by W.

To solve the problem of synchronization between application-level state and the communication state in the OS, we use a *limited* form of log-based rollback recovery [33] to restore the session state in a process at the new server. Upon restarting service for a migrated client at the new server, a process in the service set imports the last state snapshot from the SC and uses it to initialize its local session state. After resuming execution, the process may *replay* execution already done at the old node since the snapshot. This includes reading data that was read at the old node, and writing data that was already written.

To support the replay of data read from a communication channel, the system must *log* and transfer from the old node data that cannot be generated by the writer at the new node. In Figure 2.5, the synchronization log consists of only the $[lsr, lsw)$ region of the buffer. Data at sequence numbers less than lsr can be discarded as it will not be needed by R after restart. Data in the interval $[lsw, migration]$ can be ignored by the system (not included in the SC state), since W's replay will regenerate it after restart.

Figure 2.6 shows the case when the reader is ahead of the writer in taking its snapshot, i.e., $lsr > lsw$. In this case, during replay, R will read only from sequence numbers (starting with lsr) that can be regenerated by W's replay, and thus ignored by the system. When R takes its snapshot at lsr, the system discards any previous log. During replay, the system will discard any write issued by W in the interval $[lsw, lsr)$.

We note that this synchronization scheme requires the OS to maintain logs of communication activity. However, this should not be a burden since for byte-stream channels the OS kernel already performs buffering. For example, IPC channels buffer data for communication between processes, TCP buffers data in socket buffers, etc. An efficient SC implementation can exploit the kernel buffering, performing zero-copy, in-place logging in kernel buffers.

Because it relies on execution replay, SC synchronization must take into account potential nondeterminism. Recall that the SC model in Section 2.4 allows nondeterministic intervals (NDI) of process execution. When executing in an NDI, writes from a process must not propagate to client or to other processes, as this may trigger state changes that cannot be reproduced by replay. In Figure 2.6, suppose that S_w marks the beginning of an NDI. Then, if resuming from S_w after migration, W may issue different writes from those issued at the old server. Note however that R's state has already advanced based on old writes. In effect, when resuming the session at the new server, the S_r and S_w snapshots will be inconsistent and may cause incorrect replay. This is an instance of the well-known *output commit problem* in rollback recovery [33] which states that nondeterministic side-effects of an execution must not be committed until they become stable.

The solution is to disable write propagation from an NDI, and re-enable it when writes have become stable with respect to a potential migration, i.e., when the NDI can no longer be replayed. This happens at the moment a new snapshot is recorded in the SC with the `export_state` call. With this observation, the problem can be easily solved by annotating NDIs. For example, as a simple extension of the API, `export_state` can include a flag to declare the type of interval the process will enter after export. If the interval is an NDI, the OS will block write propagation on output channels until the next export operation. This solution covers the case of "synchronous" nondeterministic execution, e.g., due to results of nondeterministic system calls. Section 2.6 provides an extensive discussion on issues related to nondeterminism.

The most recent service continuation, including application-level continuations as state snapshots and the communication state, allows any process in the service set to: *(i)* restart servicing the client session from the state snapshot on, *(ii)* replay data read at the old server that cannot be supplied (regenerated) by a writer process or by the client, and *(iii)* supply data it had produced at the old server before the last snapshot (i.e., data that it cannot regenerate by execution replay at the new node). The above three guarantees, coupled with piece-wise deterministic execution and controlled propagation of writes issued in nondeterministic intervals, ensure that the new server can resume service to the client consistently after migration.

2.5 Service Continuation Migration Using Migratory TCP

In this section, we describe the design of Migratory TCP, a protocol for lazy migration of TCP connections between equivalent servers. M-TCP uses the SC model of state synchronization and can be used to lazily transfer SC-encapsulated state along with the migrated TCP connection.

2.5.1 Migration of the Client-Server Connection

The log-based SC synchronization model described in the previous section can be easily mapped into a TCP connection migration protocol to be used to seamlessly migrate the server endpoint of the client-server connection. There are two key observations that make this mapping possible: *(i)* the TCP connection abstracts two unidirectional byte-stream channels, and *(ii)* TCP implements a reliability mechanism based on positive (explicit) ACKs.

For the following discussion, we assume a migration protocol in which the client-side TCP initiates migration (according to some policy, e.g., after a delay in getting a reply from the server) at some moment in time t_{init}. In the intiation phase, the client sends a special control packet to the destination server, which in turn requests the connection state from the origin server. The actual migration takes place at a subsequent moment in time $t_{migr} > t_{init}$, when the state of the TCP connection is transfered from the origin server S_1 to the destination server S_2.

Although the client does not (and cannot) take state snapshots, we can emulate its lsr and lsw by considering the data that the *protocol* endpoints at client C and origin server S_1 have received and acknowledged at the moments when migration is initiated and takes place (t_{init} and t_{migr}, respectively). The "client" lsr^C and lsw^C

values are computed at S_1, at the time of migration t_{migr}, using the protocol state of the connection endpoints.

We use the following notations for the values of byte sequence numbers maintained by the TCP protocol at a connection endpoint E, at a given moment in time t: $una^E(t)$ is the sequence number of the first unacknowledged byte that was sent from endpoint E, $nxt^E(t)$ is the sequence number of the next byte expected to be received at E, and $ack^E(t)$ is the sequence number of the first received byte not yet acknowledged by E (which means all bytes up to $ack^E(t)$ - 1 have been acknowledged). Then,

For the client as a reader:

$$lsr^C = \max(nxt^C(t_{init}), una^{S_1}(t_{migr})) \tag{2.1}$$

For the client as a writer:

$$lsw^C = \max(una^C(t_{init}), ack^{S_1}(t_{migr})) = ack^{S_1}(t_{migr}) \tag{2.2}$$

The first terms under `max` in the above equations account for the client TCP's view of lsr and lsw as, respectively, the next byte to be received from the server and the first byte yet to be acknowledged by the server.

The second terms account for events that may occur between the moment of initiation of migration t_{init} and the moment of actual migration t_{migr}, when sequence number information is transferred from S_1 to S_2: *(i)* bytes from the server that were acknowledged by the client up to the moment of migration can be included in lsr^C since we know they were received by the client TCP, and *(ii)* acknowledgements sent from the server up to the moment of migration could have reached the client TCP, discarding data from its send buffer, so lsw^C must be adjusted.

In the next section, we will describe how the above equations are used in the design of Migratory TCP to synchronize the client and server endpoints of a migrating connection.

2.5.2 Migratory TCP: A Connection Migration Protocol

To make SC practical, a connection migration protocol is needed as carrier of SC-encapsulated state of a client session. The protocol must also support transparent resumption of communication on a migrated client connection.

To prototype our system, we have used Migratory TCP (M-TCP) [90, 92, 83], a protocol designed by us that supports connection migration in single-process server instances. In addition, the protocol enables a stateful server to resume execution by transferring an application-controlled amount of specific state with the migrated connection.

M-TCP provides enabling mechanisms for the cooperative service model of Figure 2.1, in which an Internet service is represented by a set of (geographically dispersed) equivalent servers. A client connects to one of the servers using a reliable connection with byte-stream delivery semantics. The complete client interaction with the Internet service from the initial connect to termination represents a service session. M-TCP enables the session to be serviced by different servers through transparent connection migration. Connection migration involves only one endpoint (the server side), while the other endpoint (the client) is fixed. The M-TCP protocol layers at the old and the new server cooperate to facilitate connection migration by transferring endpoint state.

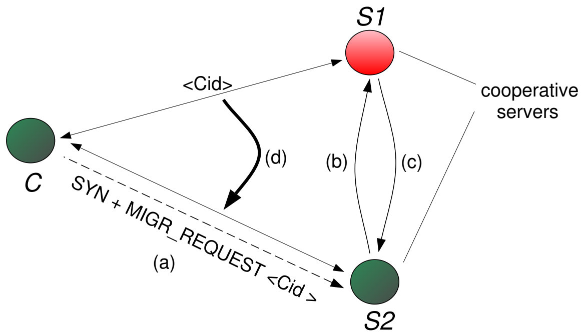


Figure 2.7: *Migration mechanism in M-TCP. Connection C_{id} migrates from S_1 to alternate server S_2.*

Figure 2.7 describes the sequence of steps in a migration. Initially, the client contacts the service through a connection C_{id} to a server S_1. During connection setup, S_1 supplies to the client TCP the addresses of its cooperating servers, sent as a TCP option. In addition, in order to prevent an attacker from exploiting the protocol to hijack connections by issuing fake migration requests, S_1 may also provide the client-side M-TCP with a migration certificate. The certificate is built using an authentication scheme like that devised by Snoeren and Balakrishnan for authentication of a mobile host with open TCP connections after a change in its network attachment point [81]. The client M-TCP will present the certificate to an alternate server, at the time of migration, for authentication as a legitimate initiator of migration.

At some later point during connection lifetime, the client-side M-TCP may initiate migration of C_{id} by opening a new connection to one of the cooperating servers (S_2), sending the migration certificate in a special option (Figure 2.7 (a)). To reincarnate C_{id} at S_2, M-TCP transfers its associated state from S_1 *lazily*, at the time migration is requested. Figure 2.7 shows the lazy state transfer: S_2 sends a request (b) to S_1 and receives the state (c). If the migrating endpoint is authenticated and reinstated successfully at S_2, then C and S_2 complete the handshake, the new connection takes over and S_1 drops its endpoint after receiving an acknowledgement from S_2 for the successful state transfer (d).

During the migration handshake, the client, the destination and the origin server exchange information (sequence numbers, buffered unacknowledged segments, etc.) to synchronize the new endpoints of the connection. M-TCP is heavily optimized to minimize the amount of data (protocol specific state) transferred between servers during a migration. Also, the protocol overlaps the migration with data transfer on the original connection in order to reduce its impact on the client application.

When migration is initiated, the client endpoint of the original connection enters a `MIGRATING` state, in which M-TCP continues to receive, acknowledge and deliver to the client application any data that may still arrive from S_1. While the client application is still able to do I/O operations on its connection endpoint, the `MIGRATING` state blocks the upstream data flow from C to S_1, as it would waste resources to send data to a server from which the connection is moving away. When in the `MIGRATING` state, the connection also ignores incoming ACKs to avoid data loss due to migration. There are two reasons for this: *(i)* the client cannot predict or know the moment at which the actual transfer of state between servers takes place and what the server endpoint state is at that time, and *(ii)* S_1 will keep its endpoint alive until M-TCP acknowledges the successful transfer of its state to S_2. Accepting ACKs sent from S_1 after the state transfer would drop data at the client and make its state inconsistent with that received by S_2.

If migration to S_2 fails, the original connection can either be seamlessly restored to its previous state and continue with S_1 as an endpoint, or it can continue trying to migrate to another server. If migration succeeds, the client-side endpoint of the old connection is forced into a special `BLACKHOLE` state. Its role is similar to that of TCP's `TIME_WAIT` state, with the exception that protocol output from this state is completely suppressed.

Migration of a synchronized connection in M-TCP is possible in any state of its endpoints except for the `TIME_WAIT` state at the client, may occur many times and at any point during connection lifetime without disrupting the ongoing traffic, does not involve the server application, and is totally transparent to the client application.

2.5.3 Connection State Synchronization in Migratory TCP

We next describe the mechanism by which M-TCP achieves synchronization between the TCP state of the client endpoint and the connection endpoint at the new server. We again refer to the steps in Figure 2.7 and describe in detail the information exchanged during migration between the three parties involved. As before, C_{id} is the 4-tuple identifier of the connection between C and S_1.

We denote by lsr^S and lsw^S the sequence numbers of the last snapshots taken at S_1 corresponding to the two unidirectional byte-streams associated with the connection. This means that lsr^S and lsw^S are the sequence numbers of the next byte to be read, respectively to be written *by the server application* at the moment `export_state` was called to record the snapshot. It is precisely this binding between the application snapshot and the TCP sequence numbers of the connection that is central to the synchronization mechanism.

When the server application on S_1 calls `export_state`, the OS records lsr^S and lsw^S, the application-level state snapshot, the TCP connection snapshot (the state of the TCP finite-state-machine, TCP flags, and MSS value) and the socket state snapshot (the state of the socket, socket options, and the linger field value).

The following is the sequence of events involved in migration:

- The client TCP *initiates* migration by opening a new connection to S_2. It sends a `SYN` packet with a special option `MIGRATE_REQUEST`, carrying C_{id} and $nxt^C = nxt^C(t_{init})$.

  During the migration handshake, the client M-TCP maintains both the old and new connection. The old connection is placed in the `MIGRATING` state (in which it can still receive data from S_1 and is still in control of the application socket), while the new connection is hidden from the user application until migration completes.

- Upon receiving the SYN packet with the `MIGRATE_REQUEST` option, S_2 sends to S_1 a **state request** message for the state of C_{id} on a TCP control connection established between the S_1 and S_2 protocol layers. The **state request** carries the C_{id} and nxt^C received in the `MIGRATE_REQUEST` option.

  The new connection is put on hold at S_2 (in `SYN_RCVD`) until a **state reply** message with the state of C_{id} will be received from S_1.

- Upon receiving a **state request**, S_1 computes the client lsr^C and lsw^C, according to Equations 2.1 and 2.2, to determine what portions of the send and receive buffers must be transferred to S_2:

  – The lsr^C and lsw^S values determine the portion of the TCP send buffer to be sent to S_2, as described in Section 2.4.4. If this portion is empty (which means the server snapshot is behind the client TCP), the difference `snd_bytes_discard` $= lsr^C - lsw^S$ is the number of bytes sent and already received by client, to be discarded during the replay by the destination server TCP[2].

  – The lsw^C and lsr^S values determine the portion of the received data logged at S_1 to be sent to S_2.

[2]The discard could be also performed by the client TCP, with some additional manipulation of sequence numbers of the new connection. However, performing it at the server is both simpler and more efficient since it avoids tampering with the sequence numbers and does not uselessly send data over the network just to be discarded.

Data in the receive log *must not* be re-sent by C on the new connection, once established with S_2. For this, S_1 also computes the sequence number $implicit\_ack = \max(lsr^S, lsw^C)$. This acts as an *implicit acknowledgement* for data in the receive log moved to S_2 that the client TCP must discard from its send buffer, and will be relayed to client via S_2 with the `SYN+ACK` packet. Note that lsr^S is included in computing $implicit\_ack$, to account for the case when the receive log is empty ($lsw^C < lsr^S$) yet bytes before lsr^S must not be re-sent by the client.

- S_1 sends to S_2, in a **state reply message**: the application snapshot, the TCP and socket state snapshots, the receive log buffer, the send log buffer, `snd_bytes_discard`, lsr^C, and $implicit\_ack$.

- Upon receiving the **state reply message**, S_2 extracts the state components from the message, appends the send and receive logs to the TCP buffers of the new connection, and records the `snd_bytes_discard` value. TCP and socket state snapshots are saved and will be used to restore the TCP and socket state *after* the TCP handshake with the client completes.

  S_2 sends a **state ack** message to S_1 to acknowledge receipt of the state and confirm that resources have been successfully allocated for the migrating endpoint. This signals to S_1 that it can safely drop the old endpoint of C_{id}.

- If it has successfully received the connection state from S_1 and allocated resources for the migrating connection, S_2 continues the TCP connection establishment handshake with C by sending the `SYN+ACK` to C on the new connection, with a `MIGRATE_REPLY` TCP option. The `MIGRATE_REPLY` option carries the two numeric values computed by S_1: lsr^C and $implicit\_ack$.

  If the state transfer from S_1 was not successful or resources could not be allocated at S_2 for the migrating connection, S_2 sends a *RST* on the new connection. This allows the client TCP to give up migrating to S_2 and continue, by trying to migrate to another alternate server or by falling back to the original connection with S_1.

- Upon receiving the SYN+ACK with the MIGRATE_REPLY TCP option, the client TCP knows that the state of the server endpoint of the old connection is safely in place at S_2. It then performs the switch between the old and new connection, transparently to the application:

  – it copies the finite-state-machine TCP state, the MSS and other user-set options from the old to the new endpoint;

  – it shuts down the old connection endpoint by moving it into the BLACKHOLE state;

  – it connects the new connection to the application socket, so that any user level I/O on the socket will now communicate with S_2.

  The client also uses *implicit_ack* to correctly trim its TCP send buffer to avoid sending duplicates of bytes already transferred to S_2 from S_1.

- Because M-TCP overlaps migration with receiving data from the origin server, in-flight bytes may have trickled in on the old connection while in the MIGRATING state. As a result, the client's view of lsr may have advanced as compared to the lsr^C computed by S_1. For this reason, C must send a "fixup" control packet to S_2 to prevent it from sending the bytes it has received after lsr^C.

  To achieve this, C computes the number of bytes it had received during migration as the difference between S_1's view of lsr^C at the moment of migration (received in the MIGRATE_REPLY option) and the current sequence number of the last byte received by client:

  $delta\_fixup\_rcv = nxt^C - lsr^C$

- C completes the 3-way TCP handshake by sending an ACK on the new connection, with a TCP option RCVD_WHILE_MIGRATING carrying the *delta_fixup_rcv* value.

- Upon receiving the ACK with the RCVD_WHILE_MIGRATING option, S_2 discards up to *delta_fixup_rcv* bytes at the front of the TCP send buffer. If the buffer has less bytes than the fixup value (including 0), the difference is added to the snd_bytes_discard value to be discarded from the bytes generated during replay by the new server.

This completes the 3-way TCP handshake on the new connection to S_2. At this point, the server M-TCP can complete the migration: it restores the state of the TCP endpoint and the socket state from the snapshots it has received from S_1, then places the connection on the accept queue of the local server application.

- Upon receiving the **state ack** message, S_1 kills its endpoint of the original C_{id} by moving it into the CLOSED state and marking the socket as migrated. Any further application call on the socket will return an EMIGRATED error.

For simplicity of presentation, the above description assumes that both endpoints of the connection are in ESTABLISHED and no state transitions take place during migration. During all migration operations, our M-TCP implementation observes boundary conditions, i.e., it correctly migrates endpoints in terminal TCP states (FIN_WAIT1, FIN_WAIT2, CLOSING, CLOSE_WAIT, LAST_ACK) at both server and client, including the cases of half-closed connections. Connections in TIME_WAIT at the client side are not migrated simply because it does not make sense to migrate them.

To this end, the server-side M-TCP maintains a small per-connection log of state transitions of the TCP state machine triggered by asynchronous events like receipt of FIN segments and their ACKs. The state transition log is transferred to the new server (S_2) as part of the connection state and used to correctly replay the same transitions on the reincarnated endpoint. This ensures that the migrated endpoint performs the same transitions in the final stages of a connection at the new server and ends up in the same state, consistent with that of the client. Also, for correct operation, in all sequence number calculations described above, M-TCP keeps track of the exceptional cases of FIN and SYN accounting (since FIN and SYN consume one byte in the sequence number space without a corresponding data byte).

2.6 Discussion

2.6.1 Service Continuations and Nondeterministic Execution

The execution model adopted by SC (Section 2.4) assumes that there exist deterministic execution intervals in a process during which the only *external* cause of changes in the state of a given session is the contents of its byte-stream input channels. This model limits the sources of nondeterminism in an application to its input channels in order to achieve a practical solution to what essentially is a distributed recovery problem [33]. SC exploits this model in its synchronization scheme.

In distributed consistent recovery protocols, communication channels are a major source of nondeterminism. To deal with it, processes either coordinate during normal operation to establish a consistent recovery line, or compute it during recovery. None of these is a feasible solution in the case of Internet service sessions, as we do not want to force coordination on processes and, moreover, we have no control over those running on the client side. By replaying and generating (within deterministic intervals) the same sequence of bytes on input/output channels before and after migration, an SC ensures deterministic behavior with respect to a client throughout its process service set and eliminates reads from communication channels as a source of nondeterminism.

The remaining *known* sources of nondeterminism in the execution of a process reside in all the other synchronous interactions with the kernel (system calls). They can be eliminated if the application can ensure that a nondeterministic change in its state does not affect the environment of a process (including its output channels), so ultimately it is not made visible to the client. This can be achieved using application knowledge in two ways: *(i) annotate* nondeterministic intervals as described in Section 2.4.4, relying on the OS to block output propagation from such intervals until they are closed, or *(ii)* take a snapshot before committing a nondeterministic change of state (i.e., before propagating its effects on output channels) and include it in the snapshot. For example, if the server must send to a client the result of a `gettimeofday()` call, it must take a snapshot before the send, recording the result in the snapshot. If the session migrates to another server, then, after restart from the snapshot, the new server must send the value imported from the snapshot instead of executing `gettimeofday()` locally.

Note that our SC synchronization model cannot account for purely asynchronous events like signals. Replay of asynchronous signals is a hard problem in itself which was not directly solved even in dedicated fault-tolerant systems like [14]. Such systems achieve accurate signal replay in two steps: *(i)* convert signals to messages sent over *signal channels* and log them at dedicated backup processes; *(ii)* keep a primary-backup process pair strictly synchronized at the time a signal is delivered, so that a restart of the backup always begins with a signal delivery event. A similar scheme could be used with SC if the system can take forced snapshots in a process, which may not be possible in general. What the SC synchronization scheme essentially achieves in the general case is a reasonable tradeoff between controlled nondeterminism and system complexity.

2.6.2 Migration Policies

SC provide only the mechanism to support migration of live instances of service sessions. Definition and evaluation of migration policies is beyond the scope of this work. In [90] we proposed, in the context of M-TCP, a migration architecture that decouples a migration mechanism from policy decisions, including the events that trigger a migration. This allows various migration policies to be designed and evaluated independently. In this architecture, migration triggers can be placed: *(i)* at the client side, to dynamically switch to another server if the current server is perceived as unavailable or if it does not provide a satisfactory level of service, or *(ii)* at the server side, under control of the server application, to implement a load balancing scheme by shedding some of the connections to other less loaded servers, or to implement an internal policy on content distribution among groups of clients. In two of the experiments in Section 2.7.2, we use sample migration policies designed for experimental purposes which can also be applied in practice.

2.6.3 Classes of Applications

Under the cooperative service model, SC can be used to react to adverse conditions that hamper service availability and/or quality of service received by clients. Such conditions include server overload, network failure or congestion on a given path, etc. In these cases, migrating the session to another server ensures sustained service to the client.

To benefit from migration, an application must be willing to incur a cost, which should be amortized in terms of either better service quality or increased service availability over the lifetime of the connection. For example, it does not make sense to enable migration for short lived connections when the service is not critical to the client. We identify two classes of services that can benefit from SC support:

(i) Applications that use long-lived reliable connections. Examples are multimedia streaming services [41], applications in the Internet core (e.g., BGP [72]) that use TCP communication over large spans of time, etc. In Section 2.7.2 we describe our experience with such applications.

(ii) Critical applications. Characteristic to this class is that users expect both correctness and good response time from the service. Examples are Internet banking, e-commerce, on-line trading, etc. In [91] we made a detailed case study of integrated system support for migration in a PostgreSQL transactional database system [70] accessed over the Internet, that can be used in this class of applications.

2.6.4 Service Continuations Programming Tradeoffs

With the SC API, a server application may have to exploit certain performance tradeoffs. One tradeoff is between the size of the state snapshot and the amount of SC state in the system. Delaying export of a snapshot for too long in order to optimize its size may increase the amount of logs. In general, a reader from a "fat" communication channel should take snapshots frequently to allow the system to discard logs it maintains for it. In Section 2.7 we describe an efficient implementation of the export primitive that practically eliminates the overhead of frequent snapshots.

Another tradeoff is between the snapshot frequency and the amount of work redone at the new server after migration: a larger frequency may mean more overhead but less work to be redone. Increasing the snapshot size may, in some cases, capture more computation and might result in less work being redone after restarting at the new server. On the other hand, larger snapshots may increase the run-time overhead and impact the migration time.

Programming with SC requires an effort in understanding the SC model (Section 2.4) and defining computation of a server process on behalf of a client as a sequence of state intervals. Most existing servers work in this way. Still, programming errors may occur if application logic does not obey the SC model, as illustrated by the following example. Suppose a process servicing a client reads a chunk A of data from a channel and, before consuming it, it immediately exports to an SC a snapshot S that does *not* reflect a change in the client session state as a result of reading A. If a migration occurs right after the export, the corresponding process at the new server will resume service from S. If the process naively attempts to "read" A now, will instead receive the next chunk in sequence. While it may appear as if chunk A has been "lost," this behavior is in fact normal. The OS discarded A from the channel log at the time S was exported, as A had been read before and the OS assumes that S incorporates changes in session state caused by A. A correct program must not attempt/expect to "read" data it has read before exporting a snapshot since, according to the state-driven execution model of Section 2.4, the state snapshot must reflect the fact that the read has already been performed. Because an `export_state` call aggressively discards logs from the system, data that has been read from a channel is not "safe" until written on another channel or reflected in a state snapshot.

However, it is possible that a process does not *directly* change the session state based on data it reads from a channel. As an example, consider a process whose only task is to forward data in a process chain by reading chunk A from an input channel and immediately writing it to an output channel. While apparently the process does no explicit computation based on A, its state with respect to the client session *does* change after the read, as it now includes the raw data in A. This buffered raw data is *volatile* with respect to migration between the read from a channel and its forwarding on another channel. For this reason, the process must ensure that, at the time it does an export, it has forwarded on its output channels all data it has previously read. If not, it has to save this data in an exported snapshot to avoid losing it in case of migration.

With SC, a server application must obey a certain programming discipline, using our API to export/import to/from the system state associated with a potentially migrating client session. The client application is not required to change. From our experience with three real, widely-used server applications described in Section 2.7.2, we believe that the programming effort involved in using SC is fairly low, and expect the API not to be very intrusive to application logic. Adhering to a certain programming discipline and API is the effort a server writer may want to invest in order to take advantage of dynamic server-side migration of live client sessions.

2.7 Prototype and Evaluation

We have implemented SC in the FreeBSD 4.3 kernel, including support for migrating TCP sockets and OS pipes. We use M-TCP [90] as the underlying connection migration protocol. Our M-TCP implementation is compatible and inter-operates with the existing TCP/IP stack. The protocol works as an extension to TCP, with M-TCP control information sent as TCP options. For state transfer between two server hosts, M-TCP uses a TCP connection established between the kernels of the two machines.

In terms of overhead, supporting SC requires *time* (spent for export operations and logging data in the system) and *memory* (used to store SC state). Logging is essentially zero-cost in terms of time, being done in-place in already existing kernel buffers, on every write operation. However, since logs are kept in system memory, their size must be limited.

An important performance issue with the SC API is that a server process must cooperate with the system and export state snapshots. As seen in Section 2.4.4, an export operation by a reader is beneficial in two ways: it discards logs maintained in the system, and reduces the amount of state to be transferred to a destination server during migration. In general, an export operation may also benefit the application as it reduces the amount of work to be re-executed after migration. However, to the application, frequent exports mean run-time overhead during migration-free execution.

We thus have two conflicting requirements: for the system, frequent exports are desirable to reduce resource usage; for the application, they may incur high overhead during migration-free execution. This means that the `export_state` primitive must be carefully implemented.

One way to implement the export primitive is *eager export*, which copies the snapshot in the SC on every call. Eager export suffers from the unavoidable overhead of copying state from user to kernel space, which may become significant if snapshots are large and frequent. To optimize it, we provide an alternative primitive called *lazy export* which achieves better performance in two ways: *(i)* it defers *recording* a snapshot until the first `read/write` operation of the process on one of its communication channels, thereby saving an extra system call; *(ii)* it delays *copying* of the snapshot into the SC until actually needed, i.e., until migration time.

To use lazy export, the calling process must use a special `register` system call to register a pair of alternating snapshot buffers with the SC. The `register` call pins the user-level buffers into physical memory and maps them into kernel memory; the system copies the last snapshot into the SC at migration time. When a new snapshot becomes available, the process increments a sequence number in the buffer to help the system identify it as the most recent. A pair of buffers is needed since migration may occur asynchronously and the snapshot might be copied into the SC while its buffer is being modified in the application. With these optimizations, the cost of lazy export reduces to a one-time `register` operation. A performance comparison is presented in Section 2.7.2.

2.7.1 Microbenchmarks

We perform two experiments to estimate the migration costs and the overhead of the SC API. The experimental setup consists of two identical Intel Celeron 400MHz PCs as servers and a 233 MHz Pentium II PC as client, running our modified FreeBSD 4.3 kernel that includes SC and M-TCP. Nodes have 128 MB RAM and are connected by 100 Mb/s Ethernet on an isolated subnet. Server hosts are connected through a second 100 Mb/s Ethernet interface dedicated to M-TCP control traffic. The cooperative server

State size [*KB*]	*1-process SC* [μs]			*2-process SC* [μs]		
	C	S_2	S_1	C	S_2	S_1
0	360	252	77	485	358	132
1	503	388	86	608	488	146
5	980	865	135	1,066	945	183
10	1,590	1,473	132	1,681	1,557	232

Table 2.1: *Breakdown of migration time for one-way migration with 1-process and 2-process SC.*

applications service requests on the primary (service) interfaces, while M-TCP uses the secondary (control) interfaces.

Migration Costs

The first experiment estimates migration costs for one-process and two-process SC, as a function of SC state size. In the two-process case, server processes are connected by three OS pipes. We use a migration-enabled synthetic server application that does not generate data. This eliminates any logging for SC synchronization, and thus variability in the SC state size. We control the amount of SC state by varying the size of exported snapshots in the server from 0 to 10 KB. In the two-process SC, the whole state is exported by the root process. The client-side M-TCP initiates a one-way migration.

Table 2.1 shows times measured at the client (C), new (S_2) and old (S_1) server, for the one-process and two-process SC. The values shown are (in the order of actions starting with the migration initiated at the client): *(i)* C: time to wait for migration completion; *(ii)* S_2: time to fetch SC state; *(iii)* S_1: time to prepare a SC state reply. Times are measured inside the kernel, using the processor cycle counter. To eliminate the impact of the nondeterministic network on the fine-grained timers, we selected values corresponding to the minimum client waiting time observed over 200 runs.

Table 2.1 shows that the migration cost is, as expected, dominated by network latency, mainly due to the transfer of SC state from S_1 to S_2. We have also measured the time spent in host processing, for manipulation of SC state at the server side. Our in-kernel SC and M-TCP implementation takes only 213 μs and 331 μs at the two

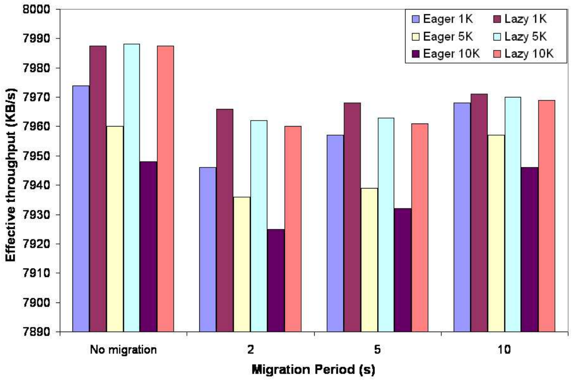


Figure 2.8: *Throughput of a TTCP transfer from a single-process SC-enabled server.*

servers combined, for the single and two-process SC, respectively, for a state size of 10 KB. These results show that SC migration can be implemented efficiently.

