System Support for Nonintrusive Failure Detection and Recovery Using Backdoors *

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

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

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

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

Abstract

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. Heavy-weight approaches to preserve such state across failures, e.g., process/VM checkpointing, hot backups, etc., are intrusive to failure-free execution and/or require dedicated machines.

In this paper, we describe Backdoors (BD), a novel system architecture that enables applications to survive software failures that render the system unavailable (system hangs, OS crashes, etc.) using a light-weight state extraction mechanism. The architecture builds on two components: (i) a programmable NIC that enables access to the memory of a machine even when its processor is unavailable due to severe OS failures, and (ii) OS extensions that enable remote access to light-weight application and OS state.

We have implemented a Backdoors prototype by modifying the FreeBSD kernel and using Myrinet NICs for remote access. We describe a case study in using Backdoors in cluster-based Internet services in which nodes mutually monitor their liveness and recover client sessions from failed nodes.

1 Introduction

System recoverability and survivability in face of software failures without compromising performance have become more and more the focus of systems research [28]. Operating system failures are particularly harmful because they render a whole computer system unusable. An OS crash directly affects the application programs, which depend on core OS services (memory allocation, process management, I/O control, etc.).

This paper addresses the problem of system failures caused by an OS hang, crash, deadlock, panic, etc. Such failures require a reboot of the OS, a destructive action that causes all the live state of the machine (still in the system memory) to be lost. A reboot has unwanted side-effects like loss of data, disruption of service, inconsistencies in file systems, etc. Existing general-purpose operating systems lack mechanisms for salvaging and reusing this state, except for rudimentary crash dumps that can only be used for post-mortem analysis of the crash.

Our goal is to develop a system architecture that can reliably detect such failures and can enable critical state of the system (that would otherwise be lost) to survive them. In developing our system, we are looking for a solution that is light-weight, relies only on commodity components, and does not require re-design of the OS.

Past research in improving system reliability has focused on bug/fault isolation in extensible OSs [31, 33]. More recently, [39] has used isolation in a commodity OS kernel through lightweight kernel protection domains, to make it survive memory protection faults in extension code (device drivers). Previous work in the area of OS reliability includes full-fledged fault-tolerant operating systems [5] or making certain OS subsystems recoverable across crashes [4, 11].

Full replication of system state on a dedicated machine using virtual machine monitors (VMM) to survive failures has been developed in [7]. Solutions like process checkpointing or snapshotting the state of an OS running inside a VMM have been developed for migration of computing environments between two *healthy* systems [26, 30]. While these approaches can preserve and restore computation state of a system on another machine, they are heavy-weight and rely on the original OS/VM being available.

In this paper, we describe Backdoors (BD), a novel system architecture that enables applications to survive software failures that render the system unavailable (system hangs, OS crashes, etc.) using a light-weight state extraction mechanism. The architecture relies on two components: *(i)* a programmable NIC that enables access to the memory of a machine even when its processor is unavailable due to severe OS failures, and *(ii)* OS extensions that enable remote access to light-weight application and OS state. Using Backdoors, a remote system can perform accurate monitoring without CPU overhead on the target system and, upon detecting a failure, can extract light-weight state from the

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

failed system and reinstate it on a healthy system. Our current system relies on the availability of memory contents after an OS failure. Power failures or failures that may cause extensive corruption to system memory are not covered.

We present the design of two OS extensions used in Backdoors for monitoring the liveness of a system, detecting failures, and recovering critical functionality. The first extension is the *Sensor Box* (SB), a structured collection of meter variables (sensors) that track the state of OS subsystems in terms of health, liveness, performance, etc. The second mechanism is a light-weight *Continuation Box* (CB) that maintains fine-grained OS and application-specific state associated with a running application. The key idea behind the CB abstraction is that only essential state needs to be extracted and reinstated in a healthy OS/application running on another system, while the rest of the state is either redundant or can be easily re-created.

We have built a BD prototype using Myrinet NICs and extending the FreeBSD kernel with SB and CB implementations. We describe the design and implementation of a CB that can salvage client sessions from failed nodes in a cluster-based Internet server. Our system does not require changes or intervention of client OS or applications for recovery, while requiring minimal changes to server applications.

We demonstrate the viability of our approach with a case study in making client sessions of a complex multi-tier Internet service similar to eBay [9] survive multiple server node failures. We perform an extensive evaluation which shows that Backdoors does not have any impact on client-perceived performance during failure-free execution. In our experiments monitoring a system with BD has low overhead (under 1% CPU utilization), and recovery of a client session takes under 25 ms in the worst case (for the last session extracted from a system under load and for failures in both front and mid-tier nodes), with no impact on client-perceived performance and service correctness.

