Nonintrusive Failure Detection and Recovery for Internet Services Using Backdoors *

Florin Sultan†, Aniruddha Bohra†, Yufei Pan†, Stephen Smaldone†, Iulian Neamtiu‡, Pascal Gallard∓, and Liviu Iftode†

† *Department of Computer Science*
Rutgers University,
Piscataway, NJ 08854-8019
{sultan, bohra, yufeipan, smaldone, iftode}@cs.rutgers.edu

‡ *Department of Computer Science*
University of Maryland,
College Park, MD 20742
neamtiu@cs.umd.edu

∓ *IRISA / INRIA Rennes*
Campus Universitaire de Beaulieu,
35042 RENNES Cedex - France
Pascal.Gallard@irisa.fr

Abstract

We describe an architecture for nonintrusive failure detection and recovery in a cluster of Internet servers in which nodes mutually monitor their liveness and recover client sessions from failed nodes. The system is based on Backdoors, a novel architectural approach for remote healing of computer systems. Backdoors enables monitoring and recovery/repair of state in a computer system by remote access to system resources (memory, I/O devices) without using its processors. Backdoors allows remote actions to be performed with no overhead, and even when the processors (but not the memory) of a machine are not available.

We have implemented a Backdoors prototype by modifying the FreeBSD kernel and using Myrinet NICs for remote access. The system uses remote DMA operations to perform monitoring, detect failures and extract OS and application state from a failed machine. We have used our system to run several open-source Internet servers and to run a complex multi-tier e-commerce application. The system tolerates multiple node failures while providing correct and continuous service to ongoing sessions, with negligible disruption.

1 Introduction

As computer systems become more complex, tolerance to failures and recoverability without compromising performance have emerged as guiding principles for system design [21]. When the computer is a server, the cost of a failure is compounded by the number of users and importance of the service to its clients. The growth of the Internet has led to the development of large-scale Internet services that users expect to be available at all times and even to be more reliable than their own machines. The scale and complexity of the architectures supporting Internet services make them fail in many ways, failures being triggered by a large number of diverse factors [18]: software faults, operator errors, component failures, etc.

The simple node redundancy typically used in Internet services is sufficient to cope with failures when service sessions are short, idempotent, or when timeliness of transactions is not an important factor to the user. In such cases, a system may drop sessions and clients can "recover" them by later reissuing their requests. However, this solution is not viable for critical services more and more prevalent in the Internet (e-commerce, banking systems, auction systems, etc.). First, in such services the failure may incur a cost (in terms of money) to every single user present in the system and to the service provider. While a restart may bring the system into a clean state, it also means the loss of currently executing transactions. Moreover, re-issuing them may violate exactly-once semantics implied by this class of services. Second, if a system drops clients, there is no guarantee (depending on load, load balancing and admission policies, etc.) that they would be re-admitted soon again to be able to resume their sessions. Third, service guarantees may include uninterrupted delivery, at least as much as the internetwork permits it. We contend that current Internet service architectures lack support for salvaging ongoing stateful client sessions in case of failure, while these sessions may be critical to their users or simply cannot be restarted.

Traditional recovery methods based on process checkpointing and restart after a crash cannot be applied to an Internet service for several reasons: *(i)* checkpointing is a heavyweight operation and using it on a live service severely degrades its performance; *(ii)* most services are interactive, so recovery by reboot and restart from a checkpoint (if at all possible) may not be tolerated by clients; *(iii)* use of a stateful protocol (TCP/IP) for data transport makes restart from a previous state problematic; *(iv)* services are often structured as a collection of multiple processes running on behalf of a client, potentially on multiple nodes, which forces checkpoint synchronization or use of complex rollback techniques after restart. Previous solutions to these problems, e.g., [30, 25, 19, 1, 29] are either not directly applicable to Internet services or have serious limitations

*This work is supported in part by the National Science Foundation under NSF CCR-0133366 and ANI-0121416

in scope. With one exception [30], their common drawback is the intrusiveness during the failure-free execution of the system for which they provide recovery support, e.g., they checkpoint state to stable storage, back up full communication/computation state across server nodes, place costly taps on the critical communication path, or use massive broadcasts of recovery state.

This paper proposes a solution to the problem of failure detection and recovery in an Internet service based on *Backdoors* (BD), a novel architectural approach for remote healing of computer systems [27]. BD combines hardware, firmware and OS extensions to support highly accurate remote monitoring and effective healing actions on a computer system without using its processors or relying on its OS resources.

We describe a BD-based architecture that provides uninterrupted Internet service to clients in face of multiple server node failures, without requiring changes to client OS/applications. In this architecture, server nodes cooperatively: *(i)* observe each other's healthiness and detect failures, and *(ii)* transparently take over client sessions from failed nodes and reconfigure the service after failure. We assume that a failure can impair a server node to the extent where *the system memory is still accessible*. For example, the OS may hang, deadlock or crash, a critical hardware component may not respond, the network may suffer a partition, etc. Power failures or failures that may cause extensive corruption to system memory are excluded.

To support BD functionality, we use two novel OS mechanisms. The first is the *Progress Box* (PB), a per-node structured collection of progress meters that express some measure of system healthiness as scalar values monotonically increasing over time. A PB characterizes "liveness" of a node and can be accessed remotely. The second mechanism is a light-weight *State Box* (SB) that maintains the state associated with a client service session, for recovery on another server node after a failure. SBs support transparent failover of service for clients of a failed node without requiring server processes to synchronize between themselves or with the client to ensure consistent behavior after recovery.