Migration Overheads

The second experiment evaluates the migration-free run-time overhead of using the SC API and the overhead of migration by measuring their impact on client-perceived performance. We modify the TTCP benchmark [95], using SC on the server side to migrate and resume a data transfer at a new server from the point where the previous server left off. The performance metric is the effective throughput perceived by a client receiving a 400 MB stream. Throughout a run, the client connection is either stationary (has a fixed server endpoint), or migrates periodically between two TTCP servers.

We run experiments with single-process and two-process TTCP servers. In the two-process case, the second process writes data over a pipe to the first process, which sends it to the client. A process takes state snapshots after every 8 KB of data sent. In the two-process case both processes take snapshots using the *same* variant of the `export_state` primitive, i.e., either eager or lazy. A snapshot consists of one integer for the position in the stream reached by the server, padded up to the state size required by a run.

We first measure the throughput sustained by a "base" TTCP server that does not use the SC migration API, therefore does not incur run-time overhead for migration

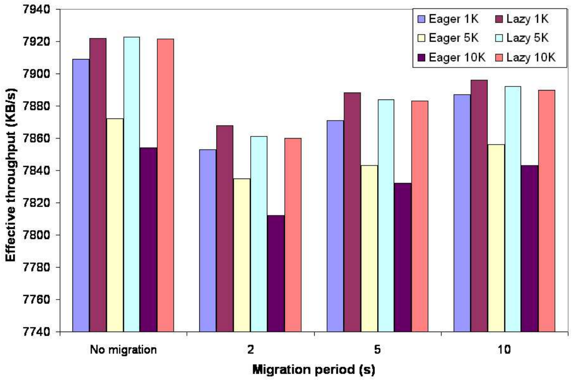


Figure 2.9: *Throughput of a TTCP transfer from a two-process SC-enabled server.*

support. The values observed for the single and two-process base server are 7,988 KB/s and 7,934 KB/s, respectively.

To estimate the overheads, we next run two SC-enabled cooperative servers and measure the throughput seen by a stationary client and by clients that migrate at intervals of 2, 5, and 10s, respectively. We run each client for exported state sizes of 1, 5, and 10 KB. For a given size, we repeat the experiment with servers that differ in the variant (eager or lazy) of the `export_state` primitive they use. To eliminate network-induced variability, we take the maximum values observed over 200 runs. Figures 2.8 and 2.9 plot the results for the single-process and two-process server, respectively, scaled to emphasize differences otherwise small in absolute value.

The loss in performance from the base server case for a stationary client gives a raw measure of the run-time overhead of adding SC to an existing server. For a migratory client, this performance loss is further compounded by migration-induced costs.

In Figure 2.8, the stationary client has no performance loss with a lazy server, while with an eager server it exhibits an overhead increasing with state size. Of all migratory clients, the largest hit (under 1%) is taken by a client with the smallest migration period (2 s), serviced by eager servers, for the largest amount of per-client state (10 KB).

In the two-process case in Figure 2.9, the stationary client with a lazy server has a small loss in performance when compared to the base server, regardless of the state size. With an eager server, the loss increases with the state size up to 1%. The fact

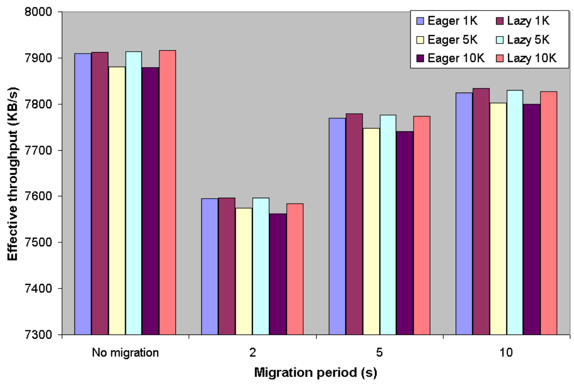


Figure 2.10: *Performance of the TTCP two-process transfer when the second process does not take snapshots.*

that in the lazy case the loss is not correlated to the state size confirms our expectation that a lazy server should not introduce direct overheads related to taking snapshots (as also seen in the single-process case). The largest performance hit taken by a migratory client in the two-process case is 1.5%, again with the smallest migration period and eager 10K servers.

Several observations on trends exhibited by the two figures with respect to server type, state size and migration frequency: *(i)* a lazy server does not introduce any run-time overheads related to state size, and introduces no overhead for a single process server; *(ii)* a lazy server performs consistently better than its eager counterpart; *(iii)* the eager server performs worse with increasing state size; *(iv)* for the migratory clients, the migration overhead increases with increasing migration rate, across all state sizes and server types; *(v)* for the same migration rate, the eager server performance decreases more abruptly with state size as compared to that of the lazy server. These observations confirm behaviors we expected from the SC design and implementation.

Figure 2.10 illustrates a particularly interesting case of a TTCP two-process server that demonstrates the impact of write discards during replay at high migration rates. In this experiment, the first process (the pipe reader) takes snapshots, while the second process (the pipe writer) does not. As a result, the SC state of the pipe records that, after a migration and restart, writes by the second process must be discarded before it is

State size [KB]	*eager export* [μs]	*import* [μs]	*associate* [μs]	*register* [μs]	*unregister* [μs]
1	22	26	150	124	69
5	47	68	151	148	80
10	114	121	154	154	81

Table 2.2: *Cost of SC system calls.*

allowed to generate new data for the reader (and implicitly for the client). Because the reader does advance its state (the position in the stream) with every snapshot, while the writer always starts its replay from the beginning of the world, more and more data from the second process has to be discarded after each migration. In addition, the lower the migration period, the more time the writer spends doing useless work because its writes are discarded, relative to the time during which it generates new data. Less new data generated from each server translates into an overall drop in throughput with increasing migration rate, visible in Figure 2.10 when compared against Figure 2.9. This example demonstrates the effects of "bad" application behavior (i.e., a programming error) and makes the point that an application should be concerned with the length of replay in case of migration. Note that in this case there are no pipe logs, since the reader's snapshots are always ahead of the nonexistent snapshots of the writer. The server processes should, however, stay "close" enough in taking snapshots, in order to reduce potentially long, useless replays.

Cost of SC Primitives

Table 2.2 shows the cost of SC primitives, measured as time that a process spends in the corresponding system calls. All calls, except for `export`, are one-time operations at a given server. The cost of eager `export` increases with state size, as does that of `register`. Recall, however, from Section 2.7 that `register` is a helper call used with lazy export, issued only once by a process per serviced client. In contrast, in an eager server, `export` is called every time a process must take a snapshot (about 50,000 such calls are made by a process during a TTCP run).

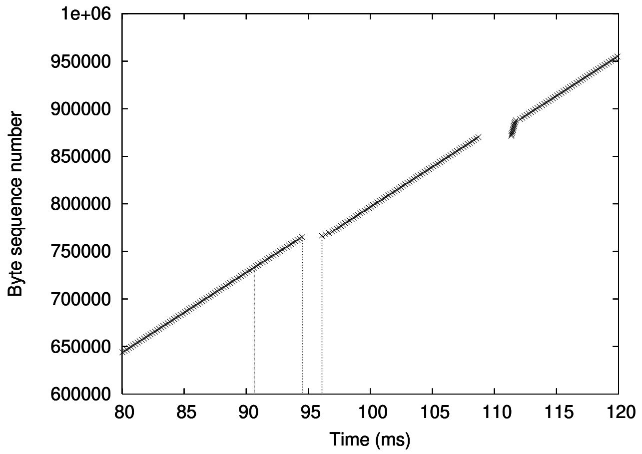


Figure 2.11: *TTCP transfer trace. The first gap corresponds to a migration.*

M-TCP Migration Performance

To further understand why a long transfer suffers such a small loss in performance even with frequent migrations, we collected a trace of sequence numbers of TCP segments received by the client TCP from single-process eager 10 KB servers. Figure 2.11 shows a detail of the trace, including a migration (first gap) along with another event that disrupts the transfer. Both time and sequence numbers are relative to the first recorded segment arrival. The first gap in the trace is caused by a migration, while the second disruption is caused by a random anomalous event in the system.

The vertical lines mark, in order from left to right: the first segment seen from the old server *after* migration was initiated, the last segment seen from the old server, and the first segment received from the new server after migration. Note the high degree by which M-TCP overlaps the migration with data being received and delivered from the old server.

The second disruption in Figure 2.11 is likely caused by a burst of incoming segments which generates a flurry of interrupts from the network interface. This prevents normal input protocol processing by the receiver, until the sender stops sending because the receiver window has filled up. During the burst, the received segments are queued up. When TCP/IP protocol processing resumes, it first consumes all queued segments, proceeding at a high rate (as seen from the steep slope of the trace after the gap), then continues at the steady arrival-driven rate.

Comparison of the two gaps in the trace shows that a disruption due to connection migration with M-TCP is not much worse than those caused by other events in the system, which means that migration should not have a heavy impact on the client. The low migration costs and the ability of M-TCP to deliver data from the old server during migration explain the overall good performance of SC and M-TCP migration.

Streaming Performance with SC Migration

This experiment tests the sensitivity of an SC migration-enabled streaming service to degradation in server performance, when using a rate-based migration policy. We use a synthetic single-process streaming server that sends data to a client at regular intervals as a sequence of chunks of 1 KB. After sending a chunk, the server takes a snapshot (using the eager export primitive) recording the position it has reached in the stream. After sending 32 KB on a connection (new or migrated), we simulate a degradation in server performance by gradually increasing the delays between successive chunks. This behavior affects the received data throughput as perceived by the client.

On the client side, an in-kernel migration policy module uses the inbound data rate as metric and triggers migration when the estimated rate drops by 25% from the maximum rate seen on the connection from the current server. We use a simple smoothed estimator S_rate for inbound data rate, with a low-pass filter of the form $S\_rate = \alpha * M\_rate + (1 - \alpha) * S\_rate$, and a smoothing factor $\alpha = 1/8$, where M_rate is the measured rate sample. While this may not be a proper rate estimator for TCP traffic [26], it serves its purpose in this experiment. As the delays engineered at the server are gradual, the estimator gives a good measure of the effective rate.

Figure 2.12 shows the trace of byte sequence numbers observed by the client-side M-TCP on the connection, up to the maximum of 256 KB, where we cut the stream. The graph shows four cases, one without migration and three with migration for SC state snapshot sizes of 2, 10, and 16 KB. The stationary trace exhibits the decaying service profile of a server. In the migratory traces, each slope discontinuity corresponds

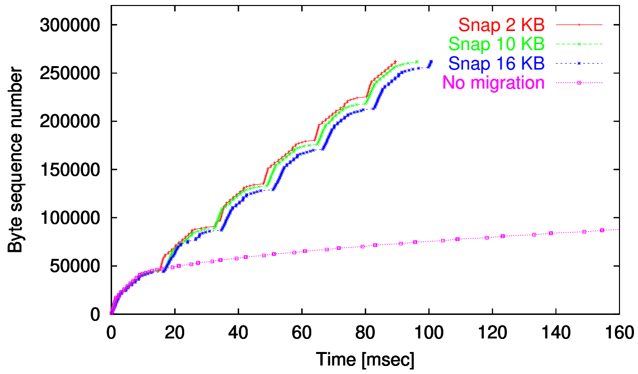


Figure 2.12: *Traces from a single-process streaming service on stationary and migrating sessions. Migration to an alternate server takes place when the data rate drops by 25% from the maximum rate seen from the current server.*

to a migration, after which data is received at the best rate from the new server. The net result is that the effective rate at which the client receives data is fairly close to the average sending rate of a server before its transmission rate drops sharply. The graph also exhibits the cumulative effect of the time taken by each migration and of the (eager) export overhead, reflected in the longer time it takes the client to receive the stream as the snapshot size increases. The experiment shows that the effective throughput perceived by a client improves by transparently migrating the connection in case the server cannot provide a satisfactory rate, and validates the feasibility of using SC for sustained streaming service.

2.7.2 Real Applications

In this section, we describe our experience with using SC in several real servers. We present an extensive evaluation of a web server in which session migration is used to improve the number of successfully completed transfers within a given deadline from an overloaded server, thereby improving the client-perceived availability of the web service.

Web Server. We have modified the Apache [3] web server to use our SC API, enabling transparent migration of client browser connections between M-Apache (Migratory Apache) servers. The SC API adds just 50 lines of code to base Apache and

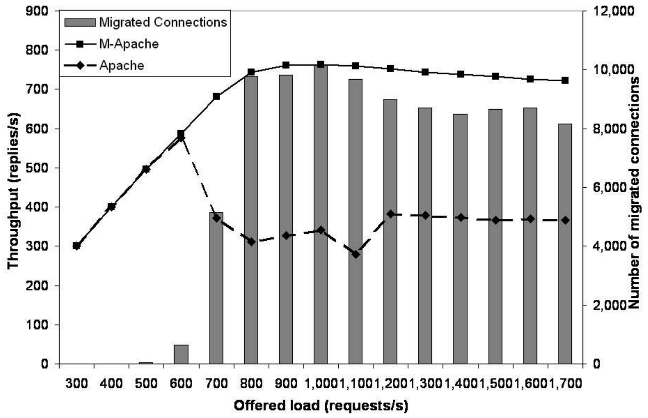


Figure 2.13: *Throughput of successful replies from Apache and M-Apache. The bars represent migrated transfers in M-Apache.*

supports migration for transfers of both static and dynamic (CGI) content. A state snapshot in M-Apache consists of the client request and one integer for the file offset reached during transfer. M-Apache exports a state snapshot after every 8 KB sent to a client.

We run two load experiments on base Apache and M-Apache to show that SC can improve availability of the service under load while not affecting server performance. The load is generated by httperf [62] clients running on four 300 MHz Pentium II PCs over 100 Mb/s Ethernet. The servers are 550 MHz, 1 GB RAM Pentium II PCs. We use a real trace collected on the Rutgers CS web server over 18,370 files with average file size 27.3 KB and average reply size 19 KB. Each run of the trace lasts 300s. A transfer that has completed within 20s is considered successful, otherwise is counted as failed. The performance metric is the number of successful replies/s.

In the first experiment, M-Apache uses SC to increase the number of successful transfers by migrating the sessions that do not complete within 10s to a second equivalent server. Figure 2.13 plots the throughput with base Apache and M-Apache. Vertical bars show the number of sessions migrated by M-Apache. At small loads both servers perform identically and, as expected, M-Apache performs very few migrations. Base Apache reaches saturation at 600 requests/s, while M-Apache continues to sustain

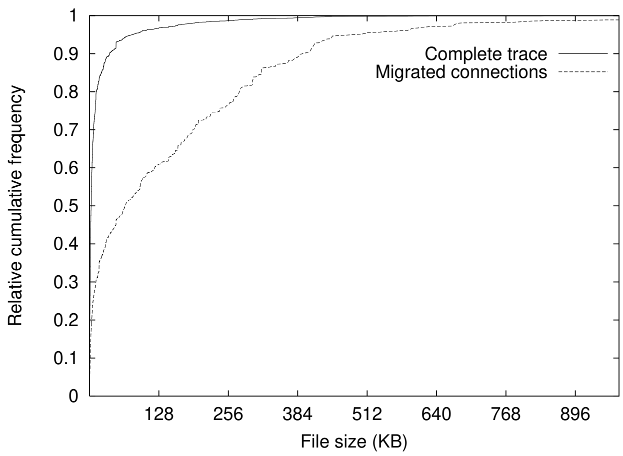


Figure 2.14: *Cumulative frequency of file sizes in the trace and in transfers migrated by M-Apache.*

throughput well beyond this point by migrating ongoing transfers. When M-Apache reaches saturation at 800 requests/s, it can sustain twice the throughput of base Apache by migrating just about 10,000 of the 240,000 requests in the run. The average state size of a migrated SC was 9,386 bytes.

To explain this result, Figure 2.14 compares the distribution of file sizes over all requests against the file size distribution on migrated connections. While most requested files are small (90th percentile less than 128 KB), file sizes for migrated transfers are much larger (90th percentile larger than 384 KB). Migrating requests for large files allows the server to service more requests for small files and thus sustain higher throughput. Finally, note that the system was able to sustain up to 10,000 migrations under heavy load, proving its ability to handle high migration rates under load.

The second experiment evaluates the run-time and memory overhead of using SC in M-Apache for the two variants of export primitive (eager and lazy). We run the three servers under the same static load, i.e., without migrating transfers in M-Apache. We observe the same throughput behavior for all servers regardless of load. The M-Apache snapshot amounts to a fixed part of the request of 18 bytes, plus, for our trace, a constant 24 bytes for the anonymized file name. Internally, a per-client SC uses an amount of system memory equal to the snapshot size for eager M-Apache, and a worst

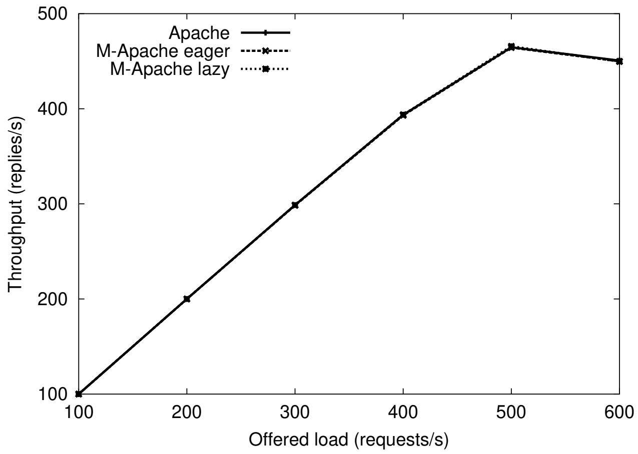


Figure 2.15: *M-Apache web servers exhibit no throughput loss compared to the base Apache server.*

case of 5 memory pages for lazy M-Apache.

Figure 2.15 plots the reply throughput with increasing request rate until saturation, the perfect overlap of the three curves showing that M-Apache servers have performance similar to Apache regardless of load. The system memory overhead is negligible for eager, and a maximum of 1.44 MB for lazy M-Apache. We conclude that the use of SC to support migration in M-Apache has practically no impact on server or system performance.

In summary, our experiments show that the availability of a web service can be improved through lazy session migration. We make three important observations on the methodology we used in our first experiment. First, our metric (throughput of *successful* transfers within a given deadline) qualitatively reflects the service availability as perceived by the clients. In most cases, human users acting as clients will abort a transfer and declare the service unavailable if the request is not serviced in a finite amount of time. Second, such a metric also applies to situations in which the service is unavailable due to network failures. Our experiment provides an extreme-case scenario of client-perceived unavailability combined with overload on the server that stresses the system to the maximum extent. Third, session migration achieves the re-distribution of *ongoing* transfers, which is different from front-end load balancing techniques since it is performed dynamically and during overload, when clients already perceive a degradation in service. The key finding is that lazy session migration can improve the end-to-end availability of an Internet service and that, when efficiently implemented, it works even when the origin server operates under extreme load conditions.

Transactional DB Server. We have integrated migration support through database reconnects and queries for the state of a transaction in the PostgreSQL transactional database system [70]. We have built a sample web-interfaced PostgreSQL application that uses SC and runs as a CGI script in M-Apache. The system allows a client to start a sequence of transactions with one front-end and transparently continue its execution on other front-ends if necessary, while preserving ACID semantics and determinism. The design of Migratory PostgreSQL is described in detail in [91].

Audio Streaming Server. We have incorporated SC in Icecast [41], an open-source Internet audio streaming server. The SC API adds 350 lines to the base server code. The server exports a state snapshot after every 8 KB sent to a client. A snapshot depends on the type of client media player, but has a core size of about 50 bytes. The use of SC enables client streaming sessions to transparently migrate between Icecast servers, without interruption or distortion of received streams.

2.8 Future Work

The SC idea opens interesting avenues of research in fine-grained OS support for fault-tolerant and restartable systems and services. We plan to develop and apply it to these areas in our future research.

SC can be augmented with support for fault tolerance by making the volatile SC-encapsulated state persistent across server crashes. Two solutions are possible: *(i)* use some form of stable storage to store the SC state on the server side, or *(ii)* store the SC state with the client. In the first case, due to the highly dynamic nature of the SC state, the stable storage support has to be efficiently implemented. This is possible for example if the servers are connected (in a cluster) through a communication architecture that supports RDMA operations, e.g., VIA [96], Infiniband [1]. This idea has been applied succesfully in previous work [103], where memory-mapped communication was used for fast failover of nodes in a cluster by establishing efficient checkpoints in the memory of other nodes.

A mechanism similar to SC can be developed and used in the area of system restartability for recovery-oriented computing (ROC) [67]. Restarting a system may be necessary, for example, because its internal state has been corrupted due to software bugs, environmental factors, etc. An OS mechanism similar to SC may be used to preserve some portion of system and application state that can be salvaged, like that associated with serviced clients. An SC can be used to (temporarily) transfer the clients to other servers while the system is being restarted. The server application can later resume from this state and rebuild missing components based on internal semantics, external (logged) input, etc.

We also intend to explore how SC can be used in IP-based storage networking for providing transparent fault tolerance and load balancing. Newly emerging industry standards for the transport of SCSI storage protocols over IP, like iSCSI [74], use TCP connections from a client to a "target" host that controls the storage device. A device server receives commands, relays them to the device, and returns results to the client. Such a system presents an interesing case for SC, as client connections are long-lived, have unpredictable load, and are sensitive to host unavailability or device failure. SC can provide an immediate mechanism for offloading client iSCSI connections to alternate storage device servers, for example in case of an unexpected load surge or to hot-swap a host system. In systems with replicated storage, SC can provide transparent failover in case of device failures. We plan to study the possibility of using SC in the iSCSI protocol.

2.9 Summary

In this chapter, we have described Service Continuations (SC), a novel OS mechanism that supports lazy dynamic migration of Internet service sessions between multi-process cooperating servers. SC provides a server application with a simple and easy to use abstraction to migrate the service state along with the serviced connection to another server, at any point during service. SC does not force synchronization (inter-process at the server, or client-server) and can migrate service sessions transparently to the client/server execution.

We have implemented SC in FreeBSD along with Migratory TCP (M-TCP), a connection migration protocol that we developed based on the SC model. We have presented results of an experimental evaluation which shows that SC can efficiently support dynamic migration of client sessions and can improve the end-to-end availability of an Internet service. We have successfully used SC in three real applications: the Apache web server, the PostgreSQL transactional database server, and the Icecast streaming server.

Chapter 3

Remote Healing Using Backdoors

3.1 Problem Statement

Self-healing and recoverability from events that impair the functionality of a system have recently become more and more the focus of systems research, as illustrated by the concept of *recovery-oriented computing* (ROC) [67]. This trend reflects a shift from raw performance towards intelligent, self-manageable computer systems, driven by recent industry initiatives like IBM's *Autonomic Computing* [40]. In this context, self-healing denotes the ability of a system to automatically detect and identify problems affecting its functionality and to take automated corrective actions.

From an operating system viewpoint, to support healing, a system must be capable of at least two functions: *(i)* monitoring, for detecting exceptional events (failure, damage, policy violations, etc.), and *(ii)* taking action in response to these events by recovery, repair, fault containment, etc.

Past and recent operating systems research has examined problems related to today's self-healing systems. On the monitoring side, OS-level monitors were used for run-time adaptation of an OS [79, 82]. Fault detection and containment was specifically addressed in extensible operating systems [10, 78], using clever forms of encapsulation to protect against faulty or malicious code. On the action side, the common approach is to force the crash/reboot of an operating system that develops a problem, without any attempt at saving its state. Recent designs like the K42 operating system [82] have introduced support into the OS for *replacing* an OS component dynamically. However, this involves complete re-design of the OS, and, even with it, no system we know of is capable of repair *and* recovery actions after a failure.

Despite obvious advantages like instant access to the entire system state, observing a system and taking actions to correct its behavior from within has serious limitations, for example: *(i)* it has limited effectiveness, e.g., if the machine runs out of resources or certain components fail; *(ii)* cannot perform integrity checking on a system if the integrity checking code itself is bad or corrupted; *(iii)* cannot detect intrusion from within an already compromised system; *(iv)* cannot recover state from a failed system unless checkpointed externally to some stable device, with the incurred consistency and performance problems.

In this chapter, we propose *remote healing*, a novel approach to system-level survivability and recoverability in which monitoring a system for detection of exceptional events (e.g., failure, errors, damaged state, policy violations, etc.) and *lazy* recovery/repair actions can be performed remotely from another system. The physical separation of the "healing system" from the "healed system" achieves robust detection and reaction to anomalies. Remote healing is a last resort approach to recovering the functionality of a system without proactively mirroring its state on other machines.

Key to remote healing is the ability to access the state of a computer system remotely, *without using its processors and relying on resources of its operating system.* To enable it, we propose *Backdoors* (BD), a novel system architecture that combines hardware, firmware and OS extensions to support automated observation and intervention on a computer system (through monitoring, diagnosis, state recovery and repair), *(i)* even if its capabilities have been severely compromised, and *(ii)* without overhead during its normal operation. BD relies on specialized, intelligent network interfaces for *nonintrusive* remote access to computer resources (memory, I/O devices, etc.), i.e., without involving its processor(s) or relying on its OS.

Existing "healing-from-within" approaches cannot solve problems like depletion of system resources, system-hang failures, damage or corruption to the software state of an OS subsystem. In contrast, remote healing can still ensure accurate monitoring, correct diagnosis of the problem, and can take corrective actions to bring the system back to normal operation or as close as possible. A Backdoor-based remote healing system can either extract useful state from a failed system and recover it on another healthy machine, or can perform in-place state repair towards restoring the system. In this chapter, we describe the Backdors architecture and OS support for nonintrusive monitoring, recovery of light-weight state from OS failures that render a system unusable, and repair of OS state damaged by resource exhaustion.

3.2 Background and Related Work

3.2.1 Failure Detection

A BD architecture can be used to perform monitoring of a computer system from another system to detect failures, without using the processors or relying on the OS resources of the monitored system.

Failure detection and fault isolation inside an OS have been studied in extensible operating systems like VINO [78] and SPIN [10], with the goal of recovering from bugs in extension code. Self-monitoring has been used to adapt OS behavior for increased performance [79] or to alleviate effects of DoS attacks on system/application [71]. While highly accurate because it has access to the whole state of a system, its reliance on resources of the same system makes self-monitoring useless in detecting and reacting to system-wide failures that make it impossible to report the fault to an external observer. The widely-used alternative is to assess liveness of a system through external monitoring from another machine, using ping/heartbeat messages sent over a network.

In external monitoring, the monitoring traffic (e.g., periodic ping/reply packets) is usually carried "in-band," over the same physical network and through the same (TCP/IP) protocol stack used for regular data transfers. This has several drawbacks: *(i)* it may generate false positives, i.e., it declares the target system dead when it is actually not, e.g., if the system is overloaded or under DoS attack; *(ii)* it can offer no immediate information on what happened in case of a crash or deadlock of the OS; *(iii)* it generates overhead on the target system, increasing with the ping frequency and/or the volume of data collected from the system.

In contrast, a BD architecture *(i)* can perform external monitoring of a target system without relying on its protocol stack or OS, thereby eliminating false positives specific to TCP/IP heart-beating techniques, *(ii)* does not incur any CPU overhead on the target machine for monitoring, *(iii)* enables a remote system to perform automated post-mortem inspection of a crashed system.

Recently, Fetzer has proposed the *timed-perfect failure detector*, a distributed protocol that eliminates false positives in detection of computer crash failures [35] . The protocol relies on custom hardware watchdogs to force-crash a system before it is (wrongly) suspected to have failed by a higher-level unreliable failure detection protocol. A timed-perfect detector is fairly expensive to implement, requiring three independent machines to detect failure of a participant.

A large body of theoretical work exists in the area of efficient failure detectors in distributed systems [23, 39, 36]. In using BD for external monitoring we build on theoretical results on the guaranteed accuracy of unreliable failure detectors under specified deadline constraints for detection [39]. A BD provides a trade-off between practical and highly accurate failure detection, through provably negligible probability of false positives and reliable enforcement of a fail-stop model, without requiring complex detection protocols.

3.2.2 OS Reliability

A BD architecture can be used to extract light-weight OS and application state from the memory of a dead system (after an OS hang, crash, deadlock, etc.) and recover it on another healthy system.

Traditional fault-tolerant systems like Tandem [9] rely on hardware and/or software redundancy to mask component failures. The high cost of hardware and maintenance practically prohibits cost-effective use of such machines. BD does not provide component or OS fault tolerance, but offers a cost-effective and light-weight solution using off-the-shelf components for recovery of critical state from a general-purpose OS after a failure.

Nonvolatile memory has been used to preserve select system/application state across crashes and reboots [7, 6]. In the Recovery Box [7], an OS-controlled NVRAM region is used to store system state and retrieve it after crash/reboot. The approach relies on transactional semantics to ensure recovery of server applications with remote client sessions. Recovery is not transparent to clients (which must reconnect and resubmit their requests) and does not support multiple communicating processes. In contrast, BD provides OS support for recovery of client sessions without involving the client OS/application in recovery, and supports recovery from multiple server node failures.

In the Rio reliable file cache [25], the file system cache is protected against corruption during a crash through software fault isolation techniques. The contents of the cache is preserved across the crash and subsequent reboot, and used to warm-reboot the file system, eliminating potential file system inconsistencies due to the crash.

The Rio idea has been further expanded on as a *reliable memory subsystem* used by applications and libraries. The Vista transaction library [56], built on top of Rio, provides suport for lightweight atomic and durable transactions for in-memory databases. Vista reduces the transaction overhead by eliminating costly file-system reliability operations and redo logs kept on disk. Discount Checking is a user-level checkpointing system built on top of Vista and Rio which exploits the persistent memory abstraction with transactional semantics provided by Vista to maintain in-memory checkpoints with low overhead. Essential to the good performance of these systems is that the recovery state is maintained in memory.

When used for recovery, Backdoors takes a similar approach to Rio, Vista and Discount Checking systems in using the system memory as storage for recovery data, for fast and transparent recovery. However, in contrast to Rio and its follow-ons, BD provides a generic architecture for *remote* failure detection and extraction of lightweight state after a failure, regardless of which OS subsystem manages this state. For example, a BD-based system can recover OS-level communication state associated with an application, while Rio limits the recovery support to data maintained and accessed through the buffer cache. Unlike Rio, the current BD implementation does not provide fault isolation, which makes it vulnerable to memory corruption in a crash.

In [15], a virtual machine monitor has been used to intercept and back up the entire state of a system on another machine for tolerating failures, at the expense of dedicating full machines and with fairly large performance penalties. Because they rely on full-state replication, such primary-backup approaches can only deal with hardware or power failures, since OS failures or crashes would equally impair the backup machine.

Hot-swapping of whole OS subsystems is supported by OS designs like K42 [82]. While attractive for its ability to preserve live OS state across a swap, hot-swapping requires structural OS changes and is ineffective on a dead system with no cycles available to execute even the hot-swapping code.