In summary, this paper has three main contributions:

(i) It proposes a novel system architecture based on remote memory communication to enable nonintrusive failure detection and recovery of system state even when the system is dead (hung).

(ii) It uses commodity hardware (a programmable NIC that can be inserted in any PCI-bus based system) and extends a general-purpose OS to achieve the proposed functionality.

(iii) It demonstrates the viability of using Backdoors with a realistic, multi-tier, transactional Internet service to achieve tolerance to multiple node failures.

The remainder of the paper is structured as follows. Section 2 gives background and reviews related work. Section 3 presents our approach. Section 4 describes the BD idea. Sections 5 and 6 present the SB and CB abstractions. Section 7 discusses limitations of our system and possible extensions. Section 8 describes our BD prototype. Section 9 presents a case study. Section 10 presents an experimental evaluation of the prototype. Section 11 concludes the paper.

2 Background and Related Work

2.1 System Support for 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 [31, 33] with the goal of recovering from bugs in extension code. Self-monitoring has been used to adapt OS behavior for increased performance [32] or to alleviate effects of DoS attacks on system/application [29]. Despite obvious advantages like instant access to entire system state, 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 generates 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, [14] has proposed the *timed-perfect failure detector*, a distributed protocol that eliminates false positives in detection of computer crash failures. 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 [10, 17, 15]. 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 [17]. 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.

2.2 System Support for 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 [5] 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 [4, 11]. The Recovery Box [4] is an OS-controlled NVRAM region used to store system state and retrieve it after crash/reboot. The Rio reliable file cache [11] preserves file system data in NVRAM, protects it during a crash and uses it for warm reboot. In contrast, [4, 11], BD provides a generic architecture for failure detection and light-weight state recovery. It does not require stable memory devices, but assumes that memory contents is still available and uses off-the-shelf hardware to access it.

VMMs have been used to intercept and back up the entire state of a system for tolerating failures, at the expense of dedicating full machines [7]. OS designs like [35] provide support for hot-swapping whole OS subsystems. 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 [39] 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 [25] 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 [23].

In [6], we showed how BD can be used to repair the damaged state of a live OS by remote intervention on OS data structures. In this paper, we show how Backdoors can be used to recover critical state from a failed system in cases where BD-based repair does not work, e.g., system hang or crash failures.

2.3 System Support for Reliable Internet Services

The case study we describe in this paper focuses on providing uninterrupted and correct service to clients of an Internet service. Specifically, we tackle the difficult problem of making *live* client sessions transparently survive failures of server nodes in complex Internet services with transactional execute-once semantics, where components of the service are distributed on multiple machines (multi-tiered architectures).

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., [42, 34, 1, 41, 20, 22, 42] are either not directly applicable to complex Internet services or have serious limitations in scope. Their common drawback is the intrusiveness during the failure-free execution of the system for which they provide recovery support.

TCP wrapping [1] masks 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 [34] in a cluster-based HTTP server. 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 by mirroring their communication and computation state on another machine through active remote logging [41] or passive traffic tapping at the link-layer [22, 20]. 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 [41]. None of these 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

cluster-based Internet services.

3 Approach

This paper proposes a recovery model and a recovery mechanism that enables survival of critical software state of a computer system across 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.

In this paper, *failure* denotes 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 may not be synchronized, but their drift rates (from an arbitrary clock) must remain constant.

Monitoring. The monitoring component of the system is responsible for failure detection. It assumes the existence of 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 remote observation of T's healthiness (Figure 1, *(a)*). This involves periodic inspection of state dynamics in T's memory, e.g., examining OS statistics in T's kernel `vmmeter` structure. 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.

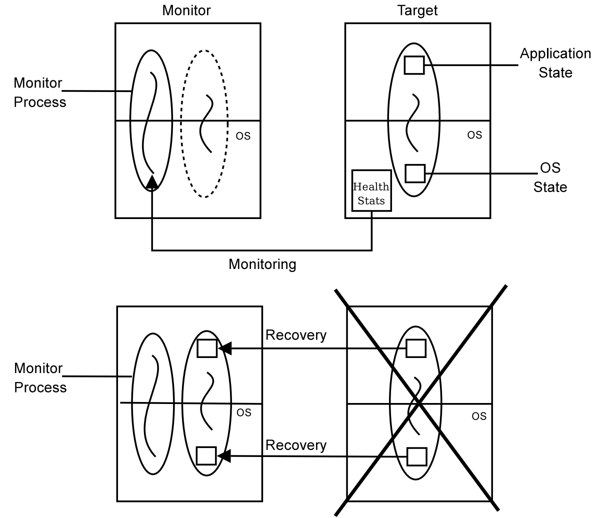

Figure 1: *An example of monitoring and recovery with Backdoors. (a) A monitor process on M inspects health indicators in T's OS memory. (b) On detecting a failure, M extracts light-weight critical application-level and OS state and reinstates it locally.*