The unique feature of our system is that both monitoring and recovery incur zero or negligible CPU overhead on server nodes and can work even after failure of a target machine. The system achieves nonintrusive failure detection and recovery in three ways: *(i)* use of a BD implemented with remote memory communication for access to a system; *(ii)* use of light-weight SBs to encapsulate recovery state; *(iii)* propagation of recovery state only in response to a failure.

The remainder of the paper is structured as follows. Section 2 presents the system model. Section 3 describes the BD idea. Section 4 presents the design. Section 5 describes our BD prototype. Section 6 presents a case study with a real Internet service. Section 7 presents an experimental evaluation of the implementation. Section 8 reviews related work. Section 9 concludes the paper.

2 System Model

In this section, we describe the system model and introduce the mechanisms on which our failure detection and recovery architecture is based: **Monitoring** using *progress boxes* (PB); **Recovery** support using *state boxes* (SB). We assume a service running on a cluster of machines in which partitions of multiple nodes with similar functionality exist. Monitoring of a node is done from a different node of the same cluster. Recovery exploits redundancy of nodes in terms of logical functionality.

For simplicity of presentation, we assume single-node failures, although the mechanisms we describe can be extended to support an arbitrary number of failures. The failure model is fail-stop: a failed node does not behave erratically (the failure is non-Byzantine). For failure detection purposes, each node has its own private clock. Clocks may not be synchronized, but their drift rates (from an arbitrary clock) must remain constant. The system *enforces* the failure model and protects against false positives by halting any OS activity on a node suspected to have failed.

A failure may reflect various degrees of "illness" and 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 locking error, a misplaced panic only reached under certain stress conditions, etc.; *(ii)* a wrong operator command or a misconfiguration that causes the node/OS to halt or crash, generate errors, or slow down under prescribed levels while servicing clients under reasonable load; *(iii)* a faulty hardware component that ceases to respond, e.g., processor, network, etc.

The *recovery action* taken when a failure is detected is to dynamically move affected sessions out of the impaired server node(s). We assume that the (soft or hard) state of a session in the memory of a node subject to failure is sufficient and can be used to continue service on another node.

2.1 Monitoring

The monitoring component of the system is responsible for failure detection. It is implemented by *monitor processes* that run on the same cluster as the service. Monitor nodes (denoted by M) run a failure detection algorithm based on observation of state dynamics in monitored nodes (denoted by m). Figure 1 (a) illustrates the model in the single-node failure case, with an example topology of a virtual ring where each node is monitored by its predecessor. In case a node fails (Figure 1 (b)), its monitor detects the failure and takes the recovery action of dynamically relocating clients to remaining nodes.

We impose two major constraints on the design of the monitoring function: First, it must be easily replicated, in

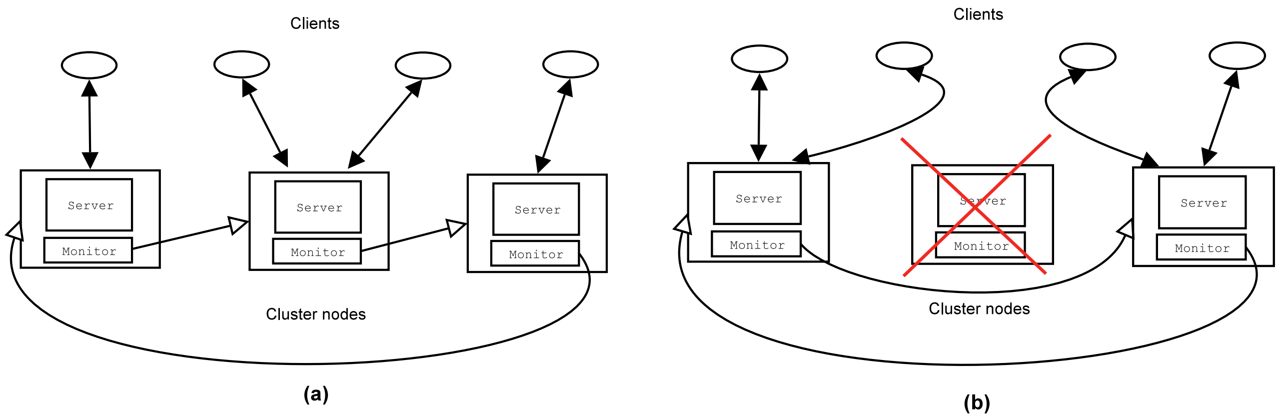

Figure 1: *System during normal operation, with nodes arranged in a virtual ring by monitor-monitored logical links (a). Failure of middle node is detected by its monitor, client sessions relocated and the ring reconfigured (b).*

the same way the service functionality is replicated across cluster nodes. This avoids single points of failure (e.g., a dedicated monitor) but at the same time it must not introduce monitoring overhead on service nodes.

Second, the (M, m) pairs must be only loosely coupled: *(i)* Observing an m node must not directly involve it. Since m might be dead, M must not rely on remote execution for fetching observed state from it, i.e., observation of m's state by M must essentially translate into a 0-cycles operation on m. *(ii)* m must not interact with M; in general, it may be unaware of its existence. *(iii)* Participation of m in monitoring must be voluntary, by local state introspection and reporting.

To achieve these goals, we introduce an OS mechanism called *Progress Box* (PB) that allows an external monitor to observe the liveness of a node by enabling a loose producer-consumer relationship between the two. Internally, the PB is a structured area in m's protected (system) memory, comprised of data records called *progress counters*. Entities (e.g., OS subsystems) running on a node are the PB producers. They allocate progress counters inside the PB and update them within pre-declared deadlines to signal their liveness to a monitor. Examples of OS-level progress counters are: interrupts for a certain device, context switches, scheduling decisions for a given processor, etc. At the other end, an M node accesses (reads) m's PB and uses it to assess m's health.