Nooks [94] is a software system that uses code interposition and virtual memory techniques to sandbox faulty kernel loadable modules. Because Nooks focuses on memory protection, it can only detect faults if they occur in extensions and involve faulty memory accesses. BD is orthogonal to Nooks, by detecting and recovering system state from system-hang failures, regardless of their place of occurrence in system code. Unlike Nooks, BD can handle failures triggered by other factors than system software (operator errors, hardware faults).

To our best knowledge, Backdoors is the first system that leverages memory-to-memory communication and intelligent NICs to perform automated nonintrusive remote monitoring and intervention on a failed system for extraction of useful state or in-place repair. Previous work done in the 1980's on DEC's Titan system [65] has used custom hardware (memory controllers equipped with Ethernet interfaces) to perform remote read/write memory operations for remote debugging and software/data patching without rebooting the kernel [61].

3.2.3 Reliable Internet Services

The case study that we describe in Section 3.6.7 demonstrates how Backdoors can be used for recovery from failures of server nodes implementing an Internet service. Specifically, we tackle the difficult problem of failure transparency for *live* client sessions of complex Internet services with transactional execute-once semantics, where components of the service are distributed on multiple machines (multi-tiered architectures). Failure transparency means that the client application/OS are not involved in recovery and that recovery does not affect the client-perceived performance.

What makes the problem difficult is the highly dynamic state in such systems (involving one or more server processes communicating over TCP/IP and IPC channels), the interactive clients (humans) that do not tolerate disruption, performance degradation, or loss of transactions with a critical service, and the strict requirement of exactly-once semantics of critical services (e-commerce, banking, auction systems, etc.). Previous recovery solutions, e.g., [103, 80, 2, 101, 51, 60, 102] are either not directly applicable to complex Internet services or have serious limitations in scope. Their common drawback, with the exception of [103], is the intrusiveness during the failure-free execution of the system for which they provide recovery support.

Zhou et al. have used virtual memory-mapped communication to mirror the address space of a process on other nodes in a computer cluster [103]. This makes user-level checkpointing and failover fast, compared to using disks as stable storage. However, since their system concentrates only on checkpointing user-level state, without considering the state of active communication channels, it cannot be used for failover of Internet services. TCP wrapping was proposed by Alvisi et al. to mask the failure and restart of a server with open connections [2]. However, its use of heavy-weight single-process checkpointing for recovery makes it impractical for Internet services. Fine-grained failover using connection migration was used by Snoeren et al. in a cluster-based HTTP server [80]. The scheme is limited to static HTTP transfers from single-process servers, requires the transport layer to be aware of HTTP, and relies on massive broadcasts of recovery state inside the cluster. Primary-backup schemes have been used to build fault-tolerant TCP servers through mirroring their communication and computation state on another machine through active remote logging (by Zagorodnov et al. [101]) or passive traffic tapping at the link-layer (by Mishra et al. [60] and Koch et al. [51]). These schemes require fully-dedicated nodes as backups and use interposition techniques that affect the performance of failure-free execution. In addition, they do not tolerate loss of the backup unless some form of stable logging is used [101]. To reduce the overhead of primary-backup approaches, Zhang et al. have proposed shadowing incomplete state of TCP connections on alternate cluster nodes, not necesarily dedicated [102]. However, their scheme requires a specialized front-end node to be involved in both state shadowing and recovery, and relies on observing communication patterns in the application to infer the complete state of a connection. When not limited to static transfer from web servers, the system requires the server application to perform consistency checks during recovery to avoid generating inconsistent results. Overall, none of the above schemes has been shown to be applicable to complex, multi-tier, transaction-oriented Internet services.

In contrast to the above approaches, we demonstrate that a BD-based system can provide both accurate failure detection and fast recovery, it is light-weight and nonintrusive, it is application independent, and can be used with complex, multi-tier cluster-based Internet services.

3.3 The Backdoor Architecture

Backdoors (BD) [88] is a novel system architecture we have designed to enable remote healing of computer systems. The central idea in BD is to combine hardware and software mechanisms to support highly accurate monitoring and effective healing actions on a computer, even in the presence of failures that make its operating system unavailable. The components of Backdoors are:

- a system architecture that enables nonintrusive access to in-memory software state of a system;
- OS extensions for monitoring system liveness;
- OS extensions for performing healing actions (recovery or repair) on in-memory data structures, after a failure.

In Backdoors, a computer is equipped with a "backdoor" - a programmable NIC placed on the system I/O bus and which is not controlled by the OS except for its initialization. The main function of a Backdoor NIC is to provide a path for access to resources of its host computer (memory, I/O devices, etc.), *without using its processors and relying on its OS*. This powerful capability makes Backdoors *(i)* nonintrusive to the

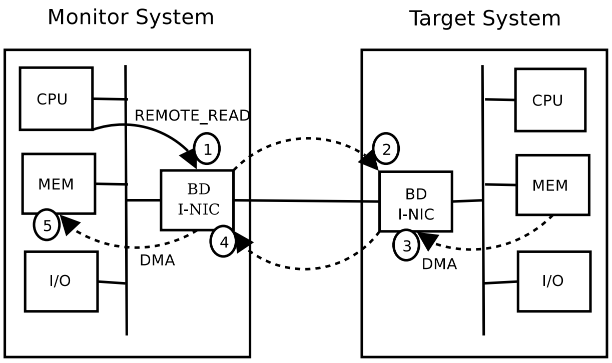


Figure 3.1: *A monitor-target pair in the Backdoor remote healing architecture. A monitor can access resources of the target system (memory, I/O devices) through the backdoor I-NICs, without using its CPU.*

system activity during its normal operation, and *(ii)* robust to OS failures that make the rest of the system unusable.

Backdoor NICs are connected through a low-latency interconnect and run a specialized firmware that does not involve processors of a remote host when performing a communication operation with it. Figure 3.1 shows the basic remote healing configuration with BD, where the system on the left acts as the monitor (M) for the target system (T) on the right. In such a M-T communicating pair: *(i)* M's CPU can initiate an operation in the T's NIC; *(ii)* T's NIC performs access operations to the local memory without using T's CPU.

To support remote healing, a Backdoor NIC must implement at least read and write access operations (local and remote). The remote read/write functionality is similar to that of remote DMA (RDMA), a communication primitive that allows a machine to access the memory of another machine for reading and writing while bypassing its processor(s). RDMA primitives are included in industrial standards [31, 1] and are implemented by specialized controllers like [58].

Figure 3.1 shows an example of a remote read operation that involves the following steps: (1) The CPU on M initiates a protocol between the local and remote NICs for transfer (read) of a remote buffer from T's memory; (2) M's NIC requests the data from T's NIC; (3-4) T's NIC performs a DMA operation to retrieve the data and send it to M's NIC; (5) M's NIC performs a DMA to place the data in local memory. Note that neither of the two CPUs are involved in the actual data transfer, while only M's CPU is involved in *initiating* the transfer. To enable remote memory access, T's OS performs a one-time initialization which *registers* with the NIC non-pageable regions of system (kernel) memory. Registration enables access control and remote memory addressing using virtual addresses by loading virtual-to-physical address mappings into the NIC.

Backdoors takes a completely novel approach in using programmable NICs and remote memory communication to support remote nonintrusive monitoring and healing operations (recovery/repair) on an impaired system. However, the simple remote access via a backdoor interface is not sufficient to heal a system. To support complex remote healing operations with BD, the OS must be extended with interfaces for remote access to OS and application state. Extensions enable access by remote BD NICs to regions of OS/application memory that hold critical state or to I/O devices of the target system. In particular, extensions may involve data structures specifically designed to support extraction and recovery of live state from the system. To ensure atomic access to the state of a live system, BD provides *remote OS locking* operations.

We next describe the OS extensions for remote healing with Backdoors. We present the design of the Sensor Box, an OS abstraction for monitoring the state of a computer system. We then describe the design of OS support for recovery of critical state from a machine affected by a system-hang failure and for repair of damaged OS state.

3.4 Sensor Box: An OS Mechanism for Nonintrusive Monitoring

We impose two major constraints on the design of the monitoring function. First, it must not introduce overhead on the monitor system M. For example, this implies that M should not do a remote dump of the whole memory of T in order to examinte its state. A low-overhad monitoring mechanism would allow nondedicated machines to be used as monitors, e.g., nodes in a computer cluster may both perform their tasks and monitor each other.

Sensor type	*Limiting threshold (L)*	*Trigger condition*
Progress	Update deadline	V did not change within L time units
Level	Max (min) value of a resource	V > L (V < L)
Pressure	Max number of events	V > L

Table 3.1: *Types of sensors in a Sensor Box. Each type defines the semantics of the threshold L and how an exceptional event is detected based on the sensor value V.*

Second, the (M, T) pairs must be only loosely coupled: *(i)* Observing T must not directly involve it. Since T might be dead, M must not rely on remote execution for fetching observed state from it, i.e., observation of T's state by M must essentially translate into a 0-cycle operation on T. *(ii)* T must not interact with M (in general, it may be oblivious to its existence). In particular, T should not freeze its execution in order for M to get a consistent view of its state. *(iii)* Participation of T in monitoring must be voluntary, by local state introspection and reporting.

3.4.1 Failure Detection with Sensor Box

To achieve these goals, we introduce an OS mechanism called *Sensor Box* (SB) that allows M to observe the liveness or health of T by enabling a loose producer-consumer relationship between the two systems. Entities (e.g., OS subsystems) running on T are the SB producers. At the other end, M fetches (reads) T's SB periodically through the backdoor and uses it to assess T's health.

A *Sensor Box* (SB) is a structured collection of records called *sensors*, allocated in the OS memory of the target system. A sensor is a tuple $< ID, C, L, V >$, where ID is a unique identifier, C is a class of sensors it belongs to, L is a limiting threshold that depends on the sensor class, and V is a scalar (the actual sensor). A monitored entity defines the limit L and is responsible for updating the sensor value V.

We define three classes of sensors based on their functionality and detection properties (Table 3.1):

(i) Progress sensors are monotonically increasing counters that indicate the "liveness" of T. The monitored entity updates V and defines a deadline L for updates. If V does not change for L time units, the monitor may interpret the stall as a failure. Examples

of progress sensors are: number of interrupts raised by a hardware clock with the clock time period as the deadline, number of context switches in the system with the time quantum as the deadline, etc.

(ii) Level sensors are counters that track resource utilization on T. If the sensor value exceeds the threshold L, an exceptional event is detected by the monitor. Examples of level sensors are: number of processes in a system with a limit on the maximum number of processes, number of wired pages in system memory with a limit on the maximum number of such pages, etc.

(iii) Pressure sensors are counters incremented on T upon occurrence of certain events. The monitor detects an exceptional condition if the number of times the event occurs exceeds the threshold L. Examples of pressure sensors are: the system could not allocate memory, the file descriptor table in the system is full, etc.

Upon creating a sensor, a monitored entity specifies its identifier ID, its type C, and the limiting threshold L. The monitored entity must cooperate with a monitor by modifying V for its associated sensor(s). This establishes a *contract* between monitor and monitored. The monitored entity commits itself to updating V, according to the type of the sensor, by increasing its value at intervals smaller than L to signal its liveness (progress sensors), by tracking the value of a measured quantity (level sensors), or by incrementing its value if it detects an exceptional condition (pressure sensors). The monitor commits itself to retrieving the sensor and comparing V and L. It detects an exceptional event if V has not been updated within L time units for progress sensors, and if V exceeds L for level and pressure sensors.

The SB is accessed by T (locally), and by M (remotely, through the BD) using a simple interface:

```
sensor = new_sensor(ID, C, L)
set_sensor_value(sensor, value)
sb_view = fetch_sb(nodeID)
```

where ID is the sensor identifier, C is the class (progress, level, or pressure), and L is the limiting value. On T, `new_sensor()` creates a new sensor and `set_sensor_value()`

is used to update its counter value. On M, `fetch_sb()` creates a local copy `sb_view` of the SB of a monitored node identified by `nodeID`. In Section 3.9 we show that an SB is light-weight both for local and remote access through the BD, and that OS-level progress counters can detect hang failures of the target OS both fast and reliably.

3.4.2 Accuracy of Failure Detection

In a BD architecture, use of remote read memory access for monitoring makes the hardware providing it (backdoor NICs, physical links) a single point of failure. Even with redundant access paths, accurate failure detection can still be undermined by two events: *(i)* catastrophic failure in the path reaching the monitored node; *(ii)* random message loss in the underlying network.

Access path failures may lead to false positives in failure detection. In particular, when using SB progress sensors for failure detection, it is impossible to distinguish path failure from a real crash of the target node based only on reading values of progress sensors in the local view of the monitored SB: in both cases, sensor values would not change. To solve this problem, we take advantage of link-layer failure detection mechanisms provided by existing hardware [63]. A monitor periodically probes the link-layer to check the healthiness of the remote access path. If the probe fails, the monitoring function is delegated to a backup monitor with a healthy link to the target system.

Message loss while reading the SB may have the same effect as a link failure: if a monitor sees no change in progress sensors values in the target's SB because a remote read request was lost, it may declare the target node dead. To cope with message loss on BD NIC links, the monitor must *adapt the sampling rate* R of the remote SB as a function of the path loss characteristics, under deadline constraints imposed by monitored entities. An efficient monitor must detect failures within a prescribed deadline and with a prescribed accuracy, given a known probability of message loss [23, 39].

In the following, we adopt the notation of Gupta et al. [39]. We assume a known bound p_{ml} on the probability of message loss in the underlying network. Then, the sampling rate is constrained by two parameters: *(i)* the minimum deadline for failure detection across all monitored progress sensors, denoted by $\mathcal{T}$, and *(ii)* the prescribed accuracy of detection, as the maximum probability with which a healthy node can be wrongly detected to have failed by a monitor within $\mathcal{T}$ time units, denoted by $PM(\mathcal{T})$. Under ideal conditions (perfect links, $p_{ml} = 0$), a monitor would not need to sample more often than $\mathcal{T}$.

Based on the results in [39], if $\mathcal{T} \gg RTT$, where RTT is the worst case round-trip time between two nodes (monitor and target in our case), we can satisfy the above two constraints on $\mathcal{T}$ and $PM(\mathcal{T})$ if a monitor samples a remote SB at least $R \geq \frac{\log(PM(\mathcal{T}))}{\log(p_{ml})}$ times during a time interval $\mathcal{T}$. This formula enforces a lower bound on the sampling rate to ensure highly accurate failure detection under deadline constraints (an obvious hard upper bound is given by the time needed to fetch the remote SB).

For all practical purposes, the above condition on $\mathcal{T}$ is met by actual hardware which has low latency and is highly reliable (e.g., $RTT \sim 10\mu s$ and $p_{ml} \ll 10^{-8}$ for Myrinet [63])[1]. In addition, using Backdoors for monitoring eliminates other common sources of false positives (e.g., software network stacks), down to the physical link. This, combined with the high reliability of intelligent NIC hardware, allows for more than reasonable sampling rates. For example, suppose that $p_{ml} = 10^{-8}$ and we impose a probability $PM(\mathcal{T})$ of false positives in failure detection due to message loss of at most 10^{-8}, for a deadline $\mathcal{T} = 1$s in detecting any failure. Then $R \geq 1 + \epsilon$ for some $\epsilon > 0$ arbitrarily small. In other words, it suffices for a monitor to sample a remote SB at intervals of just under one second in order to meet a detection deadline of one second, in the presence of losses in the backdoor path, and with high probability of *not* making errors. Thus, very high rates of sampling are possible (practically limited only by the latency of fetching the remote SB), with virtually no false positives in failure detection. In Section 3.9 we will show that a monitor can sustain sampling rates on the order of milliseconds with negligible impact on the monitor node.

[1] For lack of hardware specifications, we use as a conservative upper bound for the probability of message loss the typical figure for the (much less reliable) Ethernet hardware.

3.5 Continuation Box: An OS Mechanism for Light-Weight Recovery

In this section, we introduce a recovery model and a recovery mechanism based on Backdoors that enables survival of critical software state of a computer system across system-hang failures, with low/negligible overhead during failure-free execution and fast recovery. By "critical software state" we mean *light-weight* state components residing in system memory that are needed for performing a certain task, for example: data in TCP buffers of live connections in a network server, data in dirty buffer cache blocks not synced to a network file server, application-specific data describing the point an application has reached in its computation, etc.

3.5.1 Failure and Recovery Model

We denote by *failure* the impossibility of a computer to execute any code (hang failure), or to make progress in a certain OS subsystem or application. A failure may have multiple and complex causes: *(i)* a faulty software component in the OS that leads to a system-wide freeze, e.g., a driver bug causing permanent loss of interrupts from a device, system hanging due to a deadlock error, a misplaced panic only reached under certain stress conditions, etc.; *(ii)* a wrong operator command or a misconfiguration that causes the OS to halt or crash, generate errors, or slow down under prescribed levels while under reasonable load; *(iii)* a peripheral device that ceases to respond and prevents a certain OS subsystem from executing normally or making progress in service (e.g., a faulty disk, a disconnected Ethernet cable, a faulty NIC that stops generating interrupts, etc). In such cases, we assume that the rest of the system is not impaired, e.g., a faulty component does not lock up the system bus.

The failure model is fail-stop: a failed system does not behave erratically (the failure is non-Byzantine). For failure detection purposes, each system has its own private clock. Clocks do not need to be synchronized, but their drift rates (from an arbitrary clock) must remain constant.

A Backdoor-based system that can perform recovery from failures must execute two operations: continuous monitoring for failure detection and the actual recovery.

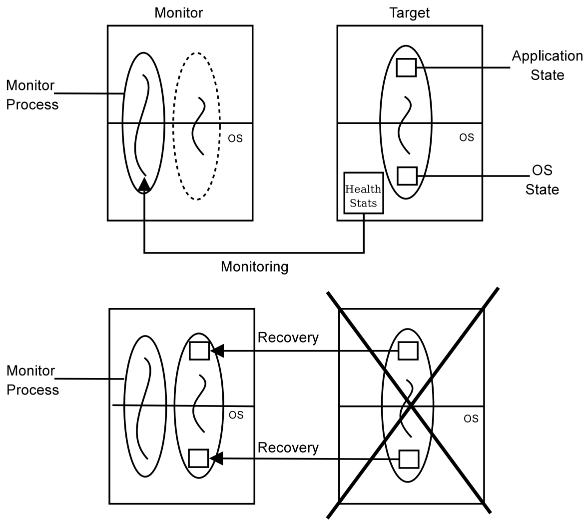


Figure 3.2: *An example of monitoring and recovery with Backdoors. (a) A monitor process on M inspects the Sensor Box in T's OS memory. (b) On detecting a failure, M extracts light-weight critical application-level and OS state and reinstates it locally.*

Monitoring. The monitoring component of the system relies on at least one other *monitor* system (denoted by M) that performs external monitoring of a *target* system (denoted by T), detects its failure, and initiates recovery actions. The monitor is *not* a dedicated machine, and the monitor-target functionality is symmetric, i.e., a target can be a monitor of another system (including its own monitor). For example, machines in a cluster of servers may mutually monitor each other. We use "monitor" and "target" only to distinguish the roles of two systems in a given peer-to-peer interaction.

An M runs a failure detection algorithm implemented by a *monitor process* using periodic remote observation of the Sensor Box in T's memory (Figure 3.2, *(a)*). When it suspects a failure, and before taking recovery actions, M *enforces* the fail-stop model by performing a remote OS locking operation that halts any OS activity on T.

Recovery. Upon detecting a failure, M takes over the lost functionality of T. The recovery action consists in extracting critical OS and application state from T's memory *after* the failure and injecting it into M's memory (Figure 3.2, *(b)*). For example, if T is an Internet server, M must re-incarnate the server-side of a TCP client connection and synchronize the two endpoints. The failover requires that M has access to the same resources as T, e.g., if T was using an external file/database server M must also have access to it.

This recovery model assumes that T's memory is available after a failure, and that T's OS does not destroy or corrupt critical state in response to a failure, i.e., T just hangs. Section 3.8 provides a detailed discussion on the general validity of this assumption and on existing mechanisms that enforce it.

To perform nonintrusive monitoring and recover from a failure that makes T unavailable, Backdoors provides the mechanisms that enable *(i)* access to the state of the system without overhead during execution (monitoring), and *(ii)* access to T even when it is dead (recovery).

3.5.2 The Continuation Box Idea

Using Backdoors for recovery relies on the insight that, in a failed machine, data structures holding "good" state are still alive in system memory while the OS is unavailable (crashed, deadlocked, hung, etc.). It is appealing then to think of a mechanism that would allow us to: *(i)* bypass the unresponsive system to access this state from an external system, *(ii)* extract it from the failed system, and *(iii)* reinstate it and continue using it on another healthy machine. While *(i)* above can be achieved using remote memory access as described in Section 3.3, to achieve *(ii)* and *(iii)*, a BD architecture must also include OS support for external manipulation of OS/application state of interest in the target system.

There are several design constraints that the OS support for recovery using BD must satisfy: *(i)* it must be light-weight, imposing only minimal overhead for recovery support operations during failure-free execution, and low cost during the recovery phase; *(ii)* it must be silent during failure-free execution, i.e., create no background traffic in the system; *(iii)* it must not require CPU cycles on the failed node for state extraction during recovery; *(iv)* if the target computer is a server, it must be transparent to client OS and applications, both during failure-free execution and recovery.

To achieve these goals, we propose a light-weight *Continuation Box* (CB), an OS abstraction that encapsulates *fine-grained* application-specific and OS state associated with a running application. The idea behind the CB abstraction is that most of the user and OS-level state maintained for an application (viewed as a collection of processes on a given system) is either redundant or soft, i.e., it is already available or can be easily re-created in the same application running on a different system. The "essential" (hard) application/OS state that distinguishes a particular instance of the application is the only component that needs to be actually recovered. The CB recovery model relies on extracting this state from the memory of a failed system *after the failure*, and reinstating it in a healthy OS/application running on another system, which can then continue the execution.

3.6 Design of a Continuation Box for Internet Servers

We illustrate the Backdoors recovery model based on the Continuation Box idea with the design of a CB that can salvage live client sessions from a failed Internet server and continue their service on another machine. Providing recovery support is particularly challenging for Internet services, which are usually structured as collections of multiple communicating processes running on multiple machines, have highly dynamic state (including that of communication channels - client TCP connections and IPC channels), and operate under heavy client loads. Designing a light-weight CB abstraction relies on the observation that most Internet services maintain well-defined, *fine-grained* state associated with each client session.

We assume an Internet service running on a cluster of machines in which multiple nodes running the same server application exist. Service failover by state extraction exploits node redundancy in terms of logical functionality. Recovery operates at the granularity of one service session: it extracts and reinstates its state on a healthy node after the failure, and assists the server application running there in resuming consistent service to the client. Any other non-specific state needed to continue the service (e.g., access to external databases, file repositories, etc.) is deemed accessible at the new node.

3.6.1 Continuation Box versus Service Continuation

We note that, when applied to cluster-based Internet servers, the Continuation Box idea involves moving state associated with a client session between two similar servers. For this reason, in the case of Internet services, a CB should provide functionality which is very similar to migration of service sessions between equivalent servers. For example, the CB must encapsulate, for a given client session, both application-specific state and OS-specific state (the state of the client-server TCP connection, and the state of IPC channels between server processes). Such functionality is already provided by our Service Continuation (SC) mechanism for lazy session migration described in Chapter 2. For this reason, it is appealing to use a Service Continuation as a Continuation Box for recovering an Internet service session from a failure.

There are, however, difficulties that prevent such a straightforward approach. They stem from key differences between *migration* of state between two healthy machines (which is what SC provides) and *extraction* of state from a failed system (which is what we want to achieve with a CB).

In the case of session migration using SC:

- The origin server is actively involved in the migration protocol in several ways:

  1. It runs a protocol stack (TCP/IP) used for transport of messages by the state transfer protocol.
  2. It implements the state transfer protocol with the destination server (interprets requests, marshals the SC state into messages, etc.).
  3. It performs computations on TCP-specific state to help the TCP connection migration protocol synchronize the destination server endpoint with the client endpoint.

- Atomicity of an SC is implemented and guaranteed by the state transfer protocol at the origin server, by mutual exclusion with the user application code and with the TCP/IP stack.

- The TCP connection migration protocol (M-TCP) involves the client-side TCP. Migration is initiated by the client TCP, which is actively involved in synchronization of the state of the connection with the destination server.

In contrast, a CB has to operate under more severe constraints, given that the server that was servicing the session has now failed. As a result, in the case of session recovery with a CB:

- The failed server cannot be involved in a state extraction protocol.
- The failed server cannot be involved in synchronization of communicaton state with the client.
- Atomicity of a CB is a problem if the failure occurs while the CB is being updated.
- Recovery must be done without changing the client TCP or OS.

The similar functionality of SC and CB as vehicles for application and OS state components means that the *logical structure* of a CB would be essentially the same as that of an SC. As a result, a similar API would be used to maintain the application state components in a CB.

However, the impossibility of involving the client and the original server in the recovery (as illustrated by the differences listed above) forces the design of a completely different state synchronization mechanism between client and server. In addition, the system must make provisions for handling the CB atomicity (or lack thereof) with respect to a failure.

3.6.2 The Structure of a Continuation Box

Like in the case of SC, we assume that the session state may span multiple communicating server processes executing work for the client. Figure 3.3 shows two such processes handling the service session of one client. The accepting process (P_1 in the figure) is the only one that communicates directly with the client through a TCP connection. Server processes communicate via byte-stream channels (IPC or TCP/IP) and may run

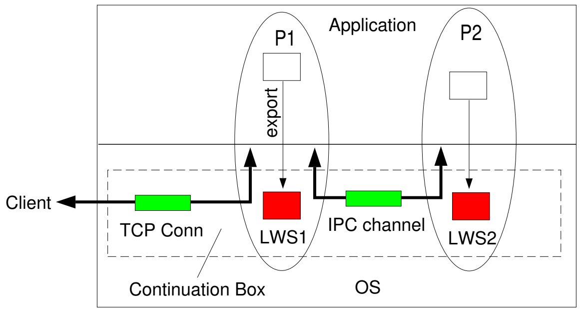


Figure 3.3: *The structure of a Continuation Box for Internet services: application-specific state and the state of communication channels.*

on multiple machines involved in servicing the session. We consider the same execution model as in the case of SC, i.e., process execution *with respect to a client* is a sequence of state intervals delimited by well-defined states starting from a reproducible initial state.

The OS of a server node maintains a per-session CB through an API that the server application uses to record *light-weight snapshots* (LWS) of session state. The OS also tracks the state of communication channels (inter-process and client-server) used by the server application. An LWS must completely describe a well-defined point reached in the ongoing service session, from which a server process can safely resume service for the client after failure. For example, for a client performing a static HTTP transfer, the web server's LWS will contain, at a minimum, the file name and the offset reached in the transfer, while the server OS will track the state of the client-server TCP connection. Figure 3.3 shows the OS-specific components and application-specific LWSs in a CB that spans the two server processes communicating over IPC channels. Note that the state of the stateful communication channels is a part of the CB.

3.6.3 The Continuation Box Recovery API

At the time a failure occurs, the server state with respect to a client may be undefined or inconsistent. Even if this state would be well-defined, we cannot rely on the OS to dispatch upcalls to user-level handlers to fetch it from the application since OS scheduling mechanisms will not work if the OS is crashed or hung.

To solve this problem, the application processes must be involved in defining their LWSs. The OS maintains CBs in the system memory and provides a minimal API that allows a server process to specify its LWS, from which a similar process on another server node can safely resume execution after the local node has failed.

The main primitives of the CB API are:

```
cb_export_state(cb, state_buffer)
cb_import_state(cb, state_buffer)
```

where `cb` is the CB object associated with a client (an OS-specific identifier) and `state_buffer` is an application memory buffer holding the LWS of the client session state in a process.

During service, server processes periodically call `cb_export_state` to save LWSs to a CB. The LWS contents are opaque to the OS and to the CB extraction protocol. Figure 3.3 shows how processes make API calls to save continuation points in the CB. The calls are made independently by each process, i.e., processes are not forced to synchronize.

After a failure, the OS of a healthy node in the cluster extracts the CB from the failed node using remote access through Backdoors. Server processes running on the new node call `cb_import_state` to retrieve their LSWs for the client session (only once) and then use them to resume service to client.

As discussed in the previous section, the CB and SC share similarities in their structure and API. Likewise, because the CB does not force processes to synchronize when exporting their LWSs, the OS of the new server must synchronize the endpoints of communication channels using the techniques devised for SC in Section 2.4.4.

The CB synchronization mechanism, similarly to SC synchronization, relies on a limited form of log-based execution replay of a process starting from a previous LWS. One issue with this approach is that a deterministic execution that leads to a crash may also crash during replay. For example, suppose a server process makes a system call which has a buggy implementation that crashes the OS when invoked with certain arguments. Then, during recovery on another healthy machine, the call will be issued again with the same arguments, repeating the crash. We note however that this is a problem only with application-induced faults of the kind we described, i.e., buggy system calls. Other software faults, e.g., transient faults like races or deadlocks that occur due to nondeterministic events in the OS will not be reproduced by re-execution. Moreover, the problem of buggy system calls can be easily addressed if the node that extracts the CB from the failed system and performs recovery runs a different OS implementation. This *design diversity* approach is a well-known technique in building dependable systems. In our case, using different OS implementations is possible because a CB encapsulates only the *logical* state of standard protocols (TCP) and OS abstractions (byte-stream IPC channels), which does not depend on any particular implementation.

3.6.4 Continuation Box Atomicity Issues

As mentioned before, atomicity of changes to the CB with respect to a system failure is a problem we have to address in order to ensure recovery of *all* sessions on a node. Note however that if a system does not provide CB atomicity, there exists a worst-case upper bound on the number of sessions that cannot be recovered. This bound is equal to the number of hardware threads running on the node which were (potentially) executing a `cb_export_state` call and failed to complete the export of an atomic LWS to some CB. Since export calls can be tracked, it is fairly easy to mark these CBs as *changing* (therefore inconsistent in the event of a failure) and simply discard them at the time of extraction. As a result, only a few out of the hundreds or thousands of client sessions serviced by a node might be lost due to nonatomic LWS updates.

There are two solutions to the problem of nonatomic LWS updates. The first solution is to *prevent* the occurence of nonatomic updates with respect to a software OS failure. This approach relies on a simple observation: if the code that updates an LWS is trusted not to crash the system, then it would be enough, while `cb_export_state` is executing, to block execution of any other system code that may potentially lead to a failure. Since the `cb_export_state` implementation is under our control and is fairly simple, we can make it bug-free and trust it not to cause a system failure. On the other hand, blocking *any* other system code from executing is a brute-force approach that may incur high overhead. For this, we would have to block execution of any interrupt and system call on the system while a single `cb_export_state` is executing in some thread. Blocking interrupts is needed to prevent interrupt handlers (e.g., code in a buggy network driver ISR) from executing and crashing the system. Blocking system calls is required on SMP or hardware multi-threaded (SMT) systems, to prevent another thread of control from entering the kernel executing a potentially buggy system call that would crash or hang the system.