Recovery. Upon detecting a failure, *M* recovers 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 1, *(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 fail-over 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 7 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, we need a uniform mechanism that enables *(i)* access to the state of the system without overhead during execution, *(ii)* access to T even when it is dead.

We describe a mechanism for recovery of critical state from a failure that combines:

- a system architecture that enables nonintrusive access to in-memory software state of a system;
- OS extensions for monitoring system liveness and for recovery of live light-weight state after a failure.

4 The Backdoors Architecture

Backdoors (BD) [38] is a novel system architecture for 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 even in the presence of failures that make a system unavailable. 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 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 2 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 local resources (memory, I/O devices), without using T's CPU.

To support remote healing, a Backdoor NIC must implement read, write and atomic 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 [12, 19] and are implemented by specialized controllers like [21].

Figure 2 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. To support complex remote healing operations under BD, an OS must

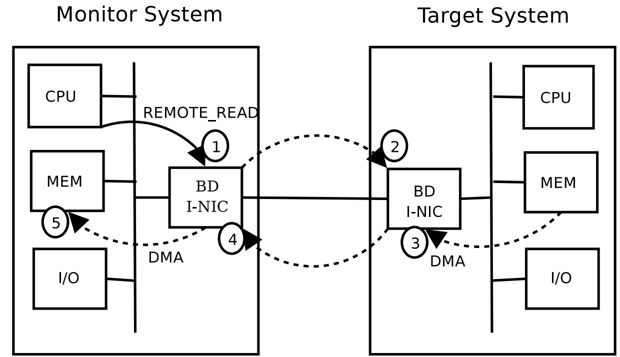

Figure 2: *A monitor-target pair in the Backdoor remote healing architecture. A monitor can access resources of the target system (memory, I/O devices) through the backdoor I-NICs, without using its CPU.*

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.

In a previous work [6], we have described OS extensions for remote repair of damaged OS state with Backdoors. This paper describes a system design using BD for failure detection and recovery by state extraction from a system after hang failures that make its processors/OS unavailable.

5 Failure Detection with Backdoors

We impose two major constraints on the design of the monitoring function. First, it must not introduce overhead on the monitor system M. This allows machines dedicated to other tasks to be used as monitors. For example, nodes in a computer cluster may both perform a distributed task and monitor each other, without requiring dedicated monitors. 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. *(iii)* Participation of T in monitoring must be voluntary, by local state introspection and reporting.

5.1 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. 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 limit 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:
(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 value 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 10 we show that an SB is light-weight both for local and remote access over the BD, and that OS-level progress counters can detect hang failures of the target OS both fast and reliably.

5.2 Failure Detector Accuracy

In a BD architecture, use of remote read operations 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 counters 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 counters in the local view of the monitored SB: in both cases, values would not change. A sound design cannot rely on the reliability of programmable NIC hardware (typically higher than that of standard network hardware). To solve this problem, we take advantage of link-layer failure detection mechanisms provided by existing hardware [24]. 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 M sees no change in progress counters in the monitored SB because a remote read request was lost, it may declare the monitored node dead. To cope with message loss on BD NIC 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 [10, 17]. Using results in [17], it can be shown that the low latency and low probability of message loss typical of existing hardware (e.g., $RTT \sim 10\mu s$ and $p_{ml} \ll 10^{-8}$ for Myrinet [24])[1] allow very high rates of sampling (practically limited only by the latency of fetching the remote PB), without false positives in failure detection.

6 Recovering with Backdoors

Using Backdoors for recovery relies on the insight that, in a failed machine, data structures holding "good" state may be 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)* rein-

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

state it and continue using it on another healthy machine. While *(i)* above can be achieved using remote memory access as described in Section 4, 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.

6.1 The Continuation Box (CB)

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/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 by 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 and reinstating it in a healthy OS/application running on another system, which can then continue the execution.