2.2 Recovery

The recovery component is responsible for dynamic relocation of service sessions off of a monitored node m, following a failure detection by a monitor M. We opt for session-level recovery because: *(i)* it enables light-weight system support, factoring out redundant heavy-weight state or which can be re-created at a new node; *(ii)* it enables graceful re-distribution of sessions from a failed node to multiple nodes, avoiding a concentrated surge of load on a single recovery node; *(iii)* it enables fine-grained policies in supporting recovery, depending on how critical sessions are, service guarantees to certain classes of users, etc. In our system, recovery operates at the granularity of one service session: it extracts, transfers, and reinstates its state on a healthy node, and assists the application running there in resuming consistent service to the client.

There are several design constraints that the support for recovery 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 involve entities running on the failed node for state extraction during recovery, since a failed node may no longer possess the necessary resources to respond; *(iv)* it must be transparent to the client OS/applications both during failure-free execution and recovery.

To achieve these goals, we use a light-weight per-session *State Box* (SB), an OS mechanism that enables a session to relocate at any point during its lifetime, transparent to client and asynchronously with respect to server execution. The SB model 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 SB represents an abstraction of *discrete* application-level state associated with a client session, spanning multiple process contexts, which can be extracted from a failed server and restored at a new server. At the OS level, an SB is an ordered sequence of fine-grained state components associated with processes involved in servicing the client. For each process, an SB stores client session state and OS state of communication channels. For example, on a front-end machine the first component in the SB sequence corresponds to the process that accepts the client connection for service. Subsequent components correspond to other processes (if any) that participate in servicing the client. For a session serviced in cooperation by multiple machines, the SB state components are distributed over nodes servicing it.

The server application must interact with an SB through a simple recovery API for saving and retrieving state pertaining to a given session. To resume the service after a failure, the system extracts the SB from the failed node to a new node where it is used to reinstate the service state and the communication state for all processes that service the session.

Figure 2 shows the two mechanisms as used in the monitoring and recovery protocol. During normal operation *(a)*, *M samples* the PB of *m* and performs liveness checks on progress counters, according to a failure-detection algorithm. If it detects a failure, e.g., caused by CPU loss, it initiates session recovery on a healthy recovery node (not necessarily *M*). The OS of the recovery node *extracts* a session's SB and reinstates the session state *(b)*.

3 The Backdoors Architecture

Backdoors (BD) [27] is a system architecture for nonintrusive remote healing of computer systems. The central idea in BD is to combine hardware and software mechanisms to enable highly accurate monitoring and effective healing actions without using processors of the target machine.

To support BD, a computer must be equipped with a specialized intelligent NIC that enables external access to its resources (memory, I/O devices, etc.). Remote access operations through a BD have two crucial properties:

Nonintrusiveness to target system operation. They do not consume processor cycles, nor do they rely on critical resources of the target. Processors are only involved in BD initialization and bootstrap operations.

Availability even after a system failure. They are not affected by a failure/crash of the OS that leads to loss of processors, by resource exhaustion, or by hardware failures.

In [27] we have identified remote memory communication (RMC) as a basic building block for BD operation. RMC is a technology originally developed to lower the overhead of communication by reducing OS involvement [11, 5]. RMC defines remote DMA (RDMA) primitives that allow external access to the memory of a host while bypassing its processor(s). With RDMA write, a sender can write into a remote memory buffer without remote processor intervention. With RDMA read, a receiver can initiate a transfer from a specified remote memory buffer without involving the remote OS or processor. Both RDMA primitives are included in industrial standards [9, 15] and are implemented in modern RMC controllers [16, 17].

Figure 3 illustrates the RDMA read/write operations between two machines. For initialization, both endpoints execute a one-time register operation notifying their NICs of the memory buffers involved in RDMA. The processor of the left node initiates the RDMA operation (vertical solid arrow). The solid horizontal arrows show the logical RDMA data path, while the dotted line follows the physical data path. The remote NIC bypasses the host processor to directly perform a DMA to/from the remote memory.

BD takes a completely novel approach on the use of RMC in the design of remote healing systems by exploiting its nonintrusive communication feature. However, while some RMC capabilities may be mapped directly into basic BD operations like access to memory of a target system, a BD is more than just simple memory access. To support remote healing through BD, an OS must provide *remote access hooks* as interfaces to the state of a system, e.g., for performing monitoring and recovery/repair actions on the target OS or the applications running on it. A remote access hook enables remote access to regions of OS/application memory that hold critical state or to I/O devices of the target system. In addition, the BD must provide *remote locking* operations to ensure mutual exclusion of concurrent accesses to a live target system.

The system described in this paper uses a BD along with two OS hooks: PBs for liveness monitoring and SBs for recovery. Remote memory read operations on PB/SB performed through the BD achieve nonintrusive monitoring and recovery through state extraction from a failed system. In addition, since BD is inherently nonintrusive, monitors can be easily replicated without extra overhead on a target system.

4 Design

4.1 The Progress Box (PB)

The PB is a structured collection of progress counters in the OS memory, allocated to monitored entities upon request. A progress counter is a tuple $< H, T, V >$, where H is a handle that uniquely identifies it, T is the failure detection deadline, and V is a scalar, monotonically increasing value (the actual counter).