While prevention of nonatomic updates is possible to achieve by disabling interrupts and using mutual exclusion in entering the kernel between `cb_export_state` and all other system calls, it would reduce the degree of kernel concurrency, particularly in kernels with fine-grained locking. Note also that, in an SMP system, this solution is not complete since not all interrupts can be disabled. For example, nonmaskable and interprocessor interrupts will still be serviced on a processor executing `cb_export_state`. This introduces the additional requirement that the corresponding interrupt handlers should be failure-free, e.g., they do not hang in infinite loops, do not leave interrupts disabled, etc.

The second approach, and the one we favor, is to *avoid* rather than prevent nonatomic updates. We rely on the observation that because an LWS is kept in memory, a failure can affect it at most at the level of a one-word memory write. This suggests that since one-word memory writes are atomic with respect to any failure of the OS (i.e., an OS failure cannot leave a memory word in an inconsistent state), a solution to the atomicity problem could exploit atomic memory writes.

Then, to avoid a nonatomic update, a CB will simply use two buffers per process, one storing the current (stable) LWS and one for a tentative LWS. The `cb_export_state` call uses the tentative buffer to write the new LWS. When done, it sets a pointer (with an atomic one-word write) in the CB to the most recent LWS. An alternative is to use a sequence number to tag the most recent (stable) LWS. For example, Figure 3.4 shows a scenario in which a failure occurs during a `cb_export_state` call, while a process is

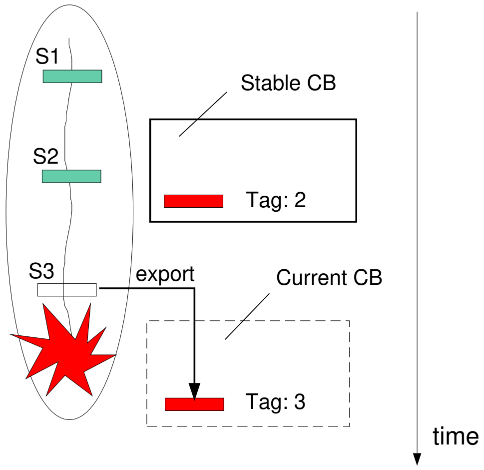


Figure 3.4: *A CB is made atomic with respect to a failure by tagging it with a one-word memory write after an export call has completed, and by keeping a previous stable CB.*

updating the current LWS (with a tentative tag 3). The OS keeps a previous stable LWS (tagged 2) that can be always used to recover. The `cb_export_state` atomically updates the tag with a one-word memory write after it has updated the current LWS, at which point the previous LWS can be safely discarded. During recovery, the CB extraction procedure will make sure it uses the LWS with the highest sequence number.

3.6.5 Synchronization of Client and Server TCP

In this section, we describe the mechanism used to synchronize the client endpoint of the client-server TCP connection with a new connection endpoint reincarnated at a new node. Without loss of generality, we assume that the failover is completely transparent using a simple form of IP address takeover from the failed node to the recovery node. In our prototype system, we have implemented the IP takeover between two cluster nodes using the well-known method of gratuitous ARP replies sent by the node which takes over the client sessions from a failed node.

The synchronization mechanism uses the log-based rollback recovery techniques described in Section 2.4.4 to restore the session state in the accepting process of the new server *with respect to a client* by replaying all communication of the process with the client using logs of TCP communication activity (send and receive buffers). The replay also synchronizes the server endpoint of the TCP connection with the client endpoint. However, to restore the state of the server TCP endpoint, we cannot use any knowledge of the client's state, since recovery must not involve the client. The state of the connection at the failed node is the only information we can use to resume communication with the client.

The state of the server endpoint of the TCP connection can be classified in three types of components:

- **hard state components**, which must be transferred and re-instated at the new node: the state of the TCP finite-state-machine, client IP address, port numbers, (data) sequence numbers, acknowledgement numbers, logs of data received and acknowledged by server TCP since the last LWS was taken, data sent before the last LWS and unacknowledged by client TCP, options negotiated with the client TCP during connection setup (window scale options, MSS), etc.

- **state components sensitive to local node failures**, for which the decision to keep or discard depends on whether the node failure may have affected them or not: RTT estimator state, congestion window value.

- **soft state components**, which can be discarded and reconstructed by the server TCP once the traffic on the new connection is resumed: TCP timer values, measured RTT.

It is important to note that for state components which are sensitive to node failures (the RTT estimator and congestion window), there is no clear-cut classification as hard or soft state. That is, the decision of keeping versus discarding certain state variables in this case depends on trade-offs in design with respect to the *likely* cause of the failure. For example, a TCP retransmission that alters the congestion window may have been caused either by actual congestion in the Internet, or by a local node failure (a crashed network driver, a faulty NIC, etc.). While it is impossible to decide whether the congestion window value is good or bad, further examination of TCP state helps decide whether it can be preserved or discarded. We will address this class of state components separately.

We next detail the mechanism used by the OS of the new node to extract the state of the TCP connection from the failed node using Backdoors. Since the state components are scattered in memory, the local OS needs multiple remote read operations to fetch the whole state pertaining to a TCP connection into the local memory. One important implementation decision is when to allocate the socket and TCP/IP protocol data structures associated with the connection on the new node. The two options are: *(i)* allocate eagerly (early) and fetch state components in-place, or *(ii)* allocate lazily, after all state has been fetched.

We decided against eager allocation for several reasons, some nonobvious: First, eager allocation would mean that the new (embryonic) socket must be placed on a server socket queue, while its state is inconsistent. Second, there is a race between a `close` of the server socket (e.g., if the server application dies) and fetching state components for an embryonic socket. A closed server socket would take down with it an embryonic connection before all its state has been extracted. For this reason, every time the OS fetches a state component from the failed node, a lookup for the new connection is needed to see if it is still alive. On a server with tens of thousands of client connections, these lookups add unnecessary overhead to CB state extraction. Third, direct, in-place remote read operations may destroy fields used by local TCP/IP for connection lookups, or may change the protocol state in an uncontrolled fashion.

For these reasons, there exist a strict order of operations to be followed for TCP state extraction and reinstantiation for one connection:

1. Fetch the remote TCP control block.

2. Fetch the LSW, the TCP snapshot and the socket snapshot recorded during the last `cb_export_state` call.

3. Determine the portions of send and receive data buffers to be fetched, according to the log-based replay synchronization logic. Fetch data buffers, possibly using multiple remote reads to traverse and fetch remote chains of contiguous buffers (mbufs).

4. Allocate the new socket, then reinstate TCP and socket state. The local TCP control block is filled in using the extracted TCP control block from the failed node and the TCP state snapshot.

We use the following notations for the values of byte sequence numbers maintained by the TCP protocol at the server: una is the sequence number of the first unacknowledged byte that was sent to the client, nxt is the sequence number of the next byte expected to be received, ack is the sequence number of the first received byte not yet acknowledged to the client TCP, and MAX is the sequence number of the first unsent byte in the send buffer. Similarly to the synchronization scheme in Section 2.5.3, lsr and lsw are the sequence numbers of the next byte to be read, respectively to be written *by the server application* at the moment the `cb_export_state` was called to record the snapshot. Next, we describe the details of the above steps, with emphasis on TCP state components:

1. The portion of the TCP send buffer to extract and number of sent bytes to be discarded during replay are determined as follows:

```
if (MAX <= lsw) {          /* snapshot is after last byte sent */
  snd_buffer = [una, lsw) /* fetch unack'ed bytes before lsw */
  snd_bytes_discard = 0
}
else {                     /* snapshot is before last byte sent */
  snd_buffer = [una, MAX) /* fetch all unack'ed bytes */
  snd_bytes_discard = MAX - lsw /* discard bytes after lsw */
}
```

The above pseudocode has the following justification:

If the snapshot lies after the last byte sent out, then all unacknowledged bytes before the snapshot must be fetched and re-sent from the new node, since they will not be re-generated. We use the replay to regenerate bytes after lsw. Accordingly, no bytes will be discarded during replay.

Otherwise, all unacknowleged bytes up to MAX must be fetched and re-sent from the new node. This is equivalent to preserving the send window at the new node, i.e., as we will show later, the values of una and MAX must be preserved. Because bytes sent out after lws (through MAX) will be regenerated at the new server, they will have to be discarded during replay.

Note that when $MAX > lsw$ (the `else` case above), it might be tempting to rewind the value of MAX down to lsw, transfer only the `[una, lsw)` portion of the buffer, and let the replay regenerate bytes beyond lsw. However, this would be wrong, as it is inconsistent with the client's view of the server's send window, and may lead to a deadlock of client and server TCPs.

To illustrate this, consider the following scenario: After TCP traffic resumes from the new server, the client sends a legitimate ACK with sequence number $ACK\_seqnum > MAX$ (since MAX was wound down from its correct value). The server TCP discards it since it falls outside its send window, and tries to send a full data segment with unacknowledged data. The hope is that this segment will cover a sequence number range large enough to recover the original value of MAX. However, the outgoing segment may be trimmed and end-up carrying no data, e.g., if the congestion window is small. The segment will elicit an ACK from the client with the same ACK_seqnum, and the cycle will repeat, leading to deadlock and a ping-pong ACK storm between client and server. The client makes no progress since it receives no data and the server makes no progress since it cannot accept client's ACKs.

2. The portion of the TCP receive buffer to extract is computed as `rcv_buffer = [lsr, ack)`, since if there are bytes received *and* acknowledged after the last snapshot, the client can no longer re-send them.

   Note that this policy attempts to minimize the size of the extracted receive buffer, relying on the client TCP to re-send the unacknowledged portion of the buffer. An alternative greedy policy is to fetch the whole `[lsr, nxt)` portion of the buffer.

3. The LSW and TCP-specific state components are extracted from the failed node,

and a new TCP control block and socket are allocated for the new server endpoint of the connection.

4. The state of the TCP finite-state-machine for the connection is restored from the TCP snapshot since it might be subject to changes triggered by user-level actions (close, shutdown) during replay. Similarly, user-set TCP options (including the MSS, which is determined at connection setup but can be controlled from user-level) are restored from the snapshot. The socket state will be restored from the socket snapshot prior to its accept by the server application.

5. The **hard TCP state components** are re-instated using information from the TCP control block extracted from the failed node. While certain variables are adjusted to ensure correct synchronization, we say that those which are copied verbatim from the old TCP control block are *preserved*:

   – The initial send/receive sequence numbers on the connection are preserved.

   – The una and MAX values are preserved, since they reflect the send window, which must be kept consistent with the client's view of the received data.

   – The sequence number of the next byte to be sent by server TCP is set to una. On the first `tcp_output()` call, this will force an immediate retransmission of data from `snd_buffer`, starting with the first byte unacknowledged at the failed node.

   – The sequence number of next byte expected to be received (nxt) is set to $max(lsr, ack)$, since only bytes up to ack were fetched from the failed node.

   – The sequence number of the last acknowledgement sent (ack) is preserved.

   – The value of the receive window advertised to the client, the receive window advertised by the client, and its maximum value ever seen are preserved.

   Note that the server-side receive window may open up during replay but in no case will the client see a shrinking receive window, so the server behavior will appear as consistent to the client.

– The window scaling variables negotiated with the client at connection setup are preserved.

– The last timestamp received and the time when it was recorded (the timestamp "age") are preserved. The timestamp age is adjusted to the local time, using the difference between the clock (time ticks) of the new and that of the failed node. The clock value is a counter of hardware clock ticks kept in system memory and is obtained via remote read through Backdoors.

6. The **TCP state components sensitive to network failures** (congestion window and RTT estimator) are handled in a special way. We describe the conditions and assumptions under which some of the variables describing them can be preserved.

   – The congestion control variables, i.e., the congestion window `cwnd` and the slow-start threshold `ssthresh`, are preserved *unless* the TCP of the failed node was performing retransmissions.

   Note that in general we do not know whether a reduction in `cwnd` is caused by the internetwork (e.g., due to real congestion in the Internet) or by the failed node itself (e.g., due to a network interface/driver fault, a disconnected network cable, etc.). To be conservative, we preserve old values assuming they reflect the state of the network.

   However, if the TCP of the failed node was doing retransmissions at the time of failure, we favor the "local" failure hypothesis. We discard the current `cwnd/ssthres` values and restore their values from their previous values, saved by TCP at the time of the most recent first retransmission. This approach eliminates "bad" retransmits due to a bad node (not a bad internet) that would otherwise destroy a good `cwnd` and kill throughput. This allows us to preserve a good throughput on the connection if the retransmission was due to a failure internal to the server node, while not violating the TCP semantics.

   – The RTT estimator *estimated values* (smoothed RTT, smoothed RTT variance, and current retransmission timer) are preserved, while the *estimator state* variables used in computing it (the current RTT, the retransmission backoff, etc.) are discarded.

The reason for this design decision is that the current estimator state might be corrupted by a node failure (e.g., a malfunctioning network interface or a bad driver), since it consists of variables like the measured RTT, which are highly sensitive to recent events. On the other hand, the TCP's smoothed RTT estimator estimated values filter out recent events and better account for the history. They are less sensitive, for example, to an immediate retransmission triggered by a local failure.

7. To complete the failover, the socket is placed on the accept queue of the server socket at the new node, its state is restored from the socket snapshot, and output is forced on the new connection to restart TCP traffic.

The CB model of session recovery incurs low-overhead during failure-free execution and enables fast recovery (as we show in Section 3.9) due to the following features:

- It maintains critical session state in system memory, allowing fast extraction after a failure;
- It enables zero-copy logging of in-kernel communication state using data buffers maintained by the OS;
- It enables zero-copy implementations of export by mapping user-level LWS buffers into OS memory;
- It does not enforce any coordination between server processes or between server and client;
- It is transparent to the client OS/applications, i.e., it does not require client OS/applications to change, nor does it involve them in recovery[2].

[2]The exception is automatic retransmission by the client TCP of packets lost during the failover.

3.6.6 Example: Session Recovery in a Web Server

We illustrate the use of the CB API with excerpts from a recovery-enabled Apache [3] web server. We limit the example to code for a static transfer, and do not include server-specific state nonessential to resumption of the *byte transfer* (SSL keys, cookies, etc.).

Apache is structured in one controller process and a pool of worker processes. The controller initializes the server and creates the listening socket `sd`. A worker starts in function `child_main` and calls `ap_accept` to wait for an incoming client connection. `ap_accept` returns the accepted connection `csd` and fills a request structure `r` with details of the request. To enable session recovery in case of failure, the worker maintains a LWS represented by:

```
struct snap {
  size_t off; /* offset in the stream */
  char *req;  /* request received from client */
  int reqlen; /* request length */
};
```

On accepting a connection, the worker creates a CB and attempts to import state from it. The `cb_import` call returns a boolean value, `TRUE` if there exists a LWS in the CB (which means this is a recovered session that was extracted by the OS from another server) and `FALSE` if the session is an ordinary (new) one. This sets a `recovered` flag in the request structure to further distinguish the two cases:

```
static void child_main(int child_num_arg) {
  csd = ap_accept(sd, &sa_client, &clen);
  r->cb = cb_create(csd);
  r->recovered = cb_import_state(r->cb, snap);
  r->snap = snap;
```

For a recovered session, the request is retrieved from the LWS, otherwise it is read from the socket, then processing continues on the normal path:

```
  if (r->recovered) /* recovered */
   memcpy(r->request,r->snap.req,r->snap.reqlen);
```

```
  else {            /* new */
    ap_read_request(r);
    memcpy(r->snap.req, r->request, r->reqlen);
    r->snap.off = 0;
    r->snap.reqlen = r->reqlen;
    cb_export_state(r->cb, r->snap);
  }
  ap_process_request(r);
```

During execution, session state changes after a write to the socket, as the offset in the serviced stream changes. The worker records the change by exporting its LWS:

```
int ap_send_file(FILE *f, long length) {
  offset = r->snap.off;
  fseek(f, offset, SEEK_SET);
  while(n = fread(buf, sizeof(char), len, f)) {
    w = ap_bwrite(r->conn->client, buf, n);
    r->snap.off += w;
    cb_export_state(r->cb, r->snap);
```

This example shows that the API can be used in complex server applications (Apache has 15,000 lines of code) with minimal code modifications. We have similarly instrumented other web servers [66], a streaming audio server [41], and the Internet auction service [22] described next.

3.6.7 Case Study: Session Recovery in a Multi-Tier Internet Service

As a test application for our system we chose RUBiS [22], a complex application that models an Internet auction service similar to e-Bay, integrated in a multi-tier architecture. RUBiS provides item selling and bidding, user accounts, and support for user rating/comments. The typical workload is a mix of browsing and updating of persistent data.

In RUBiS, bidding requests are important to users as the bidding system allows items to be listed for sale only for limited periods of time. Moreover, the typical behavior of bidders (wait until the end of the listing period in order to place "last

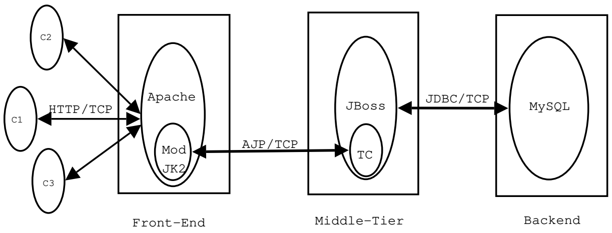


Figure 3.5: *The RUBiS multi-tier application architecture.*

minute" bids) makes their requests highly critical in the moments before an auction closes. If such a request is lost in a node failure, reissuing it from the client would run the risk of missing the deadline, duplicating the request, or not being re-admitted into an overloaded system. In contrast, in a BD-based system, the distributed state of a session is recoverable at various nodes in the service architecture. Once admitted, a request can be salvaged from any number of failed nodes, starting from the accepting front-end through the intermediate tiers down to the database back-end.

System Configuration. Figure 3.5 shows the RUBiS request processing path in a three-tier architecture running web servers on front-end (FE) nodes, application servers in the mid-tier (MT), and a transactional DB system on back-end nodes. We run Apache 2.0.47 [3] with the *mod_jk2* connector module on FE, the Tomcat 4.1 servlet container and JBoss 3.2.2 EJB server on MT, and MySQL 4.0.15 as the DB server. Client requests enter the system at the FE, pass from Apache through the Tomcat connector on to the specified application servlet running on the MT in the Tomcat container, and then on to JBoss, where RUBiS EJB Beans implement the e-commerce logic of the application. From here, queries are made via the JDBC driver directly to the DB server.

RUBiS with Backdoors. We run RUBiS on a system in which the front-end and mid-tier nodes have Backdoors recovery support. The Backdoors OS support is implemented in a modified FreeBSD kernel and uses Myrinet programmable interfaces [63] as backdoor NICs, as described in Section 3.9. The back-end is assumed fault-tolerant through well-established methods, e.g., DB replication. We make client sessions recoverable by modifying Apache and RUBiS beans to use the CB API. The CB API adds only 500 lines of code in Apache and 30 lines in the RUBiS beans, without any changes to the JBoss server. We have implemented the interface to the export/import system calls as a standalone service integrated with the JBoss server.

When a request enters the system, the FE tags it with a globally unique request ID, used to identify CB-encapsulated state belonging to the same session. On an FE node, the LWS of a session contains the request, its ID, and the offset reached in the output stream sent to client. On an MT node, where the request is translated into a DB transaction, the LWS contains the request ID, the transaction identifier, and the result of the transaction (one database record). The snapshots are light-weight, averaging only 99 and 44 bytes in front-end and mid-tier, respectively.

If an FE node fails, its monitor notifies other FE node(s) to extract the session CBs from it and reissue pending requests to the MT. If an MT node fails, its monitor notifies all FEs to reissue requests serviced by the failed MT node. For requests replayed during recovery, an MT node obtains the status (abort/commit) of the transaction from the database and retrieves the transaction result from the CB. It then uses this information to decide whether to reissue the transaction, and to rebuild the reply to be sent back to the client. This scheme relies on simple DB support for reconnects to achieve exactly-once semantics for DB transactions, while correctly (re)generating replies. To implement it, we have modified MySQL to support database reconnects and queries for the status of a transaction.

3.7 Remote Repair of Damaged OS State

In this section, we show how Backdoors can be used to perform automated remote *in-place repair* of damage to the OS state of a computer system. Damage to OS state may include effects of corruption in OS data structures (e.g., a corrupted file system) but is not limited to these. We regard the state of an OS as damaged when a certain OS subsystem is impaired and cannot perform its normal functions. The damage can have various causes: OS bugs triggered under load or other exceptional conditions, resource exhaustion due to a runaway user program or to heavy load on a computer used as a

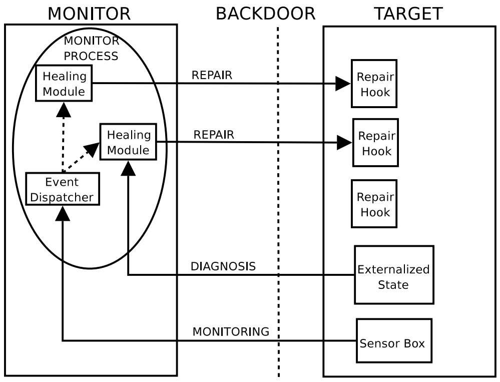


Figure 3.6: *Software architecture for remote repair using Backdoors.*

server, system misconfiguration, attack, etc.

We describe a BD-based system that enables automated monitoring, diagnosis and repair actions on a system even when that system is crippled by bad OS state. Using our system, a monitor machine can observe the state of a target machine and take repair actions when it detects exceptional conditions. We present a case study in repair of OS state damaged by resource exhaustion, exemplified by two instances: remote memory reclamation from memory hogging processes, and remote process table repair after a fork bomb.

3.7.1 System Overview

Figure 3.6 shows a BD-based remote repair pair consisting of a monitor (M) and a target (T) machine, along with their interactions. As previously described, a specialized monitor process running on M observes the state of T and identifies exceptional conditions (*monitoring*). It then determines the problem that affects the target system (*diagnosis*) and performs remote repair operations on it (*repair*). All these operations, shown as labeled horizontal arrows in Figure 3.6, are performed through Backdoors.

To enable low overhead monitoring, the target OS allocates a Sensor Box where monitored OS subsystems define and update sensors that indicate their health, as described in Section 3.4. The target OS also provides *externalized state* for validation of exceptional events and problem diagnosis by enabling remote read access to fine-grained OS data structures. It also enables remote writes to *repair hooks* (RH) in the OS to allow state repair. Repair hooks can be regions of OS memory that control the execution behavior of the monitored system (e.g., configuration variables), raw OS data structures (e.g., file, process, inode), etc.

The monitor process consists of an event dispatcher (ED) and one or more healing modules (HMs). The ED retrieves the SB (periodically or on demand) from the target and uses it to perform an efficient detection of exceptional events. The problem is that although SB provides a lightweight mechanism for detection of exceptional conditions, this might be too coarse-grained since it relies on simple counters of events. To decide whether repair is actually needed and to accurately diagnose the problem may require more fine-grained knowledge of the target system state. To achieve this, a target system enables remote read access to a part of its OS state (the externalized state), on which healing modules can perform fine-grained inspection to accurately diagnose anomalies.

The monitor associates an HM with one or more sensors in the target's SB using the following API:

```
hm = new_hm(hm_handler)
register_sensor(hm, sensor)
```

On detection of a problem reflected by sensors in the SB, the ED dispatches calls to the pre-registered HMs associated with the sensors involved. An HM is a user defined plug-in that performs fine-grained detection, diagnosis and repair actions on the target system state. HMs are opaque to the ED, which only maintains the association between them and the sensors they handle. There may be multiple sensors associated with one HM, and one sensor may be associated with multiple HMs.

To perform fine-grained detection and diagnosis, the HM may retrieve externalized state from the monitored system to validate the event and to diagnose the problem before invoking the repair action. After diagnosis, an HM identifies the corrective measures needed to bring the damaged OS back to normal operation and performs remote repair actions on it using one or more repair hooks (RHs). The repair action may involve remote modification of target OS data structures, manipulation of control variables, shutting down the target system, or even overwriting a clean system image on the target system.

RHs are defined by the target system OS as regions of OS memory to which remote write access is enabled. The actual specification of a hook depends on the domain of its intended action. For example, a file system repair hook may enable access to in-memory superblock and inode blocks, a process table repair hook may enable access to fields in the process structure that control behavior of a process (priority, signal handling), a system corruption repair hook may enable overwriting the system image to bring the system back to a trusted clean state, etc. It is important to note that the BD architecture only defines the basic infrastructure, on top of which system specific monitoring, detection and healing modules can be implemented.

3.7.2 Case Study: Repair of OS State Damaged by Resource Exhaustion

We illustrate the remote repair mechanism with two OS resource exhaustion scenarios in which traditional techniques *(i)* fail to prevent a system from becoming unavailable due to resource exhaustion, and *(ii)* cannot repair the system. We describe each scenario, show why the traditional mechanisms fail, and describe a remote healing solution.

ForkBomb: Process Table Repair

A forkbomb is a process that recursively spawns new processes, without doing useful work, until the resources on the system are exhausted. A forkbomb hogs the CPU of the system and does not allow other processes to execute. It also causes the process table in the OS to fill up, preventing new processes from being created. In addition, a forkbomb indirectly starves all processes, as the scheduler has to traverse a large list of processes to identify and update their priorities (no user or system activity is possible when the scheduler is running).

In our test system (FreeBSD), the OS protects against the forkbomb or any such runaway process by limiting *(i)* the maximum number of processes per user, and *(ii)* the maximum rate of process creation. When any of these limits is exceeded, the user is "locked out" of the system by killing all her processes, and not allowing her to create new processes. Other operating systems have similar protection mechanisms.

Such simple mechanisms provide only limited and easy to defeat protection. On a heavily loaded system, a forkbomb would exhaust the available CPU cycles before the system limits are reached, so the built-in OS protection will not work. The system is not dead, as all its hardware components and the OS are functioning correctly, but cannot do any useful processing. Obviously, the system is also inaccessible for repair through the console or the network since new processes (at least the shell) are required to execute any repair task. The forkbomb also prevents any existing watchdog processes, e.g., daemons, from repairing the system by starving them of CPU cycles.

In contrast, in a BD-based system we can efficiently detect and eradicate a forkbomb. To achieve this, we use an SB with two level sensors: *(i)* `NPROCS`, the number of processes in the system, and *(ii)* `MAXPROCRATE_PERUID`, the maximum rate at which a user spawns processes. The monitored system enables remote access to its process table as externalized state, and defines RHs with the signal masks of all existing processes.

Monitoring and detection. The monitor process registers a `Proc_Repair_Healer` healing module with the event dispatcher and associates the two sensors with it. The ED performs the coarse-grained detection using the sensors and passes control to the `Proc_Repair_Healer` when a forkbomb is suspected. The `Proc_Repair_Healer` HM retrieves the remote process table and creates its local view.

Diagnosis. `Proc_Repair_Healer` uses three basic policies (which can be combined to define more complex policies) to identify a forkbomb on the monitored machine:

`POLICY_TOO_MANY_PROCS`: The HM traverses the process table to identify the user with the largest number of processes as the culprit.

`POLICY_RATE_TOO_HIGH`: The HM traverses the process table to identify the user who creates processes at a rate higher than the threshold as the culprit. This policy prevents a malicious user from avoiding the hard limits imposed on the number of processes per user by creating short lived processes at a high rate.

`POLICY_TREE_TOO_DEEP`: The HM traverses the process table to identify the user whose process tree depth exceeds a threshold. The child of a process at depth n is defined to have depth $n+1$. A deep process tree is generated when a process recursively spawns child processes. With this policy, the user with maximum process tree depth is identified as the culprit.

Repair. If a culprit is identified, `Proc_Repair_Healer` traverses the remote process table and posts a non-maskable signal (`SIGKILL`) to terminate all processes owned by the user found to be the culprit. The signal is posted by setting a flag (using remote write) in the process signal mask repair hook.

MemoryHog: Memory System Repair

A process or a group of processes that allocate large amounts of memory may cause the system to exhaust its memory. The virtual memory abstraction allows each process to allocate the maximum addressable memory. Under memory pressure, unused memory is moved to a backing store for anonymous memory called swap space. We define the usable memory on the system as the sum of the physical memory size and the swap space size.

In our test system (FreeBSD), the OS limits the maximum amount of memory allocated per process. This limit cannot be too low since useful processes with a large memory footprint would be hampered. As a result, the maximum usable memory in the system can be exhausted with a small number of processes that allocate the maximum allowed memory without freeing it.

When the entire usable memory is exhausted, the OS has no alternative but to reclaim memory from processes. It calls an out-of-memory handler that chooses the process with the largest memory footprint and kills it. Unfortunately, such a brute force policy does not prevent a process that creates several child processes, each of which continuously allocate memory, from exhausting system memory. (In one experiment, with as few as 30 such processes, we were able to block any useful execution on the system.) Moreover, this random policy can choose as victim a useful process if it happens to have the largest memory footprint.

Despite built-in protection, a system running a memory hog becomes unusable since no new processes can be created. Local repair is also impossible if it requires allocation of memory - the resource that has been exhausted. In contrast, a BD-based system can accurately identify the memory hog and perform state repair.

Monitoring and detection. To detect a memory pressure situation remotely, we use a pressure sensor `OOM_KILLER_RUNNING`. The monitored system increments the sensor value when the out-of-memory handler starts execution. The monitored OS externalizes its process table and defines an RH with the signal masks of all processes.

The monitor process registers a `Mem_Reclaim_Healer` healing module with the ED and associates the sensor with it. The ED performs the coarse-grained detection using the sensor and passes control to the `Mem_Reclaim_Healer` if it detects memory pressure. The `Mem_Reclaim_Healer` retrieves the process table and creates its local view.

Diagnosis. `Mem_Reclaim_Healer` uses three policies (or a combination thereof) to identify a memory hog on the monitored machine:

(i) `POLICY_MAX_RSS`: The HM traverses the process table to identify the process with the largest memory footprint as the culprit. This policy is identical to that used by a local out-of-memory handler.

(ii) `POLICY_MAX_RSSUID`: The HM traverses the process list and classifies all processes according to the userid. The user whose processes have allocated the maximum amount of memory is identified. With this policy, all processes owned by that user are chosen as culprits.

(iii) `POLICY_MAX_RSSUID_SAVEUID`: This policy is identical to `POLICY_MAX_RSSUID`, but the administrator can configure a list of users whose processes are critical to the system execution and must be spared. The processes belonging to the users in this list are filtered out from the process table before applying `POLICY_MAX_RSSUID`.

Repair. To reclaim memory, `Mem_Reclaim_Healer` posts a non-maskable signal (`SIGKILL`) to terminate all the culprit processes. The signal is posted by setting a flag (using remote write) in the process signal mask repair hook.

In Section 3.9 we will present results from experiments with process table and memory system repair, showing that our Backdoor-based system can detect the OS damage fast and with low overhead, and that it effectively recovers the affected machine.