6.2 A CB for Internet Servers

We illustrate the Backdoors recovery model with the design of a Continuation Box 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, 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, 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

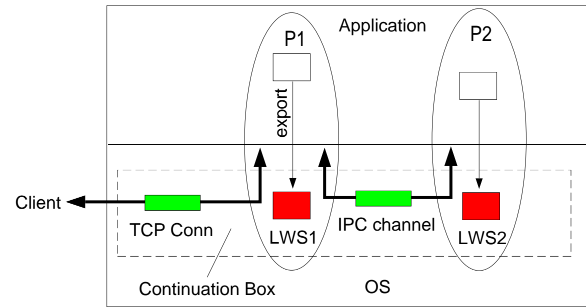

Figure 3: *The OS view of a continuation box: application-specific state and the state of communication channels.*

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 model is described by Figure 3, where the process service set $\{P_1, P_2\}$ handles the service session of one client. Only a select process in the set, the *accepting* process (P_1 in Figure 3), communicates directly with the client. 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.

For a server application, a CB 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 the client. For the OS, a CB is an ordered set of per-process light-weight state components. To support seamless communication across a failure, the OS also includes in a CB the state of stateful communication channels (inter-process and client-server).

6.2.1 The CB 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 there will be no cycles to execute upcalls on a dead machine.

To solve this problem, we maintain CBs in OS memory and provide a minimal API that allows a server application to specify *light-weight snapshots* (LWS) from which it can safely resume execution in case of failure. A process must export/import a LWS to/from a CB, completely describing the point it has reached in the ongoing service session. The LWS contents are opaque to the OS and to the CB extraction protocol. Figure 3 describes the OS view of a CB consisting of continuation points of processes in the service set (the S_1 and S_2 snapshots), along with communication state in IPC channels and the TCP client connection.

The CB API establishes a *contract* between an application

process and the system. A process must execute the following actions: *(i)* export LWSs periodically during service; *(ii)* import the last LWS at the new server (during recovery) and resume service to client. In exchange, a remote OS: *(i)* extracts and reinstates the state of the CB 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 CB API are:
cb = cb_create(conn)
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), `conn` is the client connection in the root process, and `state_buffer` is an application memory buffer holding the LWS of the client session state in a process. The accepting process creates a CB using `cb_create` on the accepted client connection. Processes in the service set use `cb_export_state` to save LWSs to a CB and `cb_import_state` to retrieve them (only once) for recovery after a failure.

The OS of the new server must synchronize the two endpoints of a communication channel after reinstating a CB. To achieve this, we adapt a mechanism previously developed by us in [37] which is based on a *limited* form of log-based rollback recovery [13]. The synchronization mechanism restores the session state in a process of the new server *with respect to a client* by replaying all communication of the process with peer processes or with the client using logs of communication activity in the OS. To implement state extraction from a dead machine, we adapt the mechanism in two ways: we make it transparent to client (i.e., we do not require the client OS/application to be modified in order to coordinate with the server during recovery) and we eliminate its dependency on the CPU/OS of the failed node (since CB extraction must not involve the failed node).

The CB model of session recovery incurs low-overhead during failure-free execution and enables fast recovery (as we show in Section 10) 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 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 retransmission by the client TCP of packets lost during the failover.

6.2.2 Example: Session Recovery in a Web Server

We illustrate the use of the CB API with excerpts from a recovery-enabled Apache [2] web server (we limit the example to code esential to a static transfer).

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 conection, 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 [27], a streaming audio server [18], and the auction service [9] described in Section 9.

7 Discussion and Limitations

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 even after a failure or attack 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 this paper describes a prototype and a case study 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 are currently exploring ways to build backdoors over wide-area and by using standard access interfaces like USB. One promising direction is the IETF effort for development and standardization of remote memory access over the Internet [3].

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 over application-controlled LWSs (Section 6) can prevent it from injecting accidentally corrupted state into another healthy system, while moving a whole process context or VM would reinstate *all* "bad" state it may encapsulate.

Fine-grained memory protection through software and/or hardware support has been studied in [39, 11, 40]. 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. The 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 [36], synthetic fault-injection tests [11], and examination of problem report databases for open-source kernel development [16] 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.

8 Prototype

We have implemented a BD prototype in the FreeBSD 4.8 x86 kernel, using Myrinet Lanai-XP programmable NICs [24]. 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 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, an infrequent event, the penalty from not caching is paid only once.

Remote OS locking. 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. 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 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. We used remote OS locking during recovery to freeze a sus-

pected target and completely eliminate unwanted effects of false positives in failure detection.

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 10 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 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.