A monitored entity must cooperate with a monitor by increasing V for its associated progress counter(s) to prove its liveness. Upon creation of a progress counter, the monitored entity specifies the detection deadline T. This establishes a *contract* between monitor and monitored: *(i)* the monitored commits itself to updating V at intervals smaller than T; *(ii)* if the monitor sees no change in V after T units of time, i.e., the progress counter is stalled, it decides that a failure that affects the monitored entity has occurred.

4.1.1 Monitoring Interfaces

Monitored side. The main primitives of the OS monitoring interface are:

```
pc_handle = new_os_pc(τ)
set_pc_value(pc_handle, value)
```

where τ is the failure detection deadline for the counter. `new_os_pc()` creates an OS-level counter and `set_pc_value()` is used to update the counter value.

Progress at the OS level translates into various metrics, depending on the activity of the monitored subsystem,

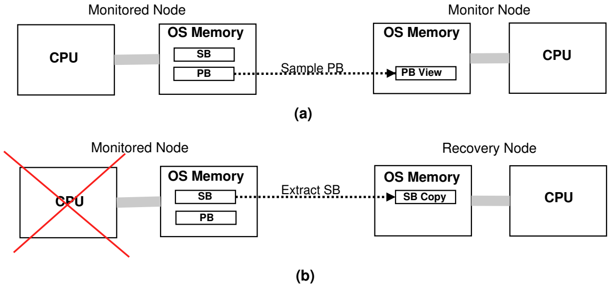

Figure 2: *(a) A monitor node samples a monitored node's PB to detect failures; (b) When failure is detected, a recovery node extracts the SB of a service session from the failed node and reinstates it locally.*

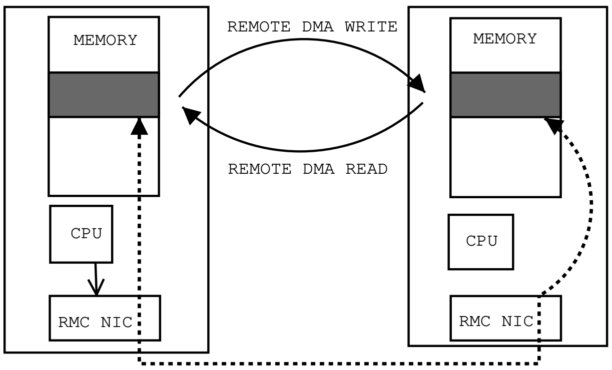

Figure 3: *Remote DMA read/write operations in RMC.*

e.g., a device taking interrupts, the OS scheduler making scheduling decisions, the page daemon performing page swapping, etc. A stalled OS counter is a first indication that a node is ill and can be used by a monitor to declare a general node failure.

Monitor side. To implement failure detection and initiate recovery, a monitor process uses an API to access remote PBs and to trigger session relocation:

```
pb_view = fetch_pb(nodeID)
session_tag_list = get_recoverable_sessions(nodeID)
extract_session(session_tag, nodeID)
```

`fetch_pb()` creates a local copy `pb_view` of the PB of a monitored node. `get_recoverable_sessions()` returns a list of tags (e.g, initial connection id's) that identify recoverable sessions serviced by the node. `extract_session()` initiates relocation for a session identified by `session_tag` from `nodeID` to the local node. For a null `session_tag`, it extracts all recoverable sessions serviced on the node.

The monitoring API decouples recovery from monitoring and supports fine-grained recovery actions. A monitor may use `get_recoverable_sessions()` as a helper call if it needs to perform fine-grained redistribution of recovered sessions to other nodes. Alternatively, all recoverable sessions of the target node can be extracted with a single call.

4.1.2 Failure Detector Accuracy

In a BD architecture, use of remote read operations for monitoring makes the hardware providing it (NIC, physical links) a single point of failure. Even with redundant RMC 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.

RMC path failures may lead to false positives in failure detection. Since a remote read operation is silent, it is impossible to distinguish path failure from a real crash of the remote node based only on reading values in the local view of the monitored PB: in both cases, progress counters would not change. A sound design cannot rely on the reliability of RMC hardware (typically higher than that of standard network hardware). To solve this problem, we take advantage of link-layer failure detection mechanisms provided by existent RMC hardware. A monitor would periodically probe the link-layer to check the healthiness of the RMC path. If the probe fails, the monitoring function is delegated to a backup monitor with a healthy link to the monitored node.

Message loss while reading the PB may have the same effect as a link failure: if M sees no change in the monitored PB because the RDMA request was lost, it may declare the monitored node dead. To cope with message loss on RMC links, the monitor must *adapt the sampling rate* R of the remote PB 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 [8, 13]. Using results in [13], it can be shown that the low latency and low probability of message loss typical of RMC ($RTT \sim 10\mu s$ and $p_{ml} \ll 10^{-8}$)[1] allow very high rates of sampling (practically limited only by the latency of fetching the remote PB), without false positives in failure detection.

4.2 The State Box (SB)

Essential to the SB 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. As a consequence, the point reached in servicing a given client session can be defined independently of other concurrent sessions and used to resume it on another node. Any other non-specific state needed to resume service of a migrated session (e.g., access to external databases, file repositories, etc.) is deemed accessible at the new node.

The session state may span multiple communicating processes in a *process service set* executing work for the client. This computation model is described by Figure 4, where the process service set $\{P_1, P_2\}$ handles the service session

[1]For lack of RMC 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.

of one client. Only a select process in the set, called the *root* or *accepting* process (P_1 in Figure 4), communicates directly with the client. The other processes can be either workers (processes spawned on behalf of and dedicated to servicing one client) or servers (processes that service multiple clients concurrently). Every process has a well-defined and reproducible initial state with respect to a client. Processes in a process service set may communicate via byte-stream channels (IPC or TCP/IP) and may span multiple machines involved in servicing the session.