3.8 Discussion

We advocate Backdoors as a new way of designing systems with a built-in alternate path for remote access to be used for accurate monitoring, recovery and repair operations. Key to our approach is that these healing actions can be performed lazily, even after a failure renders a machine unavailable by conventional means. Recovery by extraction of critical state from a failed system is a last resort action that can deal with the most severe system-hang failures.

While we describe a prototype and case studies in recovery/repair with a tightly coupled implementation based on a local-area interconnect, the BD idea does not rely on or require a particular carrier or interconnection technology. We envision backdoors built over wide-area, or by using standard access interfaces like USB. One promising step in this direction is the IETF effort for development and standardization of remote memory access over the Internet [5].

One advantage of our fine-grained recovery model is that it can be made more robust to propagation of bad state than heavy-weight approaches that recover large amounts of unstructured state from a system (checkpointing, process migration, VM migration, hot backups, etc.). Smaller state components enable recovery of "good" state by identifying and filtering occurrences of bad state (caused by misconfiguration, corruption, etc.). For example, checksums computed over application-controlled LWSs (Section 3.6) can prevent the accidental injection of corrupted state into another healthy system. In contrast, moving a whole process context or VM would reinstate *all* the "bad" state it may encapsulate.

Fine-grained memory protection through software and/or hardware support has been studied in systems like Rio [25], Nooks [94], and Mondrian [98]. For full recovery, our current BD implementation relies on the assumption that critical state is not corrupted during a failure, a property that can be enforced using these or similar techniques. Our current system cannot guarantee recovery of *all* state if a faulty OS issues wild writes that corrupt critical OS/application data structures.

We note however that kernel memory corruption is not a major cause of failures. Failure statistics drawn from field error data [87], synthetic fault-injection tests [25], and examination of problem report databases for open-source kernel development [37] support this observation, showing that: *(i)* memory corruption during an OS failure is a fairly uncommon event; *(ii)* memory corruption tends to affect mostly small-sized regions and occurs near the target address; *(iii)* excluding corruption, the majority of remaining faults consist of undefined state errors, e.g., a device driver going into indefinite wait or deadlock. A simple inspection of problem report databases for Linux or FreeBSD kernels confirms that deadlocks and system hangs make the vast majority of reported system failures. Moreover, especially in a hardened OS, memory corruption errors can be reproduced, debugged and fixed over time, while timing and race errors that lead to system hangs are much harder to reproduce and fix. It is exactly this latter class of failures that Backdoors can reliably detect and recover from.

Using Backdoors for remote healing comes at the risk of enabling access from monitor(s) to the OS memory of a target system, which makes the BD a potential access point for mounting attacks. An attacker that takes control of a monitor machine could exploit the BD interface to read sensitive information or to remotely write to the target system and irreversibly compromise it. The rudimentary access control provided by the BD by selective access to parts of the in-memory state of the target OS may not be sufficient.

To address this problem, BD can be made secure through a two-level solution: *(i)* access control through trusted hardware and firmware, and *(ii)* monitor replication. The access control level will take advantage of the narrow interface offered by BD to implement protected access control and validation mechanisms in the backdoor NIC firmware, at the level of primitive operations (e.g., memory accesses). The idea is to use the I-NIC as a secure co-processor which executes trusted code that cannot be tampered with, and can even store secrets in a memory which is not accessible from the host system. To implement trusted and protected low-level access control, a *Backdoor guard* can be run as part of the firmware that implements low-level access operations (remote read/write). Placing the guard in firmware and disabling access to its implementation after system initialization (even for legitimate users of the system) makes it tamper-free.

The second level of protection against attack is the functional level, by logical replication of the monitor side of the BD across different machines. In a practical system, to achieve fault tolerance of the monitoring function, the monitor side of a BD will have to be replicated on different machines. This redundancy can be exploited to also achieve resilience to attacks. Operations on the same target system issued from multiple monitors will be subject to low-level protected validation mechanisms implemented in the BD guard, before being applied to the target machine. ince

Remote write operations can be validated by the guard on the target side of a BD through a *delayed write agreement* protocol implemented in the BD firmware. In this protocol, the target side guard will enforce consistency rules, e.g., write operations used to repair a system by modifying a region of its memory must produce the same values when performed from multiple monitors. The target BD guard (trusted, not malicious, not faulty) will monitor incoming writes for a particular memory location and only perform the write if values agree. Similarly, remote reads will only be allowed if they match in address and length. This simple all-or-none scheme specifically relies on the asymmetry of the parties involved (the possibly malicious monitors and the low-level "guard" entity on the target side of the BD). The guard is weaker, in the sense that it does not have enough intelligence to decide whether a particular operation is correct or not, but can be trusted to propagate correct operations when all or a majority of monitors are correct, according to a preset access policy.

3.9 Prototype and Evaluation

3.9.1 Backdoors Implementation

We have implemented a BD prototype in the FreeBSD 4.8 x86 kernel, using Myrinet Lanai-XP programmable backdoor NICs [63]. For remote monitoring and state extraction, we modified the Myrinet GM 2.0 library to provide in-kernel remote memory read/write operations between monitor and target machines.

Remote OS Access

Remote access is enabled by registering the kernel memory of a target system with its backdoor NIC. Because FreeBSD allocates OS memory from a kernel virtual memory map, a kernel virtual page may map to different physical pages (if freed and reallocated). This is a problem for the NIC, which uses a translation table of virtual-to-physical memory mappings and maintains a mapping cache for fast lookups of frequently used mappings. To keep the NIC table in sync with the kernel page tables, we dynamically update virtual-to-physical mappings when needed (on kernel memory allocations).

Performing dynamic mapping updates also requires flushing stale entries from the NIC mapping cache. Flushing the cache on every mapping update incurs a high cost on a critical path (kernel memory allocation) and may create synchronization problems between the host processor and the slower Lanai. To avoid them, we chose instead to completely disable caching. The incurred penalty is negligible, given the low frequency and volume of the monitoring traffic (SB is light-weight and fits in one page, requiring just one translation lookup for access). For recovery or repair, an infrequent event, the penalty from not caching is paid only once.

Remote OS Locking

We have implemented a *remote OS locking* mechanism that blocks execution of system calls and of interrupt/exception handlers on the target machine. Remote OS locking is used: *(i)* to freeze a suspected target before starting recovery and thus completely eliminate unwanted effects of false positives in failure detection, and *(ii)* to freeze the target OS in order to have a consistent view of its state while performing diagnosis and repair operations. We have also used remote OS locking to remotely freeze a machine during emulated failure experiments.

Remote OS locking uses remote read/write operations on a "giant" shared-memory lock, in a two-phase handshake protocol. To acquire the lock, a remote requester (monitor) atomically writes a one-word lock request in target's memory. Lock acquire operations on the target OS were altered to check for posted remote lock requests after acquiring the lock, but before allowing the local acquirer to enter the critical section. If a remote lock request is pending, the local acquirer on the target relinquishes the lock to the requester by writing back to signal that the remote OS lock is free, then blocks (spins) waiting on a flag. To later release the lock, the holder writes the lock free flag remotely and the target OS resumes normal operation.

This implementation relies on a giant lock maintained and used by the native OS for its own purposes (e.g., in some SMP versions of FreeBSD and Linux, for mutual exclusion in accessing related kernel data structures). However, in a kernel that allows fine-grained access to its data structures, adding a giant lock would kill concurrency. In this case, a better implementation is possible by noting that the *local* global locking primitive is actually not needed to block *remote* accesses. At the time remote access is granted by a local acquirer, the local global lock ensures that: *(i)* no other code will be able to enter the kernel (since accesses are guarded by lock acquires and the lock is held), and *(ii)* no code is currently executing in the kernel (since the lock can only be held by the acquirer, which has not yet entered the critical section).

The idea is to enforce these two conditions *separately*, from the remote node, with assistance from the target OS. To enforce *(i)*, the remote node will set a blocking flag that the target checks at all kernel entry points (system calls, interrupt handlers, fault handlers, etc.). To enforce *(ii)*, the target OS will maintain a counter of threads currently executing in the kernel for each class of entry points, atomically updated at entry and upon exit. After setting the blocking flag, the remote node will wait until all threads of execution have "drained" from the kernel, i.e., until all thread counters become zero. Note that in both implementations a timeout can be used to break an infinite wait, e.g., if the system hangs with the giant lock held or with a nonzero counter.

Failure Detection

We implemented the SB as a statically allocated region in kernel memory, and its interface as a pseudo-device to be accessed both from the kernel (at the target, for liveness reporting via progress counters) and from user space (at the monitor, for sampling the remote SB in a monitor process). A monitor process samples the target SB, compares the current and previous view of the SB, and identifies progress counters that have been stalled beyond their detection deadlines. If no progress is detected in a critical counter, the monitor initiates recovery actions. In our Internet service case study, an action may involve issuing a system call to extract all connections bound to a specific port from a failed front-end, warn front-ends to redirect their requests away from a failed mid-tier node, perform cluster reconfiguration, etc.

Recovery Support

We have implemented the CB mechanism (including support for TCP connections and OS pipes) in the server OS kernel, without changes to client OS/applications. The CB abstraction is implemented as a data structure that maintains pointers to state components (state buffers, communication logs, etc.) and log control information (read/write pointers). Logging is performed in-place using the kernel data buffers (TCP, pipe, etc.). State snapshots are copied from user to kernel buffers during a `cb_export` system call. We have also implemented an optimized (no-copy, no-syscall) version of `cb_export` using kernel mapped user-level buffers.

Extraction of CB state from a failed machine requires remote traversal of linked data structures. This involves recursively fetching a data structure and issuing remote read requests to follow pointers to other regions of memory stored in it. Following remote pointers is an unavoidable cost of the extraction protocol on chained native OS data structures. Section 3.9 shows that this is not overly expensive for a CB, which has a fairly low number of state components of small size.

Our client session CB implementation enables extraction of an established client TCP connection in any state and is made transparent to client's TCP by using IP address takeover from the failed machine. We overlap CB extraction from a failed node with traffic on other salvaged connections to the maximum extent possible, resuming traffic on a client connection as soon as its server-side endpoint has been reinstated at the new node. While inbound packets may not reach the new node until IP takeover takes place, packets buffered in the failed node are immediately sent out and the new

State size $[KB]$	*export* $[\mu s]$	*import* $[\mu s]$	*CB extraction* $[\mu s]$
1	11	8	158
5	20	10	258
10	28	24	358

Table 3.2: *Cost of CB system calls and CB state extraction.*

server can resume service and continue to generate new data for the client. We optimized CB extraction for the case of an Internet server (which may hold thousands of client connections, all bound to the same port) by providing a single system call that extracts all session CBs associated with a given server port from a failed machine.

3.9.2 Experimental Setup

The goal of our evaluation is to show that our system reliably detects failures, is non-intrusive to server applications and has minimal impact on the client. We first perform overhead microbenchmarks and then present results of the experiments run on our RUBiS service testbed.

The experimental setup consists of DELL PowerEdge 2600 2.4 GHz, 1 GB RAM dual-processors interconnected by 1 Gb/s Ethernet. Server nodes run FreeBSD 4.8 incorporating our BD prototype. The BD is implemented with Myrinet Lanai-X NICs with a 133MHz PCI-X interface [63].

To generate realistic fault injection tests, we have modified several Ethernet network drivers (Intel Pro Gigabit Ethernet, 3COM 3c59x, etc.) to insert programming errors that cause a system to crash. Victim systems were also subject to controlled synthetic failures: processor halt, disabling the interrupt controller, selectively disabling device interrupts, etc., or were simply frozen by trapping into the kernel debugger. Failures were detected by progress sensors on the number of interrupts and context switches.

3.9.3 Microbenchmarks

SB/CB API Overhead and CB Extraction Cost

In the first experiment, we evaluate the run-time overhead of the Sensor Box (SB) and Continuation Box (CB) API by measuring the latency of the calls described in Section 3.6. Of the two components, the SB API is extremely light-weight, as it only writes integer values to an SB. On the other hand, implementing an efficient CB API is crucial to the failure-free performance of a system. Table 3.2 (columns 2-4) shows the cost of CB calls for three values of the CB state size. We can see that the CB API is also light-weight and imposes little run-time overhead for updating and retrieving CB state. We conclude that participation in monitoring and providing recovery support should be light-weight on a server node.

In the second experiment, we estimate the costs of extracting a CB from a failed system as a function of the CB state size. We use a recoverable (i.e., augmented with the CB API) synthetic server application that does not generate data. This eliminates variability in the amount of state extracted from the server node while we control the amount of CB state by conveniently varying the size of the exported snapshots in the server application, between 0 and 10 KB. We measure the time taken by CB extraction to move and reinstate the state of a TCP connection along with the associated application-level snapshot. This setup is typical of state extraction from front-end nodes during recovery in a multi-tier architecture. The last column in Table 3.2 shows the results, proving that CB state extraction is light-weight for the recovery node. For comparison, the raw latency of a remote memory read is about 16 μs for small payloads, and 118 μs with a 10 KB payload.

Monitoring Overhead

On a monitor node, the overhead includes *(i)* the monitoring cost (reading the local view of the monitored SB, comparing counter values, etc.), and *(ii)* the cost of transferring the remote SB from the monitored node. To determine it, we measured the CPU usage of a monitor process while varying the sampling rate of a remote SB with 100 progress

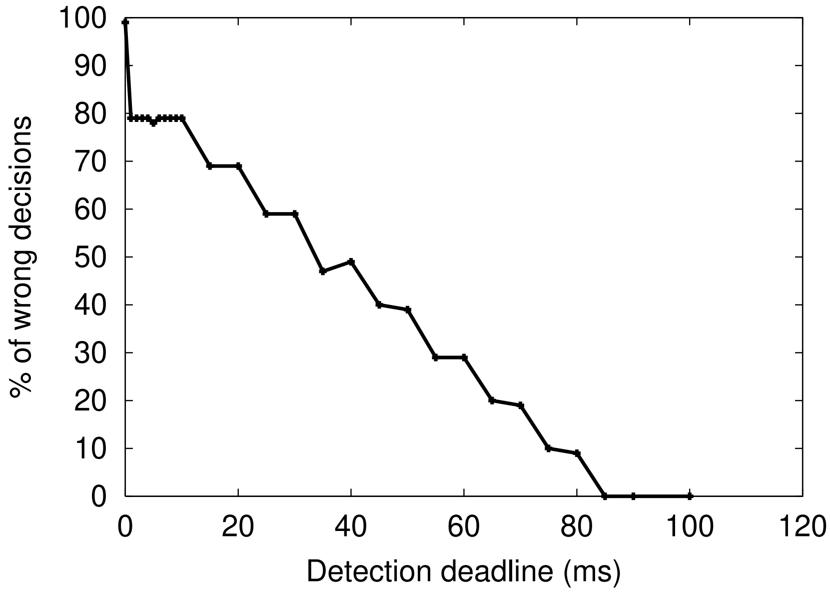


Figure 3.7: *Variation of false positives in failure detection with the detection deadline.*

counters. In the worst case (sampling the SB in an infinite loop), the CPU usage is 46%. Sampling every 10 ms (the lowest granularity of a timer), the CPU usage is about 5%, while at 100 ms it drops under 1%. This shows that fast failure detection can be performed with low overhead on a monitor node.

Detection Deadlines and False Positives

This experiment shows that the detection deadline τ of a progress counter must be carefully chosen to match the behavior of the counter in order to avoid false positives at a monitor. A *false positive* occurs when a healthy node is wrongly declared failed by a monitor.

There exists a tradeoff between fast failure detection and the rate of false positives: a greedily chosen short τ may not leave time for the counter to be updated. To illustrate this tradeoff, we artificially induce false positives in failure detection by a monitor, using as progress counter the number of scheduling decisions involving a processor P_1 of a 2-way SMP. A remote monitor samples the counter with period $\mathcal{T}$ equal to its detection deadline while a CPU-bound task runs on the node, pinned to P_1. Because the task does not block, if there are no other runnable tasks, no scheduling decision

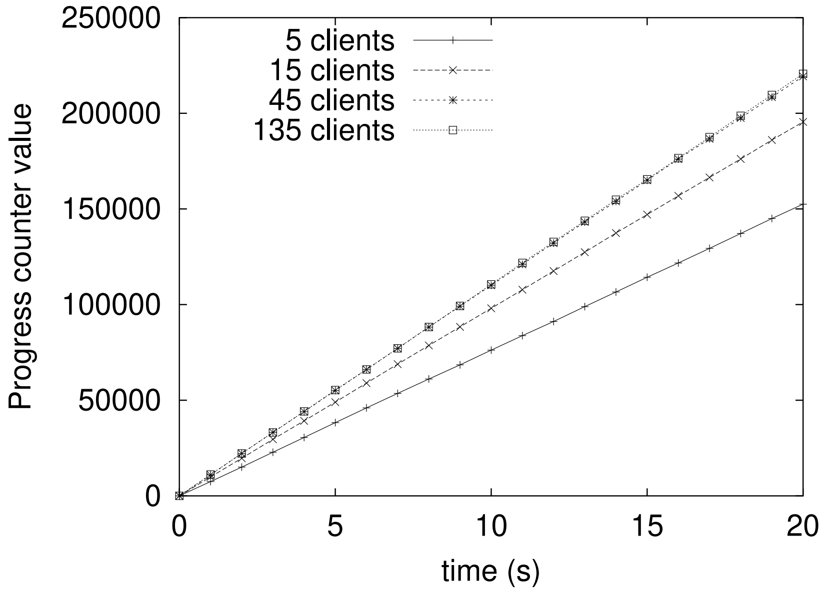


Figure 3.8: *Variation of an OS interrupt counter with time under different load conditions.*

may take place within a time-slice. The counter may stall for the duration of a time-slice, and a monitor may declare the node faulty if the detection deadline is smaller than the time-slice. Figure 3.7 shows the fraction of false positives with increasing detection deadline. Since the normal time-slice is 100 ms, deadlines under this value run the risk of inducing wrong decisions. As the deadline increases, so does the chance that a scheduling decision for P_1 is made before deadline. For deadlines larger than 85 ms, scheduling activity in the system eliminates false detection. This shows that the system is sensitive enough to fail "as-expected" and expose programming errors caused by unrealistic detection deadlines.

Counter Dynamics

To study dynamics of OS progress counters, we collected traces of a progress counter for the number of interrupts serviced in a system running a synthetic streaming server. Figure 3.8 shows the counter dynamics for various loads (number of client streaming sessions). We can see that while the counter is updated regularly (constant slopes), the rate of update depends on the load. This indicates that while such an activity-driven counter is a good indicator of overall system health, absolute reliance on it may lead to

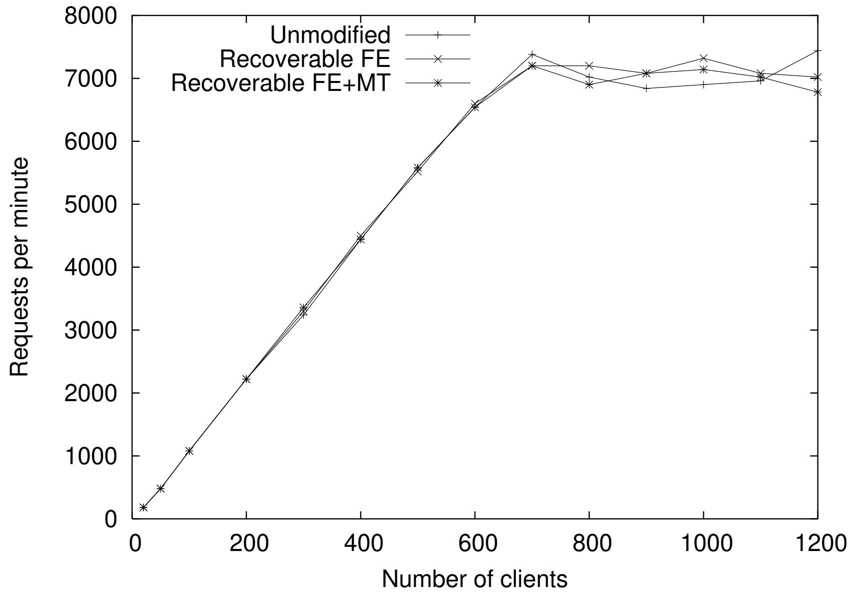


Figure 3.9: *Throughput of recoverable RUBiS is unaffected by recovery support.*

wrong decisions on an idle system with badly-chosen detection deadlines.

The last two experiments outline the need for a careful choice of detection deadlines for counters that are load-sensitive and/or can be easily overridden by a programming error. In general, such counters should be backed by more general watchdog-style counters (e.g., real-time clocks) in deciding that a node has failed.

3.9.4 Remote Recovery Evaluation

We have used our BD-based system to support recoverable sessions in several open-source servers (Apache [3], Flash [66], and Icecast [41]) and extensively used them to validate the correctness of the recovery scheme. In this section, we evaluate the performance and correctness of our system using RUBiS, the multi-tier auction service described as a case study in Section 3.6.7. The experimental setup consists of two front-end nodes (FE), two mid-tier nodes (MT), and one back-end node. In crash experiments, failures are injected in FE and MT nodes at arbitrary points during a run and sessions serviced by a victim node are recovered on the alternate node in its tier.

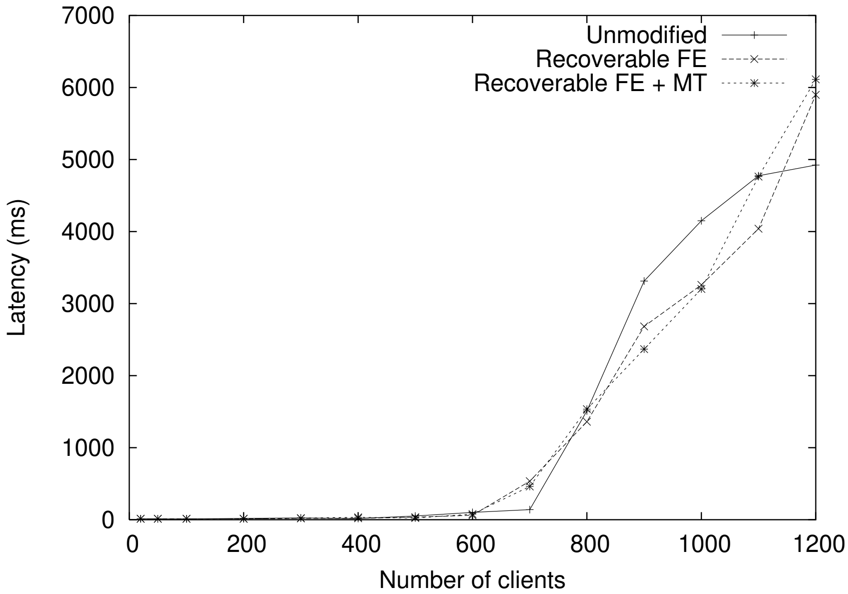


Figure 3.10: *Latency of recoverable RUBiS is unaffected by recovery support.*

Failure-Free Overhead

We show that using the CB API has no impact on client perceived performance by running the same workload on "base" servers (Apache in FE and JBoss in MT) and on recoverable servers, i.e., augmented with the CB API. The workload is a mix of auction requests that emulates client browsers using a methodology similar to [22], with think-times as specified by the TPC-W benchmark. We increase the load by varying the number of clients in runs of 6 minutes each. Figures 3.9 and 3.10 show the aggregate throughput and average latency perceived by clients for the base case, recoverable FE, and recoverable FE and MT. The system has identical behavior in all cases: the curves overlap at underload and exhibit statistically small variations at saturation (when performance degrades abruptly) due to nondeterministic system behavior.

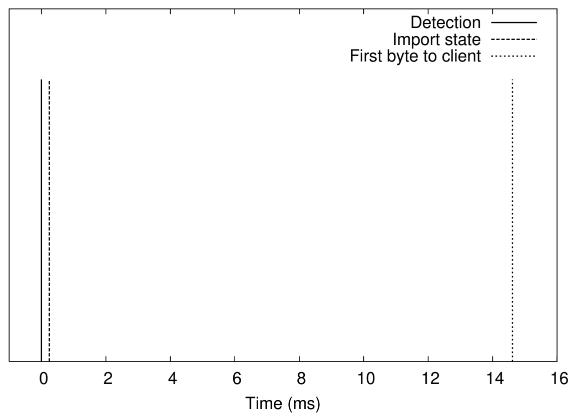
Failover Correctness

The goal of this experiment is to verify the correctness of recovery for RUBiS sessions on our BD-based architecture. The workload generator simulates requests from 200 clients, with a heavy synthetic workload in which each request performs a database transaction consisting of multiple queries and an update on the same table. We conduct multiple crash-test runs, each for one of two types of request: user registration and bid requests. After each run, we check the correctness of session failover with two tests: *(i) End-to-end consistency:* every client request is correctly matched by its expected reply; *(ii) Database integrity:* there are no missing or duplicate transactions in the database. The first test verifies the integrity of the communication channels in the request-replay path. To identify duplicate transactions, we rely on the RUBiS database schema which treats each update as a completely new one, and inserts a new record for it in the target table. All runs validate the correctness of our system: each client request receives the correct reply and every database transaction completes properly.

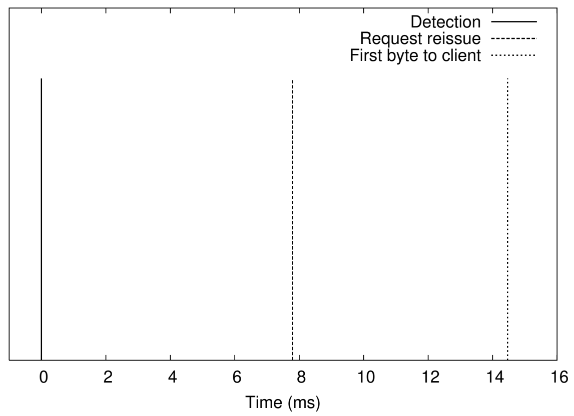
Failover Latency

To evaluate the impact of failure detection and recovery on client-perceived performance, we subject the system to crashes under a workload of 200 clients generating browse transactions in runs of 90s, with normally distributed think-times with 7s mean and a slow-down factor of 0.5 (see [22] for details). With this workload, the node CPU utilization is about 45% on FE and 15-30% on MT. We impose a failure detection deadline of 10 ms and emulate a crash in FE, MT, or both, 14 seconds into the run, by "freezing" the victim node(s) through remote OS lock operations initiated by the client machine.

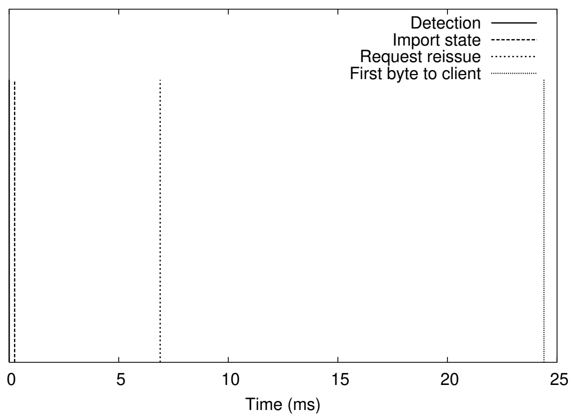
Figure 3.11 shows a timeline of events from detection of the crash to the end of recovery, in the worst-case, i.e., for the *last* recovered session. We define the end of recovery for a session as the moment the first byte sent out to the client by the FE after failover. When an MT node is a victim, we also plot the moment the request is reissued. The detection latency is only limited by our choice of detection deadlines and sampling period (as low as 10 ms). The *worst-case* recovery latency is under 25 ms in the 2-node failure case *(c)*. The best case values were 1.3 ms, 1.1 ms and 6.9 ms, with averages of 11.8 ms, 7.5 ms and 19.5 ms for the FE, MT, and FE+MT crashes, respectively. This shows that failover is fast and should practically have no effect on client perceived performance.



(a)

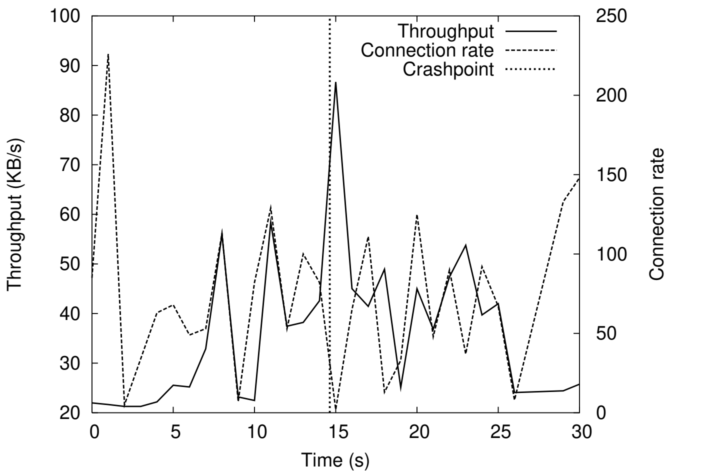


(b)

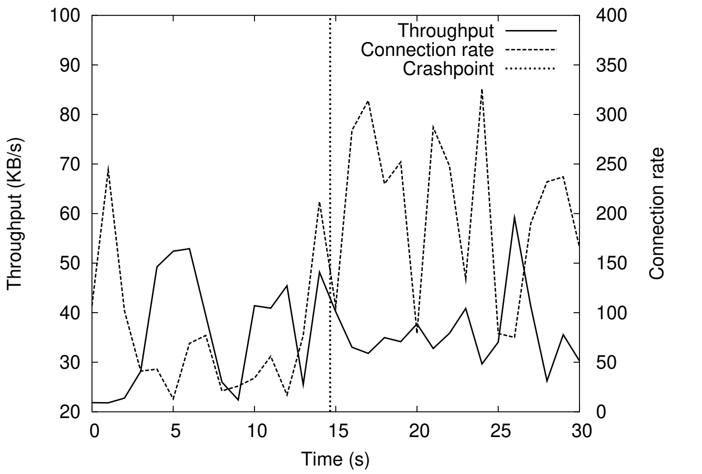


(c)

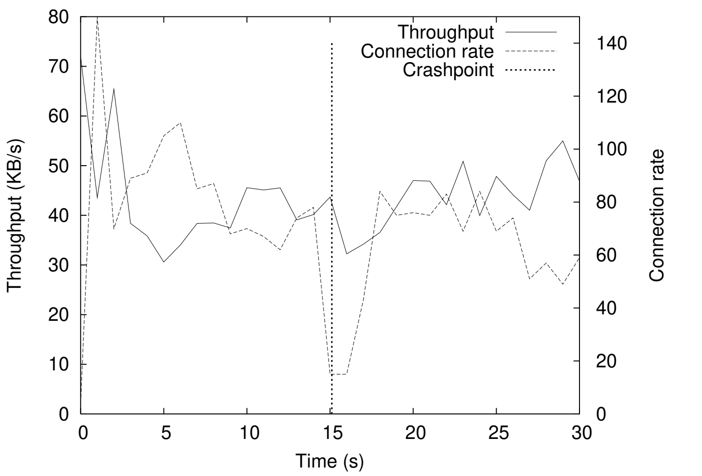
Figure 3.11: *Timeline of RUBiS session recovery from an FE crash (a), MT crash (b), and FE+MT crash (c), in the worst case (for the last recovered session). Each vertical line indicates a recovery event, after a crash has been injected at moment -10 ms. Recovery starts with the detection of the crash at moment 0. The worst-case recovery latency is under 25 ms.*



(a)



(b)



(c)

Figure 3.12: *Aggregate throughput and connection rate seen by the clients across an FE crash (a), MT crash (b), and FE+MT crash (c). Vertical lines mark the moment of the crash.*

Figure 3.12 shows the variation of aggregate throughput (as bytes received by all clients) and the rate of established client connections, measured in bins of 1s each, in a time window centered around the crash. The effect of crash and recovery is indistinguishable from normal workload variations. The jittery throughput is a well-known problem of the RUBiS client [18], and our concern was that such a "jumpy" workload profile would obscure the effects of fault handling. In fact, in Figure 3.12 there are no "hidden" side-effects of the failure simply because recovery is fast.