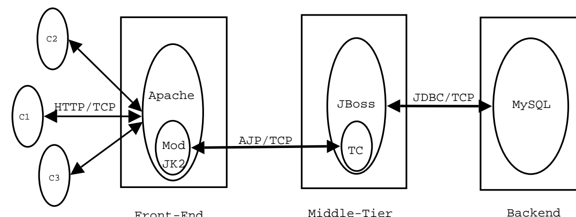

Figure 4: *RUBiS application architecture.*

9 Case Study

As a test application for our system we chose RUBiS [9], 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 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 backend.

System Configuration. Figure 4 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 backend 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 have Backdoors support. 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 RUBiS.

When a request enters the system, the FE tags it with a glob-

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

Table 1: *Cost of CB system calls and CB state extraction.*

ally 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.

10 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 Lanai-X NICs with a 133MHz PCI-X interface [24].

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 counters on number of interrupts and context switches.

10.1 Microbenchmarks

SB/CB API Overhead and CB Extraction Cost. In the first experiment, we evaluate the run-time overhead of the SB and CB API by measuring the latency of the calls described in Section 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 1 (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.

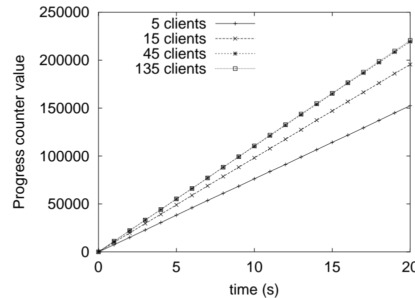

Figure 5: *Variation of an OS interrupt counter with time under different load conditions.*

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 1 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 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 SB, comparing counter values, etc.), and *(ii)* 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 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.

Counter Dynamics. To study dynamics of OS progress counters, we collected traces of a progress counter for the

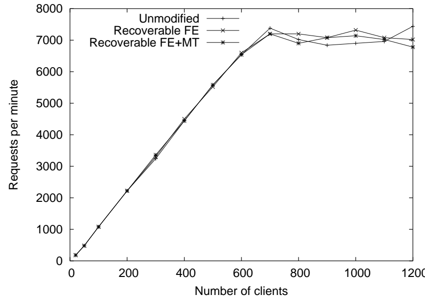

Figure 6: *Throughput of recoverable RUBiS is unaffected by recovery support.*

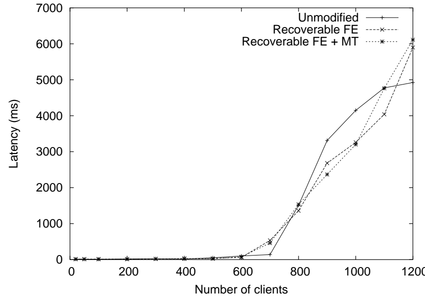

Figure 7: *Latency of recoverable RUBiS is unaffected by recovery support.*

number of interrupts serviced in a system running a synthetic streaming server. Figure 5 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. The experiment outlines 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.

10.2 Real Applications

We have used our BD-based system to support recoverable sessions in several open-source servers [2, 27, 18] 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 9. 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 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 [9], 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 6 and 7 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 90 s, with normally distributed think-times with 7 s mean and a slowdown factor of 0.5 (see [9] 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 8 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. 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

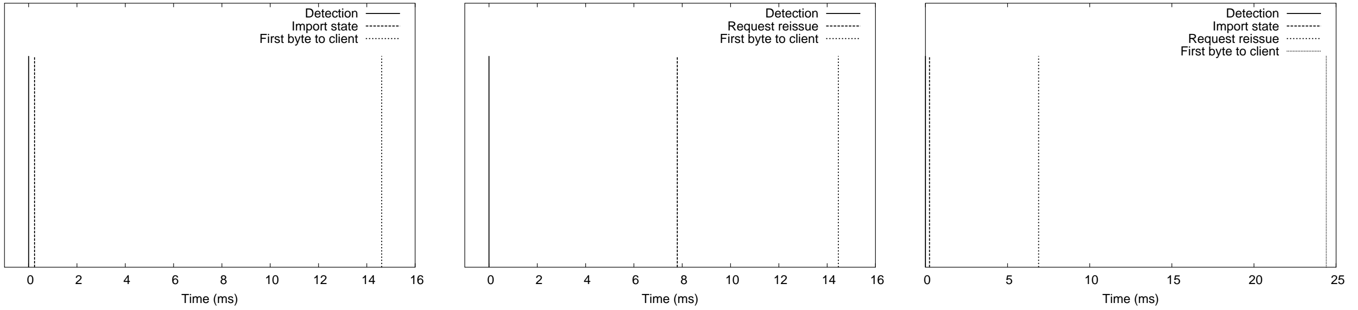

Figure 8: *Recovery timeline across an FE, MT, and FE+MT crash (left to right), in the worst case (for the last recovered session). The crash occurs at -10 ms. The worst-case recovery latency is under 25 ms.*