4.2.1 Per-Client State Box

For the server application, a *state box* (SB) is a set of well-defined states reached in servicing a client session by each individual process in the service set. Each state component can be individually and independently used by its respective process to resume service to client. In the OS, an SB is reflected as an ordered set[2] of individual per-process state snapshots, the content 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 SB. Note that the state snapshots are *not* full checkpoints of a process and do not include the processor execution context.

To resume consistent service, an SB needs OS support for reinstating the state of communication channels at the new node. The OS includes in an SB the state of stateful communication channels (inter-process and client-server). Figure 4 describes the OS view of an SB consisting of continuation points saved by processes in the service set (the S_1 and S_2 snapshots), along with communication state in IPC channels and the TCP client connection.

An SB does not impose any form of synchronization on the server processes, nor between the root process and the client. Declaring and saving a continuation point, as well as resuming from it on another node, are operations performed independently by each process in the service set. Note that SB state components cannot be forcefully synchronized at the time of extraction. The OS cannot make upcalls in the application to collect "synchronous" application state for two reasons: it might not be consistent, and there might be no cycles to execute the upcall on a dead machine. This requires the OS to synchronize the two endpoints of a communication channel *after* reinstating the SB on the new node. To support this process, an SB maintains logs of communication activity in the OS. The OS synchronizes the application state snapshots using a *limited* form of log-based rollback recovery [10] to restore the session state in a process of the new server. The synchronization mechanism is similar to that used in [26], but different in that it is completely transparent to the client TCP/IP or OS.

[2] 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.

4.2.2 The SB Recovery API

SB provides a minimal API that allows a server application to resume a client session on another node 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 SB object. The state snapshot must completely describe the point the process has reached in the ongoing service session, to be used as a restart point on another node. The exported state is opaque to the OS and to the SB extraction protocol.

The SB API establishes a *contract* between an application process and the system. A process must execute the following actions: *(i)* upon starting service to a client, associate with the SB all the communication channels it needs to be restored after failure; *(ii)* export state snapshots during service; *(iii)* import the last state snapshot at the new server (during recovery) and resume service to the client. In exchange, a remote OS: *(i)* extracts and reinstates the state of the SB at the new node, and *(ii)* synchronizes the state of processes in the service set with the state of their associated communication channels.

The main primitives of the SB API are:

```
sb = sb_create(conn)
sb_export_state(sb, state_buffer)
sb_import_state(sb, state_buffer)
sb_associate(sb, ch_set)
sb = sb_open(ch)
```

where `sb` is the SB object associated with a client (an OS-specific identifier), `conn` is the client connection in the root process, `ch` identifies a communication channel, and `state_buffer` is an application memory buffer summarizing a snapshot of the client session state in a process.

The accepting process creates an SB using `sb_create` on the accepted client connection. Processes in the service set use `sb_export_state` to save state snapshots to an SB and `sb_import_state` to retrieve them (only once) after a failure. The `sb_associate` primitive is used by a process in the service set to enable continuation support by the OS for a selected set `ch_set` of its communication channels. The `sb_open` primitive returns the SB object to which channel `ch` was previously associated.

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

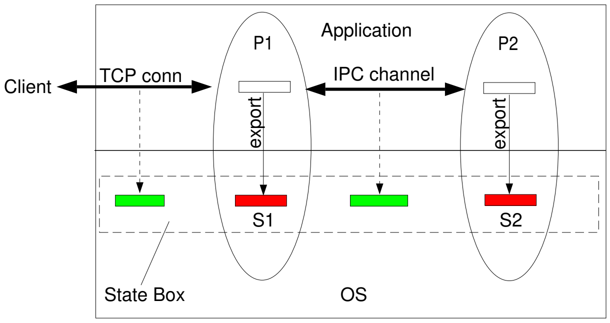

Figure 4: *The OS view of an SB: application state and the state of communication channels.*

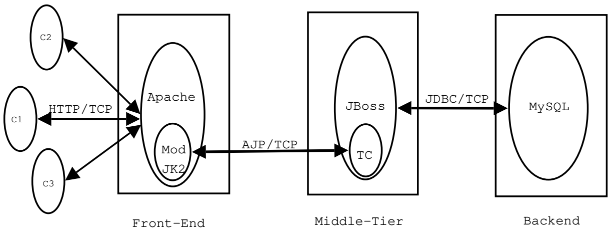

Figure 5: *RUBiS application architecture.*

5 Prototype

We have implemented a BD prototype in the FreeBSD 4.8 kernel, using Myrinet NICs [17] with Myrinet GM 2.0 for remote access. For remote monitoring and state extraction, we used in-kernel RDMA read operations between the machines involved in monitoring and recovery. Remote access is enabled by registering the whole kernel memory with the NIC and dynamically updating virtual-to-physical mappings when needed.

To ensure consistent remote access to in-kernel data structures, we implemented a *remote OS locking* mechanism that blocks execution of system calls and interrupt handlers on the target machine. We used remote OS locking to enforce the fail-stop model and completely eliminate unwanted effects of false positives in failure detection. When a monitor decides that a target node has failed, and before taking any recovery action, it acquires the remote OS lock to freeze any activity on the target machine in case its assessment is wrong. If the target machine has crashed in a critical section while holding the OS lock, the monitor waits for a timeout and then acquires it by brute force.