This should be even more evident considering that the low recovery latency compares extremely well with effects of packet loss on normal client-server TCP (data) traffic over the Internet: *(i)* For server-to-client traffic, recovery introduces a "gap" in the outgoing byte-stream comparable to Internet RTTs and granularity of TCP timers (tens to hundreds of ms over wide-area). This means that the impact of recovery as perceived by a remote client is not worse than that of a server packet loss in the Internet! This is because packet loss, a far more common event than server failures, results in a (potentially successful) retransmission by the server TCP after a timeout estimated from RTT measurements. *(ii)* For client-to-server traffic, data packets arriving at the server during recovery are lost. If all packets in a burst are lost and no other (new) packets are sent, the client TCP will timeout and retransmit. Since recovery is fast, the retransmitted packets will most likely arrive at the new server node after failover is completed, generating the expected ACK. The effect of the failover is again equivalent to a (single) packet loss. Moreover, even this may be obscured if the failover occurs within a large client retransmit timeout, and newer packets from the client reach the new server after failover. In this case, the server TCP's fast retransmit mechanism will elicit a retransmission of the missing packets by the client TCP.

3.9.5 Remote Repair Evaluation

In this section, we present an evaluation of our BD-based remote repair system presented in Section 3.7 using the two misbehaved programs described in our remote repair case study (forkbomb and memory hog). First, we show that diagnosis and repair in our system are efficient (we have already seen that monitoring is a lightweight activity for

No. of processes	Time (ms)
100	140
500	263
1000	350
1500	426

Table 3.3: *Variation of the repair time with the number of processes in the system.*

the monitor machine). Second, we show that a computer system can be brought down using the two rogue programs. We also show that the system cannot be repaired locally because it is unresponsive. Third, we show that we can detect and repair such cases using BD.

Diagnosis and Repair Cost

Diagnosis and repair involve traversing the remote process table and building a local view, identifying the culprit, and killing all its processes. The cost of repair therefore depends on the number of processes present on the target machine.

Table 3.3 shows the variation of the average cost of repair with the number of processes on the target system. While the repair cost grows with the number of processes, in the worst case, it takes less than half a second to execute. This shows that repair (an exceptional action) is fast, and also that it should not impose too much overhead on the monitor system.

Repair Effectiveness

To illustrate the two case studies of Section 3.7.2 and to show that remote repair works while local repair is practically impossible, we developed two programs: a forkbomb and a memory hog.

The forkbomb creates processes that execute in a tight loop until the CPU cycles on the system are exhausted. An I/O bound version of the forkbomb program continuously reads from a pseudo-device (/dev/zero) and writes to a null device (/dev/null). This prevents the scheduler from lowering the priority of the forkbomb and of its children.

The memory hog allocates memory until it exhausts system memory and swap space.

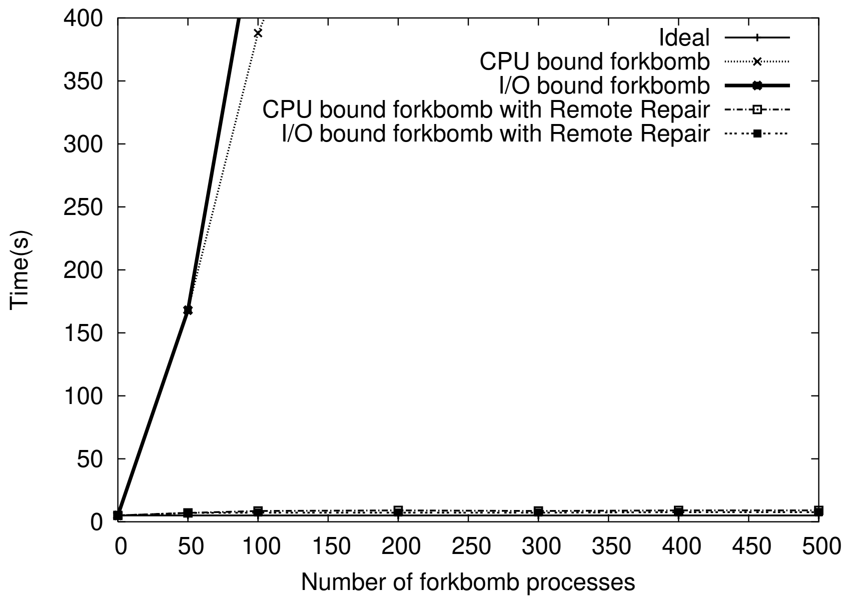


Figure 3.13: *Variation of the execution time of a test program with number of forkbomb processes, with and without remote repair.*

It is structured as a controller process that maintains a number of children processes, each of which allocates memory in a loop. If a child is killed by the system, the controller spawns a new process.

To illustrate the effects of the forkbomb and memory hog, we run a simple "useful" test program that executes in a loop, alone and concurrently with the forkbomb or the memory hog program. We measure the wall clock time the useful test program takes for each iteration. The results are plotted in Figures 3.13 and 3.14.

When there is no other load on the system, the time is constantly 5 seconds (the ideal time shown in Figures 3.13 and 3.14 as an horizontal line). When the forkbomb or the memory hog are executing, the useful process takes longer to receive its CPU share and the running time increases.

Figure 3.13 shows the variation of the wall clock time for the test program with the number of processes created by the forkbomb when *(i)* a CPU bound forkbomb executes, and *(ii)* when an I/O bound forkbomb executes. We see that the execution time grows unbounded for both forkbomb cases without repair, while it stays close to

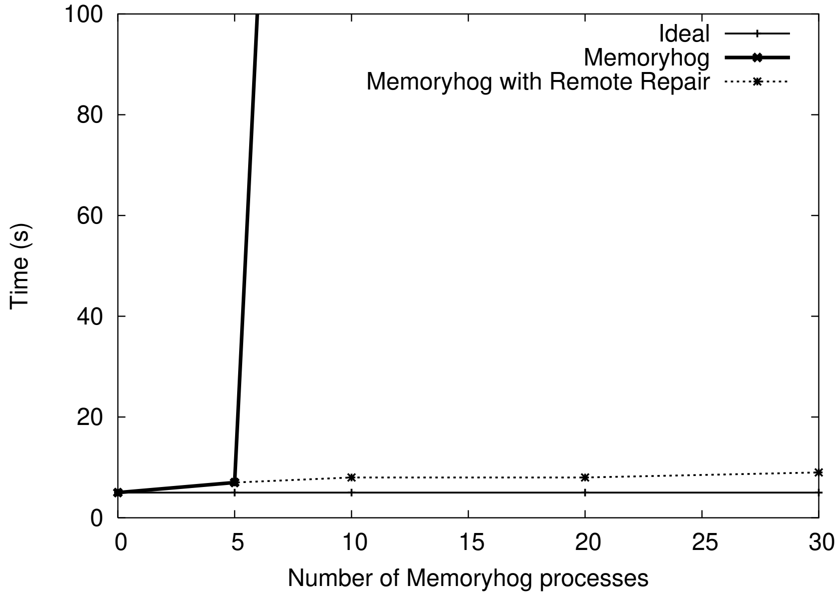


Figure 3.14: *Variation of the execution time of a test program with the number of memory hog processes, with and without remote repair.*

the ideal value of 5s when remote repair is performed.

Figure 3.14 shows the variation of the wall clock time for the test program with the number of processes created by the memory hog. Our system has 1GB of RAM and 2GB of swap space. Our test OS (FreeBSD) limits the maximum memory allocated by a process to 512MB, therefore up to 5 processes the system is well behaved and repair is not triggered. Once the memory is exhausted, the system becomes unavailable and execution time without remote repair explodes. With remote repair, the memory hog is identified, all processes with the same userid are killed and the system recovered with minimal disruption.

With around 400 processes created by the forkbomb, or with a pool of 30 memory hog processes, the test program did not complete for more than 30 minutes. In fact, we were unable to access the affected machine through the console to attempt any manual repair and we had to reboot in order to regain control over it. With remote repair, our system correctly identified the problem in all runs and was able to quickly recover the impaired machine.

3.10 Summary

We have introduced the concept of *remote healing*, a novel approach to system-level survivability and recoverability. In remote healing, a computer system performs automated monitoring of another target system to detect OS failures or damaged OS state, and performs recovery of useful in-memory state from a failed system or in-place repair of the state of a damaged system. Remote healing relies solely on lazy actions to restore the functionality of a system, without proactively mirroring its state on other machines. Essential to remote healing is that the remote access from the monitor system does not involve the processors of the failed system, nor does it use its OS resources.

To enable remote healing, we have developed Backdoors, a novel system architecture for nonintrusive remote access. Backdoors uses off-the-shelf programmable NICs for remote access to the memory of a machine even when its processors are unavailable due to severe OS failures (system hangs, OS crashes, deadlocks, etc.), and defines OS extensions for remote access to light-weight application and OS state.

We have designed Backdoors OS abstractions that support remote nonintrusive monitoring (Sensor Box) and recovery of critical application and OS state (Continuation Box). We have designed a Backdoors-based system for automated diagnosis and in-place repair of damaged OS state.

We have implemented a Backdoors prototype in FreeBSD and used it in two case studies: recovery of client sessions from server node failures in complex, transactional, multi-tier Internet services, and repair of OS state damaged by resource exhaustion. We have shown that our system can detect failures and can recover interactive client sessions from multiple failed nodes with no disruption to client sessions, without compromising consistency of the data seen by clients or database integrity. We have shown that the system can effectively restore the functionality of a computer affected by memory and process table exhaustion.

Chapter 4

Fault-Tolerant Distributed Shared Memory

4.1 Problem Statement

In the last decade, an impressive amount of research has been conducted in software distributed shared memory (DSM) mostly aiming for performance (e.g., relaxed consistency models, lazy protocols and communication hardware support) [43, 11, 20, 48, 49, 84, 76]. Advances in performance have been making it possible for DSM systems to use very large clusters as a cost-effective platform for data-intensive, long-running applications. Projects like InterWeave [24] even propose a shared-memory programming model to support applications that run on wide-area clusters of heterogeneous machines (meta-clusters). As cluster size and application running times increase, adding fault tolerance to such DSM based clusters becomes critical. At the same time, to preserve performance of the base DSM protocols, the fault tolerance support itself must be both light-weight and scalable, both in the amount of recovery state it needs to maintain and in performing automated management of the recovery state through garbage collection operations.

4.2 Background and Related Work

4.2.1 DSM Protocols

Software DSM uses the virtual memory mechanism of a native operating system along with message passing between machines to provide a shared-memory programming model on clusters of computers. Processes of an application run on distinct nodes of the cluster and use synchronization operations (lock `acquire/release` and `global barrier`), implemented by the system through message passing.

In a DSM system, like in any shared memory system, the view that a process has of the memory is governed by the *memory consistency model*, which dictates the order in which writes to shared pages on different nodes must be completed by the system. While formally defining the order in which changes to shared pages are made visible to other processes, the consistency model also impacts the performance of the shared memory implementation. The most efficient implementations derive from the Release Consistency (RC) model [38] and its variants. In release consistent software DSM, the memory consistency model dictates that processes must properly label synchronization points with `acquire/release` operations. If a program is properly synchronized, then RC guarantees that writes are completed no later than the next `release` operation, which ensures a sequentially consistent view of the memory to all processes at synchronization time.

In software DSM systems, the coherency data is propagated between nodes in the form of page invalidations. The contents of a page is updated lazily at page fault time, either from its home [104], or from the last writer(s) [48]. If the protocol supports multiple writers, then updates to a page are computed as a difference (*diff*) between the modified page and a reference copy created before the first write following a page fault [49].

Lazy Release Consistency (LRC) is a variant of Release Consistency in which propagation of writes is delayed to the time of an `acquire` [48]. Writes of a process are grouped into *intervals* delimited by synchronization operations. Page invalidations are propagated in the form of *write notices (wn)*, where a write notice is a pair (p, t) that specifies that some page p was updated during some interval t. The local logical time of a process is defined as a counter of local intervals. The partial-order relation *happened-before* [55] (denoted $\prec$) between intervals across nodes is captured as a vector of logical times called *vector timestamp* [48]. The vector timestamp T_i maintained by a process P_i keeps track of intervals for which write notices were received by that process.

4.2.2 Fault-Tolerant DSM Systems with Independent versus Coordinated Checkpointing

A common approach to building fault-tolerant systems is rollback recovery, where the state of a computation is periodically saved to stable storage (checkpointed) and used to restart the computation in case of a failure. For distributed computations, there are two options for checkpointing [33]: *(i)* coordinated checkpointing, where all processes coordinate to save a globally consistent state at each checkpoint, and *(ii)* independent (uncoordinated) checkpointing, where checkpointing is purely a local operation and, as a result, the set of the most recent checkpoints may not represent a globally consistent state of the system.

Pure independent checkpointing suffers from the problem of inconsistent recovery lines after the failure and rollback of a communicating process. To obtain a consistent global state, nonfailed processes may have to also rollback, which may in turn lead to the cascaded rollback of other processes. This phenomenon, known as "the domino effect," causes loss of useful past work and, in the worst case, may force all processes to rollback to their starting state.

One well-known solution to the domino effect is log-based rollback recovery [33], which uses checkpoints and logs of messages received by a process for execution replay during recovery. Message replay can ensure that the maximum recoverable state of the system is exactly the state before the failure. As a result, log-based rollback recovery does not force valid processes to rollback even when independent checkpoints are used.

Coordinated checkpointing has typically been used in recoverable DSM [21, 44, 50, 17, 27] because it is simpler and more efficient to implement in small-scale clusters and when non-failed nodes are assumed to be always accessible [32]. These systems either force a synchronization of all processes when taking a checkpoint, or leverage global synchronization existent in the application or in the operation of the underlying DSM system.

For very large clusters and meta-clusters, however, independent checkpointing becomes more practical than coordinated checkpointing because of the increasing cost of global coordination and the likelihood of temporary disconnections in communication. Non-blocking consistent checkpointing protocols could reduce the coordination overhead by allowing processes to take local checkpoints and continue execution without explicit (blocking) synchronization. However, such protocols have been shown to be non-optimal, forcing processes to take unnecessary checkpoints [19]. An additional advantage of independent checkpointing is that a process can conveniently choose when to checkpoint. This autonomy enables application-level optimizations on the checkpoint size, using techniques like *memory exclusion* [68] to include in the checkpoint only regions of the address space strictly needed for recovery,

Independent checkpointing with dependency tracking applied to DSM was explored by Janssens and Fuchs in [45, 46]. Recoverable DSM systems that use independent checkpointing and logging, the basic recovery technique that we also use, are described in [73, 93, 27, 64, 53, 52]. These systems differ widely in the memory consistency models they use, the frequency of logging and/or checkpointing, the logging support used (disk or volatile memory), etc. They either *(i)* use memory consistency models and coherency protocols that cannot achieve good performance in a DSM system, or *(ii)* use expensive techniques, often interposed on the critical communication/execution path, to garbage collect the recovery state during execution. From the memory model viewpoint, [73] is based on sequential consistency, an inefficient model for DSM, and logs all memory accesses, which makes it extremely expensive, while [64] uses an entry-consistent protocol, limited to a single-writer, object-based DSM. From the performance viewpoint, even when they use advanced consistency models like LRC, these systems require global coordination for their operation [27], perform frequent log flushing to stable storage [73, 52, 93], or use expensive (global) checkpointing operations to perform garbage collection [93, 27] when not ignoring it altogether [52]. Section 4.6 will provide a detailed discussion of these systems and compare them to our approach.

4.3 A Fault-Tolerant DSM Based on Independent Checkpointing

4.3.1 Research Issues and Approach

The first research issue that we address is how to efficiently combine independent checkpointing with a state-of-the-art DSM protocol in order to provide a fault-tolerant yet high-performing shared memory programming environment on commodity clusters. The choice of independent checkpointing is dictated by its lack of coordination which makes it scalable to arbitrary cluster sizes and (as mentioned above and demonstrated in [19]) by its optimality with respect to the number of checkpoints taken. Specifically, we show how a Home-based Lazy Release Consistency (HLRC) DSM protocol [42] can be extended with independent checkpointing in order to efficiently tolerate single-fault failures. Our fault tolerance algorithms are completely distributed, do not force processes to synchronize, and only make use of available protocol information.

The choice of the base DSM coherence protocol is important because *(i)* the protocol must scale well with cluster size, *(ii)* the protocol must have low memory overhead, freeing memory for fault tolerance related tasks (for example, logging), and *(iii)* the protocol must be light-weight in terms of state maintained and thus incur less overhead for logging and checkpointing. The HLRC protocol has been shown to have these properties [104]. The challenge we face is to keep HLRC efficient while adding recovery support to it. In order to keep overhead low, our system *(i)* uses volatile memory for logging, and *(ii)* aggressively exploits the HLRC semantics to minimize the amount of state that must be logged or checkpointed.

The second research issue stems from the log-based recovery scheme based on independent checkpointing. To make independent checkpointing practical we have to control the size of the logs and the number of past checkpoints retained in stable storage. With independent checkpointing, processes cannot automatically discard their logs and old checkpoints when taking a new checkpoint because failed peer processes may need state or log entries saved prior to the last local checkpoint. We need a scheme for garbage collection of obsolete checkpoints and logs without forcing processes to synchronize or otherwise create extra system traffic, and which is robust to temporary disruptions in communication.

Our solution to this problem is to allow laziness into the garbage collection mechanisms, and to determine safe bounds on the state to be retained by taking into account the consistency constraints of the memory model. Laziness is involved in two ways in our mechanisms: *(i)* they use information lazily propagated in the system to compute bounds on the recovery state to be retained; *(ii)* they are decoupled from any policy that may decide when garbage collection is to be performed, or when a checkpoint must be taken. Using this approach, we devise two algorithms, Lazy Log Trimming (LLT) and Checkpoint Garbage Collection (CGC), to control the size of the logs and the number of checkpoints. We prove the correctness of our scheme in the context of fault-tolerant HLRC, and thus address the issue of augmenting it with a scalable garbage collection mechanism.

4.3.2 A Fault-Tolerant Home-Based Lazy Release Consistency DSM System

System Model

To design our system, we adopt a system model commonly encountered in earlier log-based fault-tolerant DSM systems, e.g., [73, 93, 27, 64, 53, 52]. We consider distributed applications running on clusters of interconnected computers, where each process of an executing application runs on a distinct node in the cluster. Processes communicate by message passing using reliable communication channels. Process execution is piece-wise deterministic [86] in the interval between the receipt of consecutive messages. Failures are fail-stop. A single process can fail at a given moment in time (single-fault failure) and a process is considered failed with respect to the state of the computation until it has completely restored the state it had before the crash. We assume that there exists a mechanism for failure detection. We do not consider logging of I/O interactions for recovery support. For fault tolerance purposes, we assume that the stable storage used by a process remains available after the failure of the process, which can be restarted on the same or a different node.

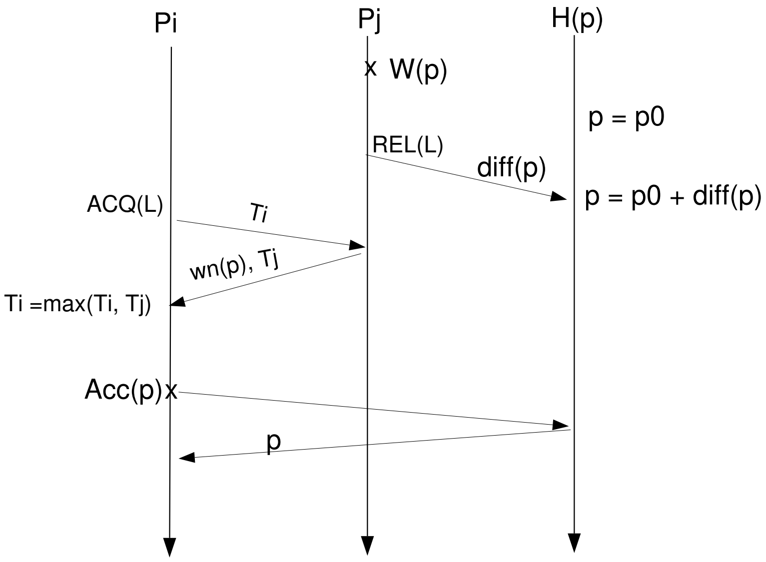


Figure 4.1: *The HLRC DSM protocol: process P_j writes to a page p homed by H(p) and which is later accessed by process P_i.*

The HLRC Protocol

In the Home-based Lazy Release Consistency protocol (HLRC) [42] every shared page p has an assigned home $H(p)$ which maintains the most recent version of the page. Suppose (Figure 4.1) that a process P_j has acquired a lock L before writing to page p (the write is labeled $W(p)$). After releasing L, P_j receives a request for L from P_i, which is acquiring the lock next. The lock granting message sent to P_i will include T_j and a write notice $wn(p)$ for p. Upon receiving L, P_i invalidates p and updates its vector timestamp as $T_i = T_i^{new} = max(T_i, T_j)$.

At release time, the writer P_j sends its diffs, labeled *diff(p)*, to $H(p)$, where they are applied to page p. The home $H(p)$ stamps p with a version vector $p.v$ that records the most recent intervals whose writes were applied to p; $p.v[i]$ advances with the application of a diff from P_i.

Following the invalidation of p at the acquire, the non-home process P_i records the version $p.v^N$ it *needs* in case of an access, according to write notices it has received. The first access Acc(p) of P_i to p after the invalidation will miss. In the page fault handler, P_i will send a request to $H(p)$ asking for a copy of the page with version at least $p.v^N$, the minimal version of p it expects.

Overview of Fault-Tolerant HLRC

Our fault-tolerant HLRC protocol operates as follows: *(i)* processes take independent checkpoints (decisions of when and what to checkpoint are purely local decisions); *(ii)* each process maintains logs of protocol data sent to its peers in volatile memory; *(iii)* a failed process will restart from its latest checkpoint and use logs from peer processes to deterministically replay its execution.

In our system, checkpointing can be transparent, or can be done at the request of the application. All logging operations take place transparently, only at synchronization points, exploiting the release consistency memory model. We use *sender-based message logging (SBML)* [47], in which the sender of a message logs it in volatile memory. SBML has low overhead, as logging can be done out of the critical message path, after the message is sent. Logs must be saved to stable storage at least on every checkpoint, since they must survive a crash and restart from the latest checkpoint.

To recover the state of a process in HLRC, we *checkpoint* shared pages only at home nodes and *log* those communication events that induce changes in the DSM state. A process P_i recovers from failure by log-based re-execution, starting from its last checkpoint. Because of the relaxed memory model, intermediate states of P_i during recovery, as well as its recovered state, do not need to be the same as during the previous normal execution. The execution replay needs only enforce that a *read access* to a page returns the same value, rather than indiscriminately applying all writes to the page.

Changes in the DSM protocol state at process P_i that must be replayed during recovery are *synchronization operations* and *shared memory accesses.* When recovering, a process P_i performs the replay of write notices received at synchronization points using logs kept by its peer processes. To replay shared memory accesses, we exploit the HLRC protocol to avoid the expensive logging of page transfers: a page p is checkpointed only at its home node $H(p)$, and diffs for p are logged by its writers. Also, since a request for a page p does not change the protocol state at $H(p)$, page requests do not need to be logged. For replay, a miss on p is serviced with a local copy of the page dynamically reconstructed from a checkpointed copy provided by $H(p)$, to which an ordered sequence of logged diffs has been applied. The home retains successive checkpointed copies of p from a sequence of past checkpoints. Because the checkpoints of P_i and $H(p)$ are not coordinated, the starting copy for P_i's replay of accesses to p may need to come from an older checkpoint of $H(p)$.

4.4 Design of Recovery Support for Fault-Tolerant Home-Based Lazy Release Consistency DSM

In this section we describe the design of the recovery support for the HLRC protocol. We describe the data structures maintained by the HLRC protocol at runtime (checkpoint timestamps, *write-notice, synchronization* and *diff* logs) and show how they can be used along with checkpoints of homed pages for recovery from single-node failures. We prove theoretical results on the minimal state needed to recover the state of the computation in the DSM protocol.

4.4.1 Checkpoint Timestamping

The basic idea of our recovery algorithms is to (partially) order checkpoints using vector timestamps: a process P_i *timestamps* a checkpoint with a vector T^i_{ckp}, equal to its vector timestamp T_i at the moment the checkpoint is taken.

Ideally, a *perfect* checkpoint timestamp T^i_{ckp} would be a *global vector time* T_g defined as $T_g[i] = T_i[i]\ \forall i$ (i.e., T_g describes the global state of the system at the moment the checkpoint was taken, assuming instant knowledge of logical times of all nodes). This is because, when re-executing after a restart from that checkpoint, the state of process P_i is guaranteed not to be affected by events at some other process P_j that happened (and were logged) before $T_g[j]$. These events with earlier timestamps will not be needed for recovery. The closer T^i_{ckp} is to T_g, the better it reflects the global state, and thus allows a more efficient trimming of recovery logs at peer processes P_j. Since in practice T_g is not available, we use the best approximation available to a process about the state of the system without any extra synchronization, which is its local vector timestamp, i.e.,

$T^i_{ckp} = T_i$ at the moment the checkpoint is taken.

4.4.2 Support for Synchronization Replay

Every process P_i logs write notices it generates in a volatile log, *wn_log*, for use in a log-based replay of synchronization operations by any recovering process P_j. A write notice log entry records the list of pages that were updated in a certain interval.

Every process P_j logs the lock acquire requests it services for any process P_i in $acq\_sent\_log[i]$, and the replies it receives from P_i to its lock acquire requests in $acq\_rcvd\_log[i]$. This means that each reply to an acquire message is logged at two nodes.

During normal execution of an acquire (ACQ), P_i sends its vector timestamp T_i to the process P_j, which owns the lock, and receives a set of write notices with the lock. During re-execution, P_i replays the ACQ using write notices from the *wn_log*'s of other processes, and advances its vector timestamp T_i exactly as before crash. The previous lock owner P_j supports the replay of P_i by computing and logging in $acq\_sent\_log[i]$ the *new* value of T_i, $T_i^{new} = \max(T_i, T_j)$. T_i^{new} is the new value of T_i after the ACQ has completed.

The $acq\_sent\_log[j]$ at a process P_i (as a lock owner) is a volatile data structure, subject to loss in a crash. Because it is created as a result of asynchronous events (lock acquire requests received from P_j), it cannot be regenerated during re-execution. In order to be able to recover $acq\_sent\_log[j]$, we simply mirror it at the acquirer P_j: after completing the acquire, P_j will log its new vector timestamp T_j^{new} in a per-process log $acq\_rcvd\_log[i]$.

Note that, for any pair of processes, the *acq_sent* and *acq_rcvd* logs are perfectly identical and reside on distinct nodes. As a result, a process does not actually need to save them in stable storage, since in case of failure they can be entirely restored from peer processes.

The support for barrier synchronization replay is similar.

4.4.3 Support for Replay of Shared Memory Accesses

Every process checkpoints pages for which it is the home. Every writer (including the home process itself) logs the diffs it produces for all pages it writes to. Accesses to a page are replayed during recovery by locally and dynamically applying an ordered sequence of logged diffs to the page to obtain a copy which is coherent according to the protocol semantics.

In HLRC, a process P_i where a page p is invalid *needs* a version $p.v^N$, according to write notices received up to its first attempted access to p after the invalidation. This is the minimal version of page p that P_i will request from $H(p)$ on its access and miss to p. During normal operation, $H(p)$ may reply with the requested version or with a higher version of p. In the latter case, HLRC guarantees that the copy of p at $H(p)$ incorporates only concurrent writes which are not conflicting with the faulting access by P_i (i.e., they did not happen-before according to the partial order enforced by synchronization operations). For this reason, during replay, it is sufficient for the access to p to be serviced with the minimal version of p, i.e., which contains only writes that "happened-before" the faulting access. As a result, changes in contents of a page after a miss and fetch from its home need not be reproduced exactly during replay, as it would be the case if pure message logging of page transfers were used.

In our system a page p is only checkpointed by its home $H(p)$. The home retains a sequence p_{ckp} of copies of p from past checkpoints. Any writer P_j of p logs every diff it creates in a per-page log *diff_log(p)*. The logged *diff* entry is stamped with its vector timestamp T_j, referred to as *diff.T*. During re-execution, a recovering process replays only the minimal changes of p after a miss by emulating the operation of $H(p)$. It maintains an evolving copy of p built from a *starting copy* p_0 obtained from $H(p)$, to which it dynamically applies partially ordered diffs obtained from all writers' logs *diff_log(p)*.

The following Lemma provides safety conditions for a *maximal* starting copy p_0 (i.e., to which nothing can be added without potentially compromising correctness of the recovery). It also gives conditions on the diffs needed for a correct replay of accesses

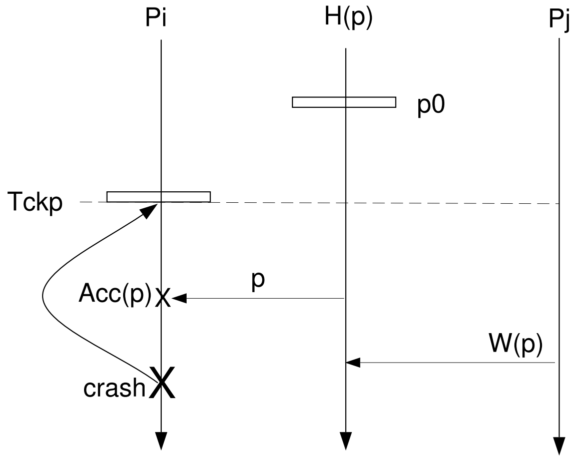


Figure 4.2: *Bounds for a checkpointed version p_0 of page p which is safe for replaying an access of P_i after crash and restart.*

to p starting from the maximal p_0. In the following, we define the $\leq$ relation on a pair of vectors to yield true iff the component-wise $\leq$ relationship holds for all elements.