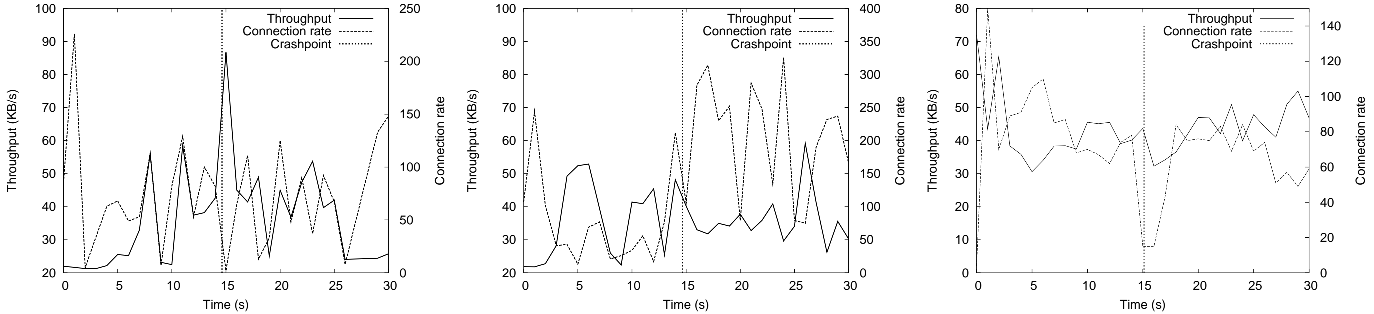

Figure 9: *Aggregate throughput and connection rate seen by clients across an FE crash, MT crash, and FE+MT crash (from left to right). Vertical lines mark the moment of the crash.*

performance.

Figure 9 shows the variation of aggregate throughput (as bytes received by all clients) and the rate of established client 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. The jittery throughput is a well-known problem of the RUBiS client [8], and our concern was that such a ”jumpy” workload profile would obscure the effects of fault handling. In fact, in Figure 9 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 bytestream 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, 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.

11 Conclusions

We have described Backdoors, a novel system architecture that enables applications to survive software failures that render a computer system unavailable (system hangs, OS crashes, deadlocks, etc.) 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, and defines OS extensions for remote access to light-weight application and OS state. Using Backdoors, a remote system can perform accurate monitoring without CPU overhead on the target system and, upon detecting its failure, can extract and reinstate light-weight state from the failed system.

We describe the design of Backdoors OS abstractions that support remote nonintrusive monitoring (Sensor Box) and recovery of critical application and OS state (Continuation Box). We show how Backdoors can be used to enable nodes in cluster-based Internet servers to perform mutual monitoring of their liveness, and to take over client sessions from failed nodes.

We have implemented a Backdoors prototype in FreeBSD and present a case study and results of an experimental evaluation with a complex, transactional, multi-tier Internet service. We show 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. The code for Backdoors and test applications is available from http://discolab.rutgers.edu/bda.

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] 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.

[4] 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.

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

[6] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote Repair of OS State Using Backdoors. In *Proc. Int'l. Conference on Autonomic Computing*, May 2004.

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

[8] G. Candea, Feb. 2004.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Computers*, 52(2):99–112, 2003.

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

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

[17] I. Gupta, T. Chandra, and G. Goldszmidt. On Scalable and Effi cient Distributed Failure Detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, Apr. 2001.

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

[19] The Infi niband Trade Association. http://www.infi nibandta.org, August 2000.

[20] 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.

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

[22] 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.

[23] J. Mogul. Personal communication, June 2003.

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

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

[26] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Dec 2002.

[27] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Effi cient and Portable Web Server. In *Proc. USENIX Annual Technical Conference*. USENIX Association, June 1999.

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

[29] 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.

[30] C. P. Saputnzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[31] 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.

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

[33] E. G. Sirer, D. Becker, M. Fiuczynski, 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.

[34] 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.

[35] C. Soules et al. System support for online reconfi guration. In *In Proc. USENIX Annual Technical Conference*, June 2003.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct. 2002.

[41] 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.

[42] 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.