We have implemented the PB and SB mechanisms in the OS kernel. The SB implementation includes support for TCP connections and OS pipes and is completely transparent to client OS/applications. The failover of an established client TCP connection can take place in any state and is made transparent by using IP address takeover from the failed machine. The system overlaps extraction of state of a TCP endpoint 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.

The PB interface is implemented as a pseudo-device accessed both from the kernel (at the monitored node, for progress reporting) and from user space (at the monitor, for sampling the remote PB). Monitor processes sample remote PBs, compare the current and previous view of a PB, and identify counters that have been stalled beyond their detection deadlines. If no progress is detected in a critical counter, a monitor declares the node dead and issues calls to an *action plug-in*. Plug-in behavior may depend on the role of the failed node in the cluster, application specifics, etc. For example, an action plug-in for a front-end node may extract all connections bound to a specific port from the failed node. An action plug-in for a mid-tier node running an application server may warn front-ends to redirect their requests away from the failed node.

6 Case Study

As a case study for our system we chose RUBiS [7], 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, *timeliness* of a request can be as important as its successful and correct completion. For example, the bidding system allows items to be listed for sale only for limited periods of time. The typical behavior of bidders (wait until the end of the listing period in order to place "last minute" bids) makes bidding 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 state of a request is recoverable as it travels through various nodes in the service architecture. A request can be salvaged from a failed front-end or from a failed node in any intermediate tier down to the database back-end.

System Configuration. Figure 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 [2] 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 FE and MT nodes are equipped with backdoors. We make client sessions recoverable by modifying Apache and RUBiS beans to use the SB API. The back-end is assumed fault-tolerant through well-established methods, e.g., DB replication.

When a request enters the system, the FE tags it with a globally unique request ID, used to identify SB-encapsulated state belonging to the same session. On an FE node, the SB of a session contains the request and its ID. On an MT node, where the request is translated into a DB transaction, the SB contains the request ID, the transaction identifier, and the result of the transaction.

If an FE node fails, its monitor notifies other FE node(s) to extract the session SBs 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 first-time requests, an FE/MT node exports the state needed to recover the request to a session SB. For requests replayed during recovery, an MT node obtains the status (abort/commit) of the transaction from the DB and retrieves the transaction result from the SB. 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.

7 Experimental Evaluation

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 LanaiX NICs with a 133MHz PCI-X interface [17].

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 caused 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.

7.1 Microbenchmarks

PB/SB API Cost and SB Extraction Overhead. In a first experiment, we evaluate the run-time overhead of the PB and SB API by measuring the latency of the calls described in Section 4. Of the two components, the PB API is extremely light-weight (6 μs per system call), as it only writes integer values to a PB. On the other hand, implementing an efficient SB API is crucial to the failure-free performance of a system. Table 1 (columns 2-4) shows the cost of SB calls for three values of the SB state size. We can see that the SB API is also light-weight and imposes little run-time overhead for updating and retrieving SB state. We conclude that participation in monitoring and providing recovery support should be light-weight on a server node.

State size [*KB*]	*export* [μs]	*import* [μs]	*associate* [μs]	*SB extraction* [μs]
1	11	8	41	158
5	20	10	41	258
10	28	24	41	358

Table 1: *Cost of the three main SB system calls and of SB state extraction.*

In a second experiment, we estimate the costs of extracting an SB from a failed system as a function of the SB state size. We use a recoverable (i.e., augmented with the SB 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 SB 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 the SB extraction protocol to move and reinstate the state of a TCP connection along with the associated application-level snapshot. This setup is typical of state transfer during recovery between front-end nodes in a multi-tier architecture. The last column in Table 1 shows the results, proving that SB state extraction is light-weight for the recovery node. For comparison, the raw latency of an RDMA transfer is about 16 μs at small payloads, and 118 μs with a 10 KB payload.

Monitoring Overhead. On a monitor node, the overhead includes *(i)* monitoring cost (reading the local view of the monitored PB, comparing counter values, etc.), and *(ii)* cost of transferring the remote PB from the monitored node. To determine it, we measured the CPU usage of a monitor process while varying the sampling rate of a remote PB with 100 progress counters. In the worst case (sampling the PB in an infinite loop), the CPU usage was 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

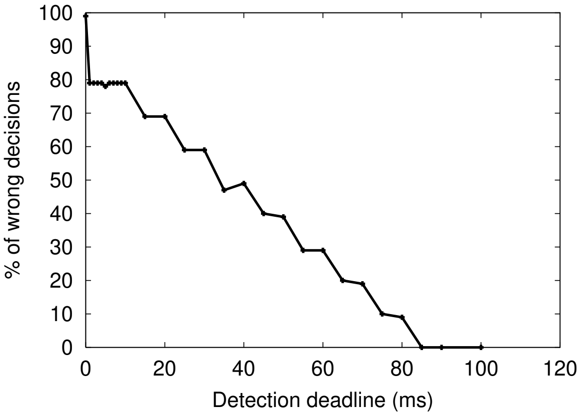

Figure 6: *Variation of false positives in failure detection with the detection deadline.*

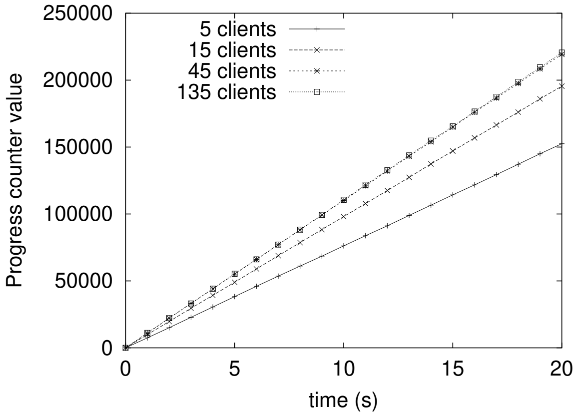

Figure 7: *Variation of an OS interrupt counter with time under different load conditions.*

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 doesn't block, if there are no other runnable tasks, no scheduling decision 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 6 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.