Lemma 1 (Maximal p_0) *1. If page p is needed for the replay of P_i starting from a checkpoint timestamped with T^i_{ckp}, then a safe $p_0 \in p_{ckp}$ must have a version $p_0.v \leq T^i_{ckp}$.*

2. To ensure the correct replay of p starting from the above p_0, any writer P_j must supply diffs with timestamps diff.$T[j] > p_0.v[j]$.

Proof.

We look at the first access Acc(p) of P_i to page p after a crash and restart from its latest checkpoint timestamped T^i_{ckp}, as shown in Figure 4.2. (A diff sent by a writer of p to home $H(p)$ is represented by an arrow pointing towards $H(p)$; a fetch of p from $H(p)$ is shown as an arrow pointing away from $H(p)$.)

1. Regardless of the memory consistency model, a *safe* p_0 for P_i's access is always one that does not contain *any* writes to p that occurred in the future with respect to the restart checkpoint of P_i. In Figure 4.2, $W(p)$ is such a write. In LRC terms, a future conflicting write $W(p)$ from a process P_j could only occur in some interval $T_j[j] > T^i_{ckp}[j]$, generating a version with $p.v[j] > T^i_{ckp}[j]$. Therefore, a page with $p_0.v \leq T^i_{ckp}$ is always safe for replay [1].

[1]Note that under LRC the condition $p_0.v \leq T^i_{ckp}$ may not be strictly necessary. A safe p_0 can be

2. Next, we show that if we select p_0 from the *last* checkpoint of $H(p)$ in which the version $p_0.v \leq T^i_{ckp}$, then this p_0 is *sufficient* for a correct replay of Acc(p), under the conditions of Part 2 of the Lemma. Note that, according to Part 1 of the Lemma, such a p_0 would be *safe* for replay.

We prove p_0 is sufficient for two cases, depending on whether, during normal operation, p was valid (*Case A*) or not (*Case B*) at the time of Acc(p).

Case A. Suppose p was invalid at Acc(p) during normal operation, so it is also invalid after the restart of P_i. Then p's page table entry in the checkpoint records $p.v^N$, the *minimal* version that P_i will need on Acc(p) during recovery. This version is minimal in the sense that servicing Acc(p) with a page that has a lower version (missing some previous writes) would compromise the correctness of the protocol.

We consider two instances, depending on whether the checkpointed p_0 incorporates all the writes needed for P_i's access or not, i.e., whether either $p.v^N \leq p_0.v$ or there exists j such that $p_0.v[j] < p.v^N[j]$.

Case A.1. The checkpointed p_0 contains all writes needed by Acc(p), i.e., $p.v^N \leq p_0.v$. This case is trivial, in the sense that p_0 alone is sufficient for replay.

Figure 4.3 shows a simplified scenario in which P_i first synchronizes with P_j, resulting in the invalidation of p. The dashed arrows between P_j and P_i represent the transfer of write notices at synchronization operations that protect conflicting accesses to p.

For p_0 alone to be sufficient during replay, it must contain the last write $W_{bef}(p)$ of P_j that happened before Acc(p): $W_{bef}(p) \prec Acc(p)$. Note that $W_{bef}(p)$ occurs at P_j before the moment of the invalidation, due to LRC constraints.

Note that if, for some j, $p.v^N[j] < p_0.v[j]$, the higher version of $p_0.v[j]$ corresponds to writes of P_j not related to Acc(p), and included in p_0 after $W_{bef}(p)$. The safety condition from Part 1 of the Lemma ($p_0.v[j] \leq T^i_{ckp}[j]$) always ensures that no conflicting writes are incorporated in p_0. Suppose for example that the *first* access conflicting with Acc(p) that occurs after it in the $\prec$ order is a write $W_{aft}(p)$, also performed by P_j

any copy of p containing writes $W(p)$ that occurred after T^i_{ckp}, but which are not related to Acc(p) under the $\prec$ order.

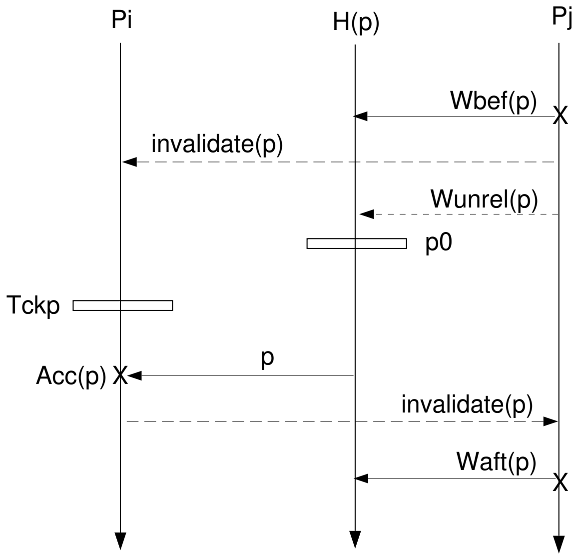


Figure 4.3: *Case A.1: Page p invalid in P_i's checkpoint and the last safe p_0 checkpointed at H(p) contains all writes needed by the access Acc(p) of P_i.*

($Acc(p) \prec W_{aft}(p)$ in Figure 4.3). Then the writes of P_j (labeled $W_{unrel}(p)$ in Figure 4.3) between the invalidation of p and the moment of W_{aft} are concurrent with Acc(p) but not related to it. Some of them may be included in the checkpointed copy p_0 of $H(p)$, as shown in the figure, making $p_0.v[j]$ larger than $p.v^N[j]$, but this would not affect the correct replay of Acc(p).

Case A.2. The checkpointed p_0 does not contain all related writes needed by Acc(p), i.e., there exists j such that $p_0.v[j] < p.v^N[j]$.

This means (see for example Figure 4.4) that there might exist conflicting writes W_{bef} previous to Acc(p) ($W_{bef} \prec Acc(p)$) that occurred prior to invalidation but after $H(p)$ took its checkpoint of p_0. Such writes would not have been captured in the checkpointed p_0, which makes p_0 alone unusable for replay (in our example, we would have $p_0.v[j] < p.v^N[j]$). To obtain the version $p.v^N$ needed for the replay of Acc(p), the recovering P_i must apply to p_0 writes like W_{bef}, logged by P_j as diffs. These writes would have occurred at logical times $T_j[j] > p_0.v[j]$, so P_j must supply all diffs with a timestamp $diff.T[j] > p_0.v[j]$.

Case B. Suppose p was valid at the moment of Acc(p) during normal operation. Since Acc(p) is the first access after the checkpoint, p was also valid when P_i took

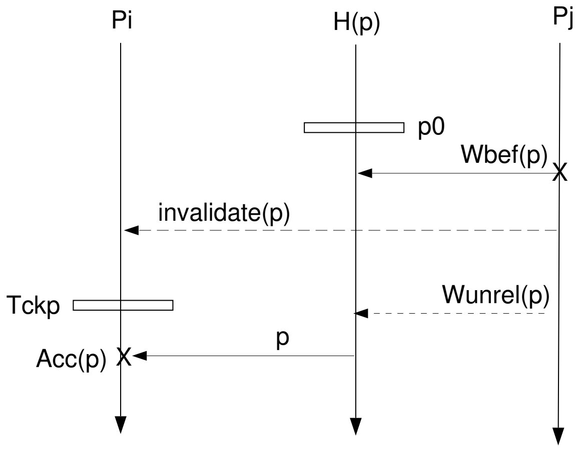


Figure 4.4: *Case A.2: Page p invalid in P_i's checkpoint and the last safe p_0 checkpointed at H(p) does not contain all writes needed by the access Acc(p) of P_i.*

the checkpoint. Then there must have been an access of P_i to p after it received the invalidation for p but before its last checkpoint at T^i_{ckp}. Suppose that such an access was a read $R(p)$, and that there was a write $W_{own}(p)$ of P_i such that $R(p) \prec W_{own}(p) \prec T^i_{ckp} \prec Acc(p)$ (Figure 4.5).

With respect to some conflicting write W_{bef} of some process P_j for which $W_{bef} \prec W_{own}$, the replay of Acc(p) is correct, under the conditions proved above in Case A.

We must only ensure that the replay of Acc(p) is correct with respect to all the writes like W_{own} issued by P_i. The starting p_0 alone cannot be used for replay since it may not contain W_{own}. For correct replay, P_i must save diffs for its writes that it created and logged after $H(p)$ took its checkpoint of page p_0, i.e., diffs with $diff.T[i] > p_0.v[i]$.

□

Note that Lemma 1 links the starting page p_0 with its corresponding diffs: if p_0 would be selected from an earlier checkpoint, then writers would have to keep more diffs for the correct replay of accesses to p.

4.5 Garbage Collection of Recovery State

In this section we prove the core results that ensure correct recovery of any process while logs and checkpoints are discarded to reclaim space. Our approach relies on checkpoint timestamping as described in Section 4.4 and allows for laziness in garbage collection

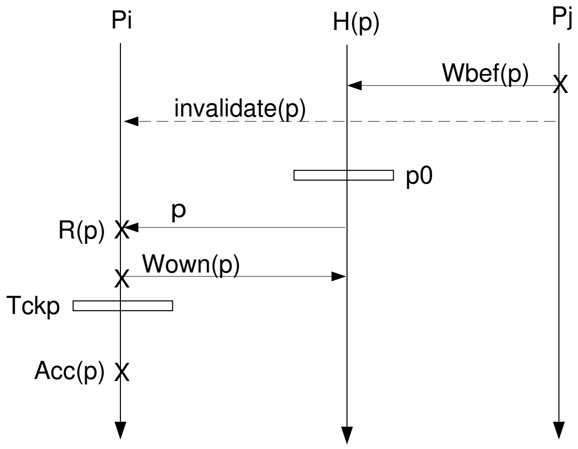


Figure 4.5: *Case B: Page p valid at the time of P_i's checkpoint. Local write $W_{own}(p)$ not captured by p_0 must be replayed.*

operations. The mechanisms we propose allow a process to locally initiate and execute these operations without involving other processes.

4.5.1 Checkpoint Timestamping Issues

Recall that in our system the most recent checkpoint (the restart checkpoint) of a process P_i has a timestamp vector T^i_{ckp}, equal to the vector timestamp T_i at the moment the checkpoint is taken. In a preliminary work [89] we showed how the global vector time T_g (or, in practice, an approximation of T_g that preserves a total order on the set of all checkpoint timestamps) can be used to timestamp checkpoints and perform garbage collection of recovery state. We proved a set of rules for log trimming and checkpoint garbage collection assuming (ideal) instant availability of the global vector time for timestamping checkpoints. In this dissertation, we prove a set of garbage collection rules for the more practical case where a process uses its local vector timestamp to timestamp its checkpoint, i.e., $T^i_{ckp} = T_i$ at the time of checkpoint.

We note that, in practice, each P_i maintains, for any P_j, its last known value $\hat{T}^j_{ckp}$ of P_j's checkpoint timestamp T^j_{ckp}. The policy of propagating the T^j_{ckp} of the last checkpoint from P_j to other nodes is flexible: it can be broadcast periodically, sent in reply to a query, lazily piggybacked on protocol messages, etc. In this work, we are not concerned with proposing such a policy or analyzing its impact. The important thing to note, however, is that our log trimming and garbage collection algorithms are robust with respect to checkpoint timestamps (therefore do not depend on the timestamp propagation method): *(i)* their correctness is not impeded by stale timestamps, and *(ii)* the known value of a given checkpoint timestamp does not need to be consistent across processes (so the algorithms do not require processes to synchronize). In Section 4.7 we show that our scheme achieves good performance using lazy propagation through piggybacking on protocol messages.

4.5.2 Synchronization Log Trimming

The next two Lemmas provide bounds on intervals corresponding to the oldest entries that a process must retain from its write notice and acquire logs, in order to be able to support the recovery of another process.

Lemma 2 (Write notice log trimming) *A process P_i can support the wn replay of any process P_j restarting from a checkpoint timestamped T^j_{ckp} by retaining only entries of wn_log corresponding to intervals starting from ckp_int* $= \min_{j \neq i} T^j_{ckp}[i] + 1$.

Proof. During re-execution, a process P_j will need *only* write notices generated by P_i that it had received after taking its last checkpoint. At the time that checkpoint was taken, the i^{th} element of P_j's checkpoint timestamp (which is its vector timestamp), $T^j_{ckp}[i]$, recorded the last interval for which write notices had been received from P_i. Therefore, to support the execution replay of P_j, P_i must retain only write notices in intervals larger than $T^j_{ckp}[i]$. The minimum of this value over all P_j is the oldest interval from P_i received by any other process after its restart checkpoint. Hence, to cover recovery of any process, P_i must retain write notices from later intervals, i.e., those with a logical time at least equal to *ckp_int*.

□

As noted before, since the $\hat{T}^j_{ckp}$ vectors available at P_i may not be consistent with the real checkpoint timestamps, the local estimate of *ckp_int* may not be exact. If this is the case, it defines a superset of the minimal set of intervals needed to support recovery at any node, since it is computed as a min over logical time (monotonically increasing).

Lemma 3 (ACQ log trimming) *A process P_i can support the ACQ replay of a process P_j restarting from a checkpoint with timestamp T^j_{ckp} by retaining only entries of $acq\_sent\_log[j]$ with $T_j[j] > T^j_{ckp}[j]$, and it can restore the strictly needed portion of the $acq\_sent\_log[i]$ of P_j by retaining only entries of $acq\_rcvd\_log[j]$ with $T_i[i] > T^i_{ckp}[i]$.*

Proof. For its *ACQ* replay, a recovering P_j will only need from some P_i entries of $acq\_sent\_log[j]$ logged for acquires that P_j has executed after its last checkpoint. Therefore P_i can safely trim $acq\_sent\_log[j]$ by discarding entries with $T_j[j] \leq T^j_{ckp}[j]$ (recall that the checkpoint timestamp is equal to the vector timestamp at the moment the checkpoint was taken).

In order to recover its $acq\_sent\_log[i]$, a recovering P_j needs from the acquirer P_i entries of P_i's $acq\_rcvd\_log[j]$. Note that P_j needs to recover only the portion of $acq\_sent\_log[i]$ that would be strictly needed for a *potential ACQ* replay of P_i, in case a crash of P_i will occur sometime in the future. This portion of P_j's log consists of entries with $T_i[i] > T^i_{ckp}[i]$. To back it up, P_i must retain from $acq\_rcvd\_log[j]$ only entries for which $T_i[i] > T^i_{ckp}[i]$.

□

In practice, because the checkpoint timestamps $\hat{T}^j_{ckp}$ known to P_i may lag behind the timestamps T^j_{ckp} of the most recent checkpoints, trimming of $acq\_sent\_log[j]$ may not be optimal. Trimming of $acq\_rcvd\_log[j]$ is always optimal, as it uses only the local checkpoint timestamp.

4.5.3 Lazy Log Trimming and Checkpoint Garbage Collection

In the following, we establish the fundamental result on which our algorithms for Checkpoint Garbage Collection (CGC) and Lazy Log Trimming (LLT) are based.

Lemma 1 proved conditions for a home $H(p)$ to retain a checkpointed copy p_0 of a page p needed for recovery of one process in the system. Note that the selection of the checkpoint depends on the particular page itself (it involves individual page versions).

Also note that, for efficiency reasons, a home node H may choose to keep a *window* of past checkpoints of all pages it manages, starting with the oldest checkpoint determined by Lemma 1.

The following Theorem simply extends Lemma 1 to provide conditions for the garbage collection of checkpointed copies of a page that are not needed for recovery of any process (CGC). It also shows that after $H(p)$ performs a CGC operation a writer can, at a later moment and decoupled from the CGC operation, *lazily* trim its diff logs using information from $H(p)$, namely the version of p_0 (LLT).

Theorem 1 (CGC and LLT) *1. A home process H can support the page replay of any process if it retains page p_0 from a checkpoint such that $p_0.v \leq T_{min} = \min_{j \neq H} T^j_{ckp}$.*

2. A writer of page p, P_i, can support recovery replay for p by retaining only diff_log(p) entries with diff.$T[i] > p_0.v[i]$.

Proof.

Directly from Lemma 1 by considering, for complete recovery support, processes $P_j, j \neq H$.

□

Note that the bounds for retaining checkpoints and diff logs in Theorem 1 (T_{min} and, indirectly, $p_0.v$) depend on the checkpoint timestamps T^j_{ckp} of the restart checkpoints. In practice, the efficiency of CGC and LLT depends on the $\hat{T}^j_{ckp}$ timestamps available at P_i, which could be stale with respect to the most recent checkpoints of peer processes P_j.

Aggregate page management for LLT and CGC

Lemma 1 and its proof implicitly assume that management of pages by homes is done on a per-page basis. This represents the optimal case, when a home process retains for *each* page exactly the restart version. As a result, for different pages, their restart versions may come from different checkpoints. In practice, such an implementation would have to selectively discard unneeded pages from checkpoints.

In practice, an implementation may trade optimal storage consumption for simplicity and efficiency, and it may choose to discard a whole checkpoint only when none of its pages is needed, instead of selectively discarding per-page checkpointed copies.

To simplify such an approach, *aggregate page management* is possible by using the *page version timestamp* V_H of a home process H. Any home process H records in the checkpoint a *page version timestamp* V_H, representing the largest diff versions that H has received from all processes, i.e., $V[i] = \max_p p.v[i]$ over all pages p homed by H.

The following versions of Lemma 1 and Theorem 1 establish the use of a single vector V_H for tracking dependencies between restart checkpoints and page checkpoints to be retained by a home process, and between home checkpoints and diffs to be retained by page writers.

Lemma 4 (Maximal p_0 for aggregate management) *1. If page p is needed for the replay of P_i starting from a checkpoint timestamped with T^i_{ckp}, then $p_0 \in p_{ckp}$ can come from the last (i.e., most recent) checkpoint of H(p) for which $V_H \leq T^i_{ckp}$.*

2. To ensure the correct replay of p starting from the above p_0 retained by home H, any writer P_j must supply diffs with timestamps diff.$T[j] > V_H[j]$.

Proof.

1. In Part 1 of Lemma 1, since V_H is an absolute bound on all page versions in a checkpoint, we have $p_0.v \leq V_H$. Therefore, the *safety* condition $p_0.v \leq T^i_{ckp}$ for the checkpoint from which p_0 is selected is automatically satisfied by a checkpoint with $V_H \leq T^i_{ckp}$.

2. Because the vector V_H tracks the latest diffs received by home H from all processes by the time of the checkpoint, any future (i.e., occurring after the checkpoint was taken) writes by some writer P_j can only generate diffs with a timestamp $diff.T[j] > V_H[j]$. Therefore, throughout the proof of Part 2 of Lemma 1, $V_H[j]$ can safely replace the per-page bound $p_0.v[j]$ for diffs to be retained by P_j.

□

Extending the previous Lemma, the following theorem describes how garbage collection of whole page checkpoints and diff log trimming can be performed at a process by using the aggregate page version timestamp V_H instead of individual page versions.

Theorem 2 (Aggregate CGC and LLT) *1. A home process H can support the page replay of any process if it retains whole page checkpoints starting with the most recent checkpoint having a page version timestamp V_H such that $V_H \leq T_{min} = \min_{j \neq H} T^j_{ckp}$.*

2. A writer of page p, P_i, can support recovery replay for p by retaining only diff_log(p) entries with diff.$T[i] > V_{H(p)}[i]$.

An important observation on results proved in Theorems 1, 2 is that approximate values of the checkpoint timestamps $\hat{T}^j_{ckp} \leq T^j_{ckp}$ can be tolerated in the bounds T_{min} on $p_0.v$ and V_H, respectively. For example, in Theorem 1 the approximate $\hat{p}_0.v \leq \hat{T}_{min} \leq T_{min}$ can only push p_0 to a checkpoint older than the ideal (optimal) checkpoint computed based on the exact T_{min}. If an older checkpoint with $\hat{p}_0.v < p_0.v$ is retained, Part 2 of Lemma 1 ensures that p can still be reconstructed by retaining necessary diffs for the older approximation of p_0.

4.5.4 An Example

Figure 4.6 shows an example of how whole checkpoints are garbage collected and diff logs are trimmed using Theorem 2 (a similar example can be devised using Theorem 1 for a given page). For ease of presentation, assume that checkpoint timestamps in the example are such that they can be totally ordered about an imaginary global time axis running vertically, as shown in the figure. Vertical brackets mark the *window* of checkpoints from which page copies are to be retained by each home. The upper limit of a checkpoint window is set by aggregate page management using page version timestamps of Theorem 2.

In this example, process P_3 is about to take checkpoint c_{34}, at which time it also performs a CGC operation. The solid and dotted arrows show the dependency (from T_{min} to the first checkpoint to be retained by P_3, using its page version timestamps V_3) that sets the upper limit of P_3's checkpoint window at the moments when c_{34} and c_{33}, respectively, are taken. The dotted bracket represents the checkpoint window of P_3 at the time it took its previous checkpoint c_{33}. When taking c_{34}, the upper limit of

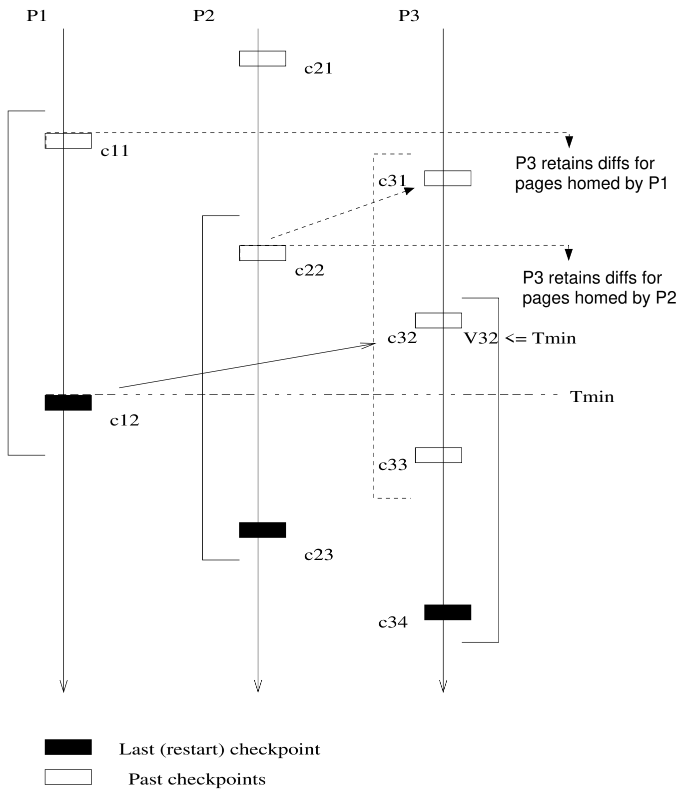


Figure 4.6: *Determining the checkpoint window for CGC and the diff log bounds for LLT by process P_3 at the time it takes c_{34}.*

P_3's window will become c_{32}, since Theorem 2 enforces a dependency from c_{12} (which determines the bound T_{min}) to c_{32}. The previous limit had been c_{31}, as set by the now obsolete dependency from c_{22} at the time c_{33} was taken. In effect, the window advances by including the new c_{34} and dropping the unneeded c_{31}. As a diff producer, P_3 trims its diff logs by retaining only diffs needed for pages homed by P_1 and P_2 if a replay would use pages from the first checkpoints in their windows (c_{11} and c_{22}, respectively). Earlier entries can be discarded. As noted before, trimming of diff logs can be performed lazily, when P_3 learns the page version timestamps of c_{11} and c_{22}.

4.5.5 Computing Trimming Bounds Using Approximate Information

Lemmas 2, 3 and Theorems 1, 2 prove ideal bounds for how each process can trim all unnecessary entries from its logs. A process P_i needs from all other processes P_j: *(i)* the checkpoint timestamp T^j_{ckp} of the restart checkpoint, for trimming the wn and ACQ logs and for its CGC, and *(ii)* $p_0.v[i]$ (or $V_j[i]$), if P_i is a writer to a page homed by P_j, for its LLT.

One approach to distributing this information would be to propagate it lazily, for example piggybacked on protocol messages: a process P_i would only have to send to P_j a vector T^i_{ckp} and one integer (a per-page version, $p_0.v[j]$, or the aggregate value $V_i[j]$). Piggybacking, or any other form of lazy propagation, may make this information stale at a particular process. The corresponding values that P_j uses for trimming logs ($\hat{T}^i_{ckp}[j]$, $\hat{T}^i_{ckp}[i]$, and $\hat{p}_0.v[j]$ or $\hat{V}_i[j]$) may become obsolete, depending on the sharing/communication pattern. The logs of write notices and the synchronization logs will be trimmed less efficiently by P_j if $T^i_{ckp}[j]$ and $T^i_{ckp}[i]$ are stale (Lemmas 2 and 3). For the page checkpoints, the bound $\hat{T}_{min}$ may be less than T_{min} of Theorems 1, 2 because of stale $\hat{T}^i_{ckp} \leq T^i_{ckp}$, and it may force process P_j, as a home of some page p, to push back its checkpoint window to include an older checkpoint than actually needed. This may also increase the diff log size at some writer P_i by the feedback effect through $p_0.v[i]$ ($V_j[i]$, respectively).

4.6 Comparison with Other Fault-Tolerant DSM Systems

Much of the research in recoverable DSM systems using log-based recovery has focused on developing new logging schemes to reduce the high failure-free logging overhead (caused by the typically high communication frequency of DSM-based applications).

Richard and Singhal have proposed the idea of volatile logs of *accesses* to shared memory for a sequentially-consistent DSM [73]. Logs were flushed to stable storage on every page transfer, which, combined with the potentially large size of the logs, made the scheme very expensive [93].

Neves et al. have added sender-based logging [47] with independent checkpointing to an entry-consistent, single-writer, object based DSM system [64]. Object copies are logged to volatile memory after every update. The simple consistency model, which does not allow multiple writers, along with logging of full object copies makes garbage collection straightforward: all processes trim their object logs eagerly, whenever a new checkpoint is created in the system. To support this scheme, each process broadcasts control information needed for trimming after every checkpoint.

Feeley et al. have used log-based coherency, a mechanism based on transactional database techniques [34]. They applied the scheme to RVM [75], a persistent virtual-memory, to build a recoverable entry-consistent DSM that provides a transactional programming model. Propagation of transaction logs between nodes achieves coherency, while recoverability is ensured by the underlying persistent store to which committed logs are saved. Several methods for log trimming are discussed, including broadcasting control information, but only off-line trimming is performed. In contrast to the above two systems, our fault-tolerant system supports a relaxed multi-writer memory model at page granularity, performs on-line garbage collection, and does not require global operations for garbage collection.

In [93], Suri et al. were the first to use log-based recovery in a Lazy Release Consistent (LRC) [48] DSM. Their system uses independent checkpointing along with pessimistic logging by a receiver at synchronization points and access misses. To reduce the overhead of access to stable storage for each logged message, log entries are stored in volatile memory and flushed to stable storage before sending a message to another process. Log flushing takes place on the critical path and can be expensive if synchronization is frequent. Every process must log all the data it needs for recovery, which leads to unnecessary replication of state and wastes stable storage. Because the log is saved in stable storage and is only used for the recovery of the process that creates it, the problem of garbage collection can be easily addressed by taking a checkpoint and discarding the log when its size exceeds a limit. In our system, we also keep logs in volatile memory but only require that they are saved to stable storage with every checkpoint taken. We log only minimal state and do not replicate it across nodes. Our mechanisms for log trimming are totally decoupled from any policy that decides when trimming is to be performed, or when a checkpoint must be taken. In our system, garbage collection operations do not require a process to take a checkpoint.

In [27], Costa et al. have integrated log-based fault tolerance support into TreadMarks [49], a DSM system that implements the LRC model. Their work is different from ours in that their system leverages the global garbage collection phases of TreadMarks to take coordinated checkpoints. While they also use intermediate independent checkpoints and volatile logs to speed up recovery from single-node failures, the recovery is not entirely based on independent checkpoints, as it may need to use pages from the last global consistent checkpoint. Another difference is that all logs and past checkpoints can be discarded at a global checkpoint, so their system does not face the problem of dynamic log trimming and checkpoint garbage collection.

Kongmunvattana and Tzeng proposed a coherence-centric logging and recovery scheme for a HLRC protocol [52]. The logging scheme is based on that proposed in [93], but combines receiver-based with sender-based logging to eliminate page logging. Logs are flushed to stable storage at release points, which may generate frequent accesses to stable storage. No garbage collection mechanism is proposed. In our fault-tolerant HLRC system, logs can be saved to stable storage infrequently, only at checkpoint time. Home nodes retain copies of pages from a window of past independent checkpoints to support fast recovery of any failed process. We develop garbage collection algorithms that effectively bound the amount of state retained in the system for recovery support.

4.7 Prototype and Evaluation

To evaluate LLT and CGC, we have extended the HLRC protocol to include: *(i)* LLT, *(ii)* a checkpointing policy to decide when to checkpoint at each node, and *(iii)* a simulation of CGC[2]. Our goals are to: *(i)* evaluate the impact of checkpointing and logging on application performance, and *(ii)* assess the efficiency of LLT and CGC in terms of both volatile and stable storage requirements, using as metrics the diff log size and the checkpoint window size.

In all experiments, log trimming, garbage collection of checkpoints and saving logs to stable storage take place only at checkpoint time. This is a limitation that we impose in order to stress the system to the greatest extent. Since LLT and CGC mechanisms

[2]The prototype used in our experiments generates checkpoints to stable storage but does not implement recovery and checkpoint storage management. Nevertheless, this implementation is sufficient to evaluate CGC on real-world applications.

Application and problem size	*Shared memory (MB)*	*Base execution time (s)*
Barnes 256 k	43	1,663
Water-Nsquared 19,683	12.6	1,634
Water-Spatial 256 k	257.3	2,569

Table 4.1: *Applications used and their characteristics.*

Application and problem size	*HLRC traffic (MB)*	*CGC traffic (MB)*	*% overhead*
Barnes 256 k	2, 224	3.3	0.15
Water-Nsq. 19,683	68.5	0.1	0.2
Water-Sp. 256 k	174.7	0.5	0.25

Table 4.2: *Message traffic overhead of CGC and LLT in a HLRC DSM system.*

are totally decoupled from checkpointing operations and can run at any point during execution, a more aggressive implementation could asynchronously write logs to stable storage, overlapping saving logs with computation. In our experiments, writing logs to stable storage was measured as added overhead.

We have selected three applications from the SPLASH-2 benchmark suite [99], Barnes, Water-Nsquared, and Water-Spatial, to drive our prototype system. We chose these applications because they have the longest running times in the suite, have various memory requirements, have a fair amount of synchronization, and generate large volumes of diffs, therefore presenting the worst-case scenario for LLT. We had initially run our experiments on all programs in the SPLASH-2 suite, and eliminated those for which we could not scale the running times or that were generating small logs or no logs at all. They are irrelevant to our log garbage collecting scheme. We modified Barnes to run for 60 steps, while for the other applications we used the default parameters in the benchmark, except for increasing problem size. Table 4.1 shows the shared memory footprint of each application and the execution times with the base protocol (without fault tolerance support).