In a separate experiment, we collected traces of a progress counter for the number of interrupts serviced in a system running a synthetic streaming server. Figure 7 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 wrong decisions on an idle system with badly-chosen detection deadlines. These 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 clock) in deciding that a node has failed.

7.2 Real Applications

We have used our BD-based system to support recoverable sessions in several open-source servers [2, 20, 14] 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 6. 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.

Failure-free Overhead. We quantify the impact of using the SB API 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 SB API. The workload is a mix of auction requests that emulates client browsers using a methodology similar to [7], 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. Figure 8 shows the throughput for the base case, recoverable FE, and recoverable FE and MT. The performance of the system is identical in all three cases: the curves overlap at underload and exhibit statistically small variations at saturation due to nondeterministic behavior of the system.

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 600 requests coming 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 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

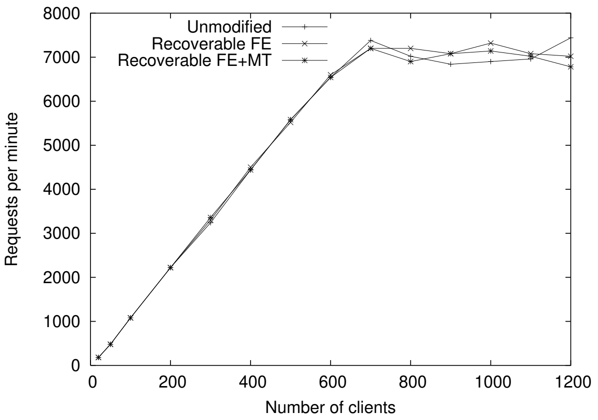

Figure 8: *RUBiS throughput.*

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 with 0.5 s think-time in runs of 90 s. With this workload, the node CPU utilization is about 45% on FE and 15-30% on MT.

We 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 9 shows the variation of aggregate throughput as bytes received by all clients over time along with the rate of client established connections (measured in bins of 1 s each), in a time window centered around the crash. The effect of crash and recovery is indistinguishable from normal workload variations, with the exception of a 2 s gap in the FE+MT case, due to cluster reconfiguration side-effects.

Figure 10 shows a timeline of events from detection to the end of recovery, in the worst-case, i.e., for the *last* recovered session. The end of recovery is marked by the first byte sent out from 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 less than 40 ms in the 2-node failure case. The best case values were 1.3 ms, 1.1 ms and 4.9 ms, with averages of 11.8 ms, 7.5 ms and 38 ms for the FE, MT, and FE+MT crashes, respectively. This shows that failover is fast and has practically no effect on the client perceived performance.

8 Related Work

A large body of theoretical work exists in the area of efficient failure detectors in distributed systems [8, 13, 12]. BD relies, in using RMC in failure detection, on theoretical results on the guaranteed accuracy of unreliable failure detectors under specified constraints [13].

Failure detection and fault isolation inside an OS have been studied in extensible operating systems [22, 24] with the goal of recovering from bugs in extension code. Self-monitoring is used in [23] for adapting OS behavior with the goal of increasing performance. Our system also relies on introspection, but uses external, nonintrusive observation of OS state and indirect inference of failure through monitored meters.

Traditional fault-tolerant systems like Tandem [4] 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 in (large-scale) clusters. BD does not provide component or OS fault tolerance, but offers a cost-effective and light-weight solution for recovery of critical state from a general-purpose OS after a failure.

Nooks [28] 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 memory accesses. Our current BD prototype relies on memory not being corrupted after a crash, a situation that Nooks can handle well. BD can detect failures regardless of their place of occurrence in system code, and can detect failures triggered by other factors than system software (operator errors, hardware faults).

RMC has been used in [30] for fast failover of nodes in a cluster of machines running scientific workloads by establishing efficient checkpoints in the memory of other nodes. TCP wrapping has been proposed in [1] to mask the failure and restart of a server with open connections. 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 in [25] in a cluster-based HTTP server. The scheme is limited to static HTTP transfers from single process servers and relies on massive broadcasts of recovery state inside the cluster. Primary-backup schemes have been used to build fault-tolerant TCP servers by mirroring their communication and computation state on another machine through active remote logging [29] or passive traffic tapping at the link-layer [19]. 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 [29]. None of these schemes has been shown to be applicable to Internet services.

In contrast to the above approaches, a BD-based system

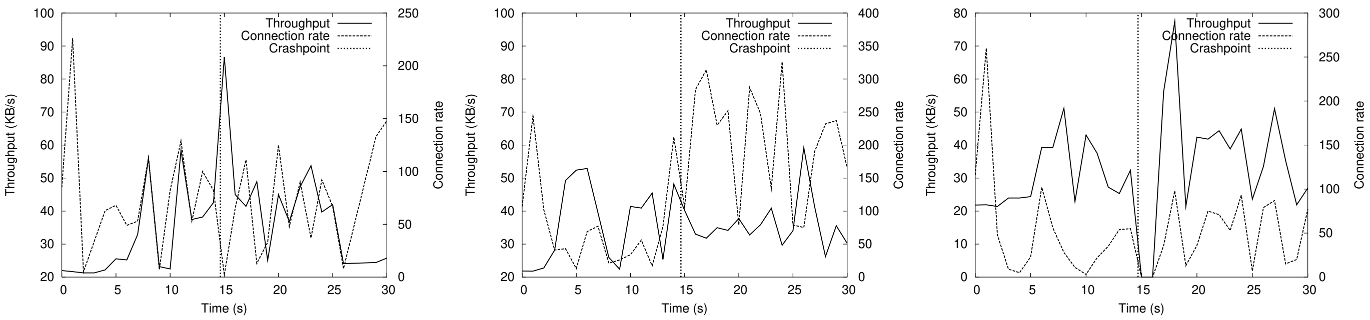

Figure 9: *Aggregate throughput and connection rate seen by clients across an FE crash, MT crash, and FE+MT crash (from left to right). Vertical lines mark the moment of the crash.*

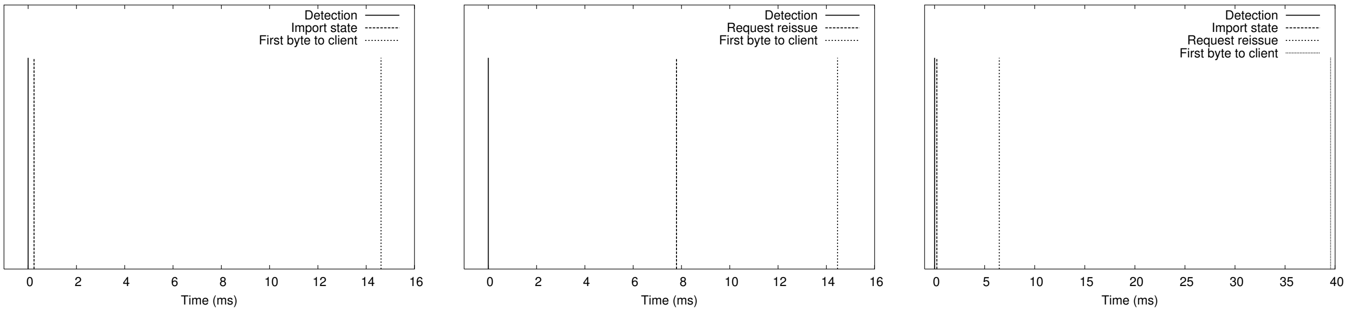

Figure 10: *Timeline of events across an FE, MT, and FE+MT crash (left to right). The crash occurs at -100 ms.*

provides both failure detection and recovery, it is lightweight and nonintrusive, it is application independent, and can be used with scalable cluster-based Internet services.

Fast recovery from crashes has been used to improve availability of a system/service. In Recovery Box [3], a stable OS memory area that can survive a crash was used to restore the live state of OS subsystems after reboot. The approach relies on transactional semantics to ensure recovery of applications with remote client sessions. Recovery is not transparent to clients (which must reconnect and resubmit their requests) and does not support communicating processes. Crash-only software [6] is a structured middleware design that relies on fast restarts to cope with component failures. Crash-only services are limited to stateless sessions, without exactly-once execution semantics, and rely on persistent state being maintained in a reliable state store that itself is crash-only. Crash-only software does not require OS changes, but it requires re-design of applications by imposing strict system-wide rules (e.g., inter-component interfaces for recovery). In contrast, the BD architecture provides generic OS support for accurate monitoring, failure detection and fast transparent failover of stateful client sessions from failed nodes.

9 Conclusions

We have described an architecture for Internet services that supports nonintrusive failure detection and recovery from node failures. The system is based on Backdoors, a novel architectural approach for remote healing of computer systems. Backdoors enables nodes in a cluster to perform mutual monitoring of their liveness, and to take over client sessions from failed nodes without involving the remote processors. We describe the design of OS abstractions that support remote nonintrusive monitoring (Progress Box) and recovery of service state associated to a session from a failed node (State Box).

We have implemented a Backdoors prototype in FreeBSD and present results of an experimental evaluation which shows that our system can efficiently detect node failures in a cluster and can recover interactive client sessions from failed nodes with minimal disruption to clients. We have used the system to run a complex multi-tier Internet service with transactional semantics, providing recovery from multiple node failures without compromising consistency of the data seen by clients or database integrity.

References

[1] 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.

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

[3] 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.

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

[5] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] G. Candea and A. Fox. Crash-Only Software. In *HotOS IX*, May 2003.

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

[8] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[9] D. Dunning et al. The Virtual Interface Architecture. *IEEE Micro*, 1998.

[10] 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 (CSUR)*, 34(3):375–408, 2002.

[11] E. Felten, R. Alpert, A. Bilas, M. Blumrich, D. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[12] C. Fetzer and F. Cristian. Fail-Awareness in Timed Asynchronous Systems. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 314–321a, Philadelphia, 1996.

[13] 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.

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

[15] The Infiniband Trade Association. http://www.infinibandta.org, August 2000.

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

[17] Myricom: Creators of myrinet. http://www.myri.com.

[18] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *Proc. 4th USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2003.

[19] M. Orgiyan and C. Fetzer. Tapping TCP Streams. In *IEEE International Symposium on Network Computing and Applications, NCA2001*, Boston, MA, USA, Feb 2002.

[20] 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.

[21] D. Patterson et al. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[22] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the 1996 Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[23] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997.

[24] E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Dec. 1995.

[25] 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.

[26] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proc. Symposium in Reliable Distributed Systems (SRDS)*, Oct. 2003.

[27] 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.

[28] 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.

[29] 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)*, 2003.

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