Application and problem size	*L %*	*Ckp. taken*	*Execution time (s)*	*Execution time % increase*	*Logging (s)*	*Disk write (s)*	*% overhead*
Barnes 256 k	100	6 - 10	2,677	61	16.2	96.8	6.8
Water-Nsq. 19,683	10	9	1,644	0.6	0.9	5.3	0.4
Water-Sp. 256 k	10	5	2,737	7	13	79.4	3.6

Table 4.3: *Performance of the independent checkpointing scheme with CGC and LLT in a HLRC DSM system. The last three columns show the time spent logging and writing to disk in absolute value, and as overhead relative to the base execution time.*

Application and problem size	W_{max}	*Max log (MB)*	*Disk traffic (MB)*	*Logs (MB)*	*Saved logs (MB)*	*Saved logs %*	*Discarded logs (MB)*	*Discarded logs %*
Barnes 256 k	3	80	390.2	371.2	348.3	94	281	76
Water-Nsq. 19,683	3	3.5	28.4	14.2	14.2	100	11.4	80
Water-Sp. 256 k	3	75.7	339.2	178.3	178.3	100	102.9	58

Table 4.4: *Overall efficiency of CGC and LLT in an HLRC DSM system. Disk traffic is generated by checkpointing homed pages and saving logs after in-memory trimming.*

The checkpointing policy used in the experiments was based on limiting the log size in volatile memory: a checkpoint is taken when the size of volatile logs exceeds a threshold expressed as a fraction L of the application's shared memory footprint. The checkpointing decision is independently made by each node, and is transparent to the application.

All experiments were run on a cluster of eight 300 MHz Pentium II PCs with 512 MB of memory each, running Linux 2.0.30. Communication is performed over a Myrinet LAN [12], using user-level virtual memory-mapped communication (VMMC) [30], to which we added fault tolerance support. Logs and checkpoints were saved to the local disk. We consider only the diff logs for trimming, as they consume the largest amount of space. Results are for one run of each application with data averaged over all nodes.

4.7.1 Performance with the Fault-Tolerant Home-Based Lazy Release Consistency Protocol

Table 4.2 shows that the message traffic overhead of the LLT/CGC control data piggy-backed on protocol messages is negligible relative to the base protocol traffic. Table 4.3 shows the impact of integrating support for recovery of the DSM shared space on application performance. For each application, with logging and checkpointing enabled, we show: *(i)* the memory usage threshold L allowed for logs by the log-overflow checkpointing policy (as a percentage of the shared memory size), *(ii)* number of checkpoints taken, *(iii)* execution time, *(iv)* percentage of its increase over the base execution time, *(v)* overhead of logging, *(vi)* overhead of writing to stable storage, and *(vii)* overhead over the base execution time represented by the time to log and write to disk.

To estimate the overhead of writing to stable storage, at every checkpoint we write

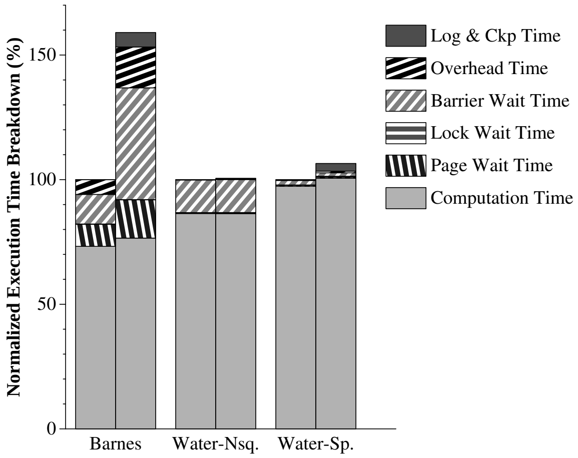


Figure 4.7: *The normalized execution time breakdown for the base protocol (left bar) compared with a failure-free execution with checkpointing and logging enabled (right bar).*

to the local disk the pages homed by a process, along with its volatile logs. This accounts for the direct overhead introduced by fault tolerance support in the DSM system. We chose to only assess the overhead of checkpointing the DSM space, as other components of the checkpointing overhead (e.g., checkpointing private data of a process) are unavoidable and therefore present in any fault-tolerant system.

Table 4.3 shows that the time spent with DSM logging and checkpointing is fairly small in all cases. This is also reflected by the small increase in running time, except for Barnes, which suffers an increase of about 60%. To see where this severe degradation in performance comes from, we measured various components of the execution time both for the base protocol and with logging and checkpointing enabled.

Figure 4.7 compares the averaged breakdown of execution time when running with base HLRC (left bar), and with fault-tolerant HLRC (right bar). The graphs are normalized with respect to the base execution times. The measured overheads are: time spent waiting for pages from home nodes, time spent waiting for locks, time spent waiting at barriers, protocol overhead time (including execution of page fault and message handlers, and of synchronization primitives), and time spent with logging and checkpointing DSM state.

For Barnes, we see that barrier waiting times have the largest contribution to the performance degradation observed, increasing from 12% to 28% of the execution time.

The reason is an imbalance in the distribution of homed pages and diffs (and therefore logs) generated across nodes: in our sample run the volume of logs created varied from 290 to 460 MB across nodes. With a log-overflow checkpointing policy, this imbalance spreads checkpoints unevenly over time across processes. Since Barnes has many barriers, it is likely that processes checkpoint in *different* intervals delimited by consecutive barriers. Therefore, the overhead incurred by only one node or a small set of nodes checkpointing in some interval adds to the barrier waiting time of all other processes. The reason is that the checkpointing policy used is unaware (and does not take advantage) of the intense global synchronization in the application. A checkpointing policy that forces all processes to checkpoint in the same barrier intervals would be better suited for applications with a large number of barriers. For the other applications, which have less barriers and have regular access and update patterns, this effect is not visible.

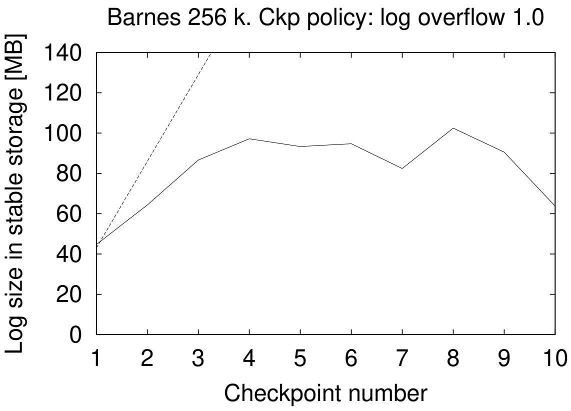
4.7.2 Efficiency of Checkpoint Garbage Collection and Lazy Log Trimming

Table 4.4 shows how efficient CGC and LLT are in limiting the amount of state in stable storage. We measured, per-node: *(i)* maximum size of the checkpoint window (W_{max}), *(ii)* maximum amount of stable storage consumed by diff logs, *(iii)* total amount of checkpointed pages and diff logs written to stable storage, *(iv)* size of volatile logs created during execution, *(v)* size of volatile logs saved in stable storage, *(vi)* percentage of saved logs from the volatile logs created, *(vii)* size of logs discarded as a result of LLT, and *(viii)* percentage of discarded logs from volatile logs created.

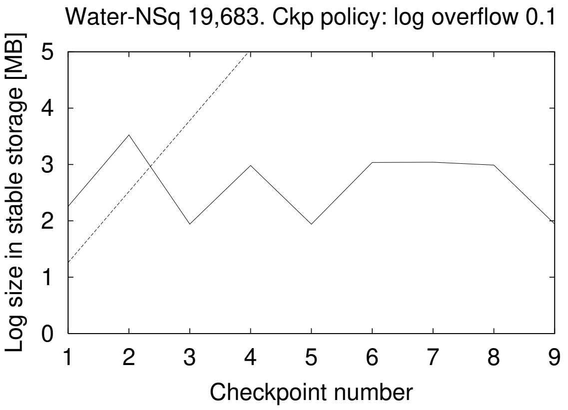
The checkpoint window size is at most three, showing that CGC is effective even with the lazy propagation of information. The good overall efficiency of our LLT scheme can be seen from the large fraction of logs discarded. The smaller value in the case of Water-Nsquare is only due to the small number of checkpoints taken. Note that almost no trimming is done in memory. This means that most of the time the home nodes are forced to retain at least two checkpoints, otherwise they could just discard their volatile logs at the time they take a new checkpoint (this assumes homes write to their homed pages, which is the case with all three applications).

Figure 4.8 plots the variation of diff log size on stable storage against checkpoints for a single node, showing that our scheme effectively bounds the log size over time. The straight line in each graph represents the theoretical growth of the log without LLT. In our implementation, sampling the size of the log takes place only at synchronization points in the application, as LLT is completely transparent. Due to this imprecision, the actual log size suffers from a minor start-up effect: with the first checkpoint, more than the threshold size of logs might be saved. However, as it can be seen with Water-Nsquared and Water-Spatial, this effect is quickly canceled out by the effective trimming and within three checkpoints the total log size falls under the theoretical unbounded increase without LLT.

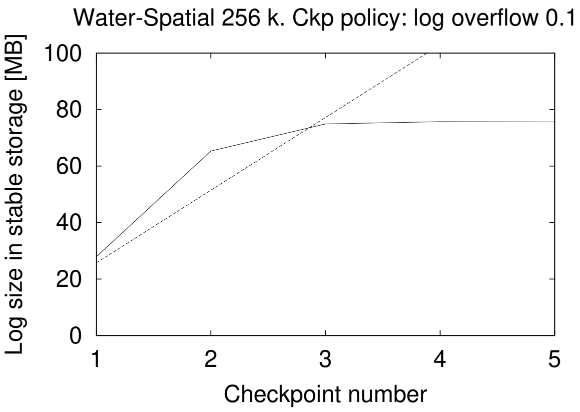
Water-Spatial is an interesting case, as it is very regular, and the volatile log-size threshold policy forces checkpoints in every iteration. The regularity effect can be observed in Figure 4.8 in the gradual build-up of the saved log in the first two checkpoints, which retains the necessary state for recovery of the last iteration. After this point, every iteration adds and discards about the same amount of logs to/from stable storage. This behavior is expected, as in the first iteration nodes perform computations and generate diffs for pages they become home for on the first access. Some trimming is performed at the second checkpoint, as writers start learning about the checkpoint windows of home nodes. In the next iteration, the trimming information has reached all writers and, starting from the third checkpoint on, diffs which are no longer needed for recovery of one past iteration are massively discarded at every new checkpoint, causing the curve to flatten out. This behavior exhibits a self-synchronizing effect of the independent checkpoints in the case of Water-Spatial, without any explicit checkpoint coordination.



(a)



(b)



(c)

Figure 4.8: *Dynamics of log size in stable storage for Barnes (a), Water-Nsquared (b) and Water-Spatial (c). Logs are sampled at checkpoint time and discrete points are connected to exhibit the trend under LLT control. Straight dotted lines show the unbounded growth the log would have in the absence of LLT.*

4.8 Summary

We have described the design and implementation of a fault-tolerant DSM protocol based on independent checkpointing and logging using lazy garbage collection of recovery state.

In our system, garbage collection of recovery state (checkpoints and logs) relies on lazy propagation of control information and operates lazily with respect to checkpointing and logging operations. We have developed two novel algorithms for lazy garbage collection of recovery state that do not require any form of coordination between nodes. We have demonstrated through experiments with applications from the SPLASH-2 suite that logging and ckeckpointing operations in our protocol have low overhead, and that our lazy garbage collection algorithms effectively bound the amount of recovery state (log size and checkpoint window size) retained in the system.

Chapter 5

Conclusions and Directions for Future Work

Replication of the live state of a machine has long been used for fault tolerance and availability. Yet, despite of its well-understood theoretical foundations and the success of commercial systems like Tandem [9] and TARGON 32 [14], dependable computer systems have yet to become a commodity. One of the reasons is that eager replication of computation state may incur high performance penalties and requires extra dedicated hardware which makes it costly.

In this dissertation, we have explored the potential of lazy propagation of computation state in implementing system support for service availability and fault tolerance. We have designed and implemented OS abstractions that enable lazy propagation or extraction of state from a system by extending general-purpose operating systems, and have demonstrated through experiments with microbenchmarks and real applications that lazy state propagation can be successfully used to provide such support. With Service Continuations and Migratory TCP, we have shown that lazy and dynamic session migration can achieve increased end-to-end Internet service availability with negligible performance overhead on an existing system. With Backdoors, we have shown that lazy recovery/repair healing operations can be performed from another system, even after a failure of the OS of the target system, with no or negligible overhead during failure-free execution. With our fault-tolerant DSM system, we have shown that lazy state propagation and lazy garbage collection of recovery state can be used to build efficient fault tolerant systems based only on independent checkpointing and volatile memory logging. All the systems described in this dissertation have been implemented as extensions to commodity operating systems and use only commodity, off-the-shelf hardware.

One of the features that makes lazy state propagation appealing is that it does not require dedicated machines to implement it. While both eager and lazy state propagation always require a "destination" system to reincarnate the state migrated or extracted from an "origin" host system, the difference is that with lazy propagation the transfer takes place only once, when actually needed, freeing the destination machine for other tasks. In contrast, as experience of other systems has shown, in seeking to maintain an exact full-state replica of a machine or a subsystem at all times, eager propagation exacts high performance penalties because it requires interposition on fine-grained system interactions.

On the other hand, the fundamental drawback of lazy propagation is that, in deferring state transfer until actually needed, it relies on the resources of the origin host being always available. Our Backdoors architecture for remote healing addresses exactly this problem: with the help of an intelligent network interface, it can perform lazy recovery/repair actions on the in-memory state of a system even after a failure or damage of its OS makes that system unavailable.

Since most of the time (if not in all cases) lazy state propagation lies on a critical path for the performance and responsiveness of the system, a crucial issue to its effectiveness and viability is the amount of state transferred lazily. With Service Continuations and Backdoors, we have demonstrated that, with careful design, it is possible to provide the right OS abstractions that allow a system or an application to define *light-weight* components of the logical state of a computation, protocol, etc. Both Service Continuations and Backdoors rely on lazy access to OS abstractions that encapsulate fine-grained OS and application level state.

The efficiency of the state transfer that is achievable with the systems we have developed raises another interesting question and a possible direction of future research: if, as this dissertation has shown, such light-weight OS abstractions can be defined, what are the conditions under which they can be used with pure eager propagation, or with hybrid eager-lazy mechanisms? In other words - given the right OS abstractions, *what is the operating range of eager/lazy approaches*? How could a system designer characterize a system so that one or the other can be selected, depending on certain performance metrics and tradeoffs?

In attempting an intuitive answer to these questions, given that both the volume and frequency of updates are critical to an eager approach, we envision a two-dimensional design space. On one dimension, *state size* favors lazy propagation towards the higher end of the spectrum and makes eager propagation interesting at the lower end. On the other dimension, *state dynamics* favors lazy propagation when the frequency of state changes is high and enables the eager propagation option if state updates have low frequency.

We note however that this is only a coarse division of the design space, and that at least one of the two dimensions is not linear. For example, as the state size increases, the transfer time may become prohibitive if pure lazy propagation is used. In this case, a hybrid approach may be better: instead of a bulk lazy transfer, the system may periodically perform eager transfers of summaries of updates to the state of interest (an approach frequently used in publisher-subscriber systems, for example).

The systems we have designed and the case studies we have presented address one point in the above design space, which is characterized by highly-dynamic state of fairly moderate size. In dynamically migrating clients sessions of an Internet service, or in salvaging service sessions from server node failures, lazy propagation was successful precisely because it can cope with the highly-dynamic state of such systems, which includes frequent changes driven by input/output on communication channels.

Once this design space and metrics for evaluating the lazy versus eager design tradeoffs can be defined, two immediate interesting questions for future research are: Can adaptive, hybrid eager-lazy mechanisms for fault tolerance and availability be developed that exploit variations in state dynamics and state size to achieve the best of both worlds? The second, related question is: What kind of OS abstractions would be needed to separate and manage state components based on how they are propagated in the system to achieve availability and fault tolerance? The answers to these questions may require an interdisciplinary approach involving operating system design, system modeling and machine learning techniques.

References

[1] "Infiniband Trade Association." http://www.infinibandta.org.

[2] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side TCP to Mask Connection Failures," in *Proc. IEEE INFOCOMM '01*, Apr. 2001.

[3] "Apache HTTP Server." http://httpd.apache.org.

[4] M. Aron, P. Druschel, and W. Zwaenepoel, "Efficient Support for P-HTTP in Cluster-Based Web Servers," in *Proc. USENIX '99*, 1999.

[5] S. Bailey and T. Talpey, "The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols ." IETF Draft, Jan. 2004.

[6] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems," in *Proc. 5th ASPLOS Conference*, Oct. 1992.

[7] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," in *Proc. Summer '92 USENIX*, 1992.

[8] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Future Generation Computer Systems*, vol. 13, no. 4–5, pp. 361–372, 1998.

[9] J. F. Bartlett, "A NonStop Kernel," in *Proc. 8th Symp. on Operating Systems Principles (SOSP)*, 1981.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proc. 15th Symp. on Operating Systems Principles (SOSP)*, Dec. 1995.

[11] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. Amorim, "Hiding Communication Latency and Coherence Overhead in Software DSMs," in *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, Oct. 1996.

[12] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.

[13] A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," in *Proc. 9th Symp. on Operating Systems Principles (SOSP)*, 1983.

[14] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault Tolerance under UNIX," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 1–24, 1989.

[15] T. C. Bressoud and F. B. Schneider, "Hypervisor-based Fault Tolerance," in *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[16] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-Backup Approach," in *Distributed Systems (2nd Ed.)*, ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216.

[17] G. Cabillic, G. Muller, and I. Puaut, "The Performance of Consistent Checkpointing in Distributed Shared Memory Systems," in *Proc. 14th Symposium on Reliable Distributed Systems (SRDS-14)*, Sept. 1995.

[18] G. Candea, "Personal communication," Feb. 2004.

[19] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1213–1225, Dec. 1998.

[20] J. Carter, J. Bennet, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proc. 13th Symposium on Operating Systems Principles (SOSP-13)*, Oct. 1991.

[21] J. B. Carter, A. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel, "Network Multicomputing Using Recoverable Distributed Shared Memory," in *Proc. IEEE International Conference CompCon '93*, Feb. 1993.

[22] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," in *Proc. 17th Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 2002.

[23] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[24] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott, "InterWeave: A Middleware System for Distributed Shared State," in *Proc. 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR 2000)*, May 2000.

[25] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," in *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[26] D. Clark and W. Fang, "Explicit Allocation of Best Effort Packet Delivery Service," *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, pp. 362–373, Aug. 1998.

[27] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro, "Lightweight Logging for Lazy Release Consistent Distributed Shared Memory," in *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI-2)*, Oct. 1996.

[28] M. Dahlin, B. Chandra, L. Gao, and A. Nayate, "End-to-end WAN Service Availability ," *ACM/IEEE Transactions on Networking*, vol. 11, no. 2, Apr. 2003.

[29] F. Douglis and J. K. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software - Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[30] C. Dubnicki, L. Iftode, E. Felten, and K. Li, "Software Support for Virtual Memory-Mapped Communication," in *Proc. 10th International Parallel Processing Symposium (IPPS-10)*, Apr. 1996.

[31] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, 1998.

[32] E. N. Elnozahy and D. B. Johnson, "The Performance of Consistent Checkpointing," in *Proc. 11th Symposium on Reliable Distributed Systems (SRDS-11)*, Oct. 1992.

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

[34] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy, "Integrating Coherency and Recoverability in Distributed Systems," in *Proc. 1st Symposium on Operating Systems Design and Implementation (OSDI-1)*, Nov. 1994.

[35] C. Fetzer, "Perfect Failure Detection in Timed Asynchronous Systems," *IEEE Trans. Computers*, vol. 52, no. 2, pp. 99–112, 2003.

[36] C. Fetzer and F. Cristian, "Fail-Awareness in Timed Asynchronous Systems," in *Proc. 15th Symp. on Principles of Distributed Computing (PODC)*, (Philadelphia), 1996.

[37] FreeBSD, "Freebsd problem reports." http://www.freebsd.org/cgi/query-pr-summary.cgi.

[38] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proc. 17th International Symposium on Computer Architecture (ISCA-17)*, May 1990.

[39] I. Gupta, T. Chandra, and G. Goldszmidt, "On Scalable and Efficient Distributed Failure Detectors," in *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, Apr. 2001.

[40] "IBM Autonomic Computing." http://www-1.ibm.com/servers/autonomic.

[41] "Icecast Streaming Server." http://www.icecast.org.

[42] L. Iftode, *Home-Based Shared Virtual Memory.* PhD thesis, Princeton University, June 1998.

[43] L. Iftode and J. P. Singh, "Shared Virtual Memory: Progress and Challenges," *Proceedings of the IEEE*, vol. 83, no. 3, Mar. 1999.

[44] G. Janakiraman and Y. Tamir, "Coordinated Checkpointing Rollback Error Recovery for Distributed Shared Memory Multicomputers," in *Proc. 13th Symposium on Reliable Distributed Systems (SRDS-13)*, Oct. 1994.

[45] B. Janssens and W. K. Fuchs, "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory," in *Proc. 13th Symposium on Reliable Distributed Systems (SRDS-13)*, Oct. 1994.

[46] B. Janssens and W. K. Fuchs, "Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems," *Journal of Parallel and Distributed Computing*, Oct. 1995.

[47] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," in *Proc. 17th International Fault-Tolerant Computing Symposium (FTCS-17)*, June 1987.

[48] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proc. 19th International Symposium on Computer Architecture (ISCA-19)*, May 1992.

[49] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proc. Winter '94 USENIX Conference*, Jan. 1994.

[50] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherency and Recoverability," in *Proc. 25th International Symposium on Fault-Tolerant Computing Systems (FTCS-25)*, June 1995.

[51] R. R. Koch, S. Shortikar, L. E. Moser, and P. M. Melliar-Smith, "Transparent TCP Connection Failover," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, 2003.

[52] A. Kongmunvattana and N.-F. Tzeng, "Coherence-Centric Logging and Recovery for Home-Based Software Distributed Shared Memory," in *Proc. of the 1999 International Conference on Parallel Processing (ICPP '99)*, Sept. 1999.

[53] A. Kongmunvattana and N.-F. Tzeng, "Lazy Logging and Prefetch-Based Crash Recovery in Software Distributed Shared Memory Systems," in *Proc. 13th Int'l Parallel Processing Symposium (IPPS-13)*, Apr. 1999.

[54] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental Study of Internet Stability and Backbone Failures," in *Proc. 29th Symp. on Fault-Tolerant Computing (FTCS)*, June 1999.

[55] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[56] D. E. Lowell and P. M. Chen, "Free transactions with Rio Vista," in *Proc. 16th Symposium on Operating Systems Principles*, Oct. 1997.

[57] M.-Y. Luo and C.-S. Yang, "Constructing Zero-Loss Web-Services," in *Proc. 20th IEEE Intl. Conf. on Computer Communications (INFOCOM)*, June 2001.

[58] "Mellanox, Inc.." http://www.mellanox.com.

[59] D. Milojicic, F. Douglis, Y. Panedeine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, Sept. 2000.

[60] S. Mishra, M. Marwah, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, 2003.

[61] J. Mogul, "Personal communication," June 2003.

[62] D. Mosberger and T. Jin, "httperf: A tool for measuring web server performance," in *Proc. 1st Workshop on Internet Server Performance*, June 1998.

[63] "Myricom: Creators of Myrinet." http://www.myri.com.

[64] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System," in *Proc. 13th Symposium on Principles of Distributed Computing (PODC-13)*, Aug. 1994.

[65] M. J. K. Nielsen, "Titan System Manual," Technical Report WRL-86-1, HP Labs, Sept. 1986.

[66] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," in *Proc. USENIX Annual Technical Conference*, USENIX Association, June 1999.

[67] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[68] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software - Practice and Experience*, vol. 29, no. 2, pp. 125–142, 1999.

[69] J. Postel, "RFC 793: Transmission Control Protocol," Sept. 1981.

[70] "PostgreSQL." http://www.postgresql.org.

[71] X. Qie, R. Pang, and L. Peterson, "Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software," in *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[72] Y. Rekhter and T. Li, "RFC 1771: A Border Gateway Protocol 4 (BGP-4)," Mar. 1995.

[73] G. G. Richard III and M. Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory," in *Proc. 12th Symposium on Reliable Distributed Systems (SRDS-12)*, Oct. 1993.

[74] J. Satran *et al.*, "iSCSI, IETF Draft," Sept. 2002.

[75] M. Satyanarayanan, H. Mashburn, P. Kumar, D. C. Steere, and J. Kistler, "Lightweight Recoverable Virtual Memory," *Transactions on Computer Systems*, vol. 12, no. 4, pp. 33–57, Feb. 1994.

[76] D. Scales, K. Gharachorloo, and C. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, Oct. 1996.

[77] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.

[78] M. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," in *Proc. 2nd Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 1996.

[79] M. Seltzer and C. Small, "Self-Monitoring and Self-Adapting Operating Systems," in *Proc. 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[80] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, "Fine-Grained Failover Using Connection Migration," in *Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2001.

[81] A. C. Snoeren and H. Balakrishnan, "An End-to-End Approach to Host Mobility," in *Proc. 6th ACM MOBICOM*, Aug. 2000.

[82] C. Soules, J. Appavoo, K. Hui, D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System Support for Online Reconfiguration," in *Proc. USENIX Annual Technical Conference*, June 2003.

[83] K. Srinivasan, "M-TCP: Transport Layer Support for Highly Available Network Services," Technical Report DCS-TR-459, Rutgers University, Oct. 2001.

[84] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," in *Proc. 16th Symposium on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[85] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzberger, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "RFC 2960: Stream Control Transport Protocol," 2000.

[86] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, Aug. 1985.

[87] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems," in *Proc. 21st Int'l. Symp. on Fault-Tolerant Computing (FTCS-21)*, 1991.

[88] F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode, "Nonintrusive Remote Healing Using Backdoors," in *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[89] F. Sultan, T. Nguyen, and L. Iftode, "Limited-size Logging for Fault-Tolerant Distributed Shared Memory with Independent Checkpointing," Technical Report DCS-TR-409, Department of Computer Science, Rutgers University, Feb. 2000.

[90] F. Sultan, K. Srinivasan, and L. Iftode., "Transport Layer Support for Highly-Available Network Services," in *Proc. HotOS-VIII*, May 2001. Extended version: Technical Report DCS-TR-429, Rutgers University.

[91] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Highly Available Internet Services Using Connection Migration," Technical Report DCS-TR-462, Rutgers University, Dec. 2001.

[92] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode., "Migratory TCP: Connection Migration for Service Continuity in the Internet," in *Proc. ICDCS 2002*, July 2002.

[93] G. Suri, B. Janssens, and W. K. Fuchs, "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory," in *Proc. 25th International Fault-Tolerant Computing Symposium (FTCS-25)*, June 1995.

[94] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," in *Proc. 19th Symp. on Operating Systems Principles (SOSP)*, Oct. 2003.

[95] "Test TCP (TTCP)." ftp://ftp.arl.mil/pub/ttcp.

[96] "The Virtual Interface Developer Forum." http://http://www.vidf.org.

[97] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," in *Proc. 9th Symp. on Operating Systems Principles (SOSP)*, 1983.

[98] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proceedings of ASPLOS-X*, Oct. 2002.

[99] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. 22nd International Symposium on Computer Architecture (ISCA-22)*, June 1995.

[100] C.-S. Yang and M.-Y. Luo, "Realizing Fault Resilience in Web-Server Cluster," in *Proc. SuperComputing 2000*, Nov. 2000.

[101] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, 2003.

[102] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic, "Efficient TCP Connection Failover in Server Clusters," in *Proc. 23rd IEEE Intl. Conf. on Computer Communications (INFOCOM)*, Mar. 2004.

[103] Y. Zhou, P. M. Chen, and K. Li, "Fast Cluster Failover using Virtual Memory-mapped Communication," in *Proc. 13th International Conference on Supercomputing*, ACM Press, 1999, pp. 373–382.

[104] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," in *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI-2)*, Oct. 1996.

Vita

Florin Sultan

Education

- Ph.D. Computer Science, Rutgers University, New Jersey (2004)
- M.S. Computer Science, Old Dominion University, Virginia (1996)
- B.S. Computer and Control Engineering, Polytechnic Institute of Bucharest, Romania (1989)

Occupations and Positions Held

- Research Staff Member, Institute for Research in Informatics, Bucharest, Romania (1989 - 12/1992)
- Consultant, Computer Sharing Romania s.r.l., Bucharest, Romania (1990 - 1993)

Publications

- A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, L. Iftode. "Remote Repair of Operating System State Using Backdoors." *The International Conference on Autonomic Computing (ICAC 2004)*, May 2004.
- F. Sultan, A. Bohra, L. Iftode. "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions." *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, October 2003.
- F. Sultan, A. Bohra, I. Neamtiu, L. Iftode. "Nonintrusive Remote Healing Using Backdoors." *The 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.
- F. Sultan, T. Nguyen, L. Iftode. "Lazy Garbage Collection of Recovery State for Fault-Tolerant Distributed Shared Memory." *IEEE Transactions on Parallel and Distributed Systems*, vol.3, no. 10, October 2002.
- F. Sultan, K. Srinivasan, D. Iyer, L. Iftode. "Migratory TCP: Connection Migration for Service Continuity over the Internet." Short paper, *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, July 2002.

- F. Sultan, K. Srinivasan, L. Iftode. "Transport Layer Support for Highly-Available Network Services." Position Summary, *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.
- F. Sultan, T. Nguyen, L. Iftode. "Scalable Fault-Tolerant Distributed Shared Memory." *Proceedings of Supercomputing 2000*, November 2000.
- H. Abdel-Wahab, I. Stoica, F. Sultan, K. Wilson. "A Simple Algorithm for Computing Minimum Spanning Trees in the Internet." *Journal of Information Science*, vol. 101, no. 1/2, 1997.
- I. Stoica, F. Sultan, D. Keyes. "A Hyperbolic Model for Communication in Layered Parallel Processing Environments." *Journal of Parallel and Distributed Computing*, vol. 39, no. 1, pp. 29-45, November 1996.
- H. Abdel-Wahab, I. Stoica, F. Sultan. "The Design and Implementation of an Internet Conference Information System." *Journal of Internetworking Research and Experience*, vol 6., no. 2, September 1996.
- I. Stoica, F. Sultan, D. Keyes. "Evaluating the Hyperbolic Model on a Variety of Architectures." *Proceedings of EUROPAR '96*, August 1996.
- H. Abdel-Wahab, I. Stoica, F. Sultan. "Universal Internet Conference Information System." *Journal of Information Science*, vol. 91, no. 1/2, May 1996.
- H. Abdel-Wahab, I. Stoica, F. Sultan. "A Simple and Fast Distributed Algorithm to Compute a Minimum Spanning Tree in the Internet." *Proceedings of the Joint Conference on Information Sciences '95*, September 1995.
- I. Stoica, F. Sultan, D. Keyes. "Modeling Communication in Cluster Computing." *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.