Recovering Internet Service Sessions from Operating System Failures

Current Internet service architectures lack support for salvaging stateful client sessions when the underlying operating system fails due to hangs, crashes, deadlocks, or panics. The Backdoors (BD) system is designed to detect such failures and recover service sessions in clusters of Internet servers by extracting lightweight state associated with client service sessions from server memory. The BD architecture combines hardware and software mechanisms to enable accurate monitoring and remote healing actions, even in the presence of failures that render a system unavailable.

Florin Sultan, Aniruddha Bohra, Stephen Smaldone, and Yufei Pan
Rutgers University

Pascal Gallard
IRISA/INRIA

Iulian Neamtiu
University of Maryland, College Park

Liviu Iftode
Rutgers University

Critical Internet services such as e-commerce, online auctions, and banking run on complex, multi-tier architectures built with commodity (off-the-shelf) machines and operating systems. These stateful services are sensitive to server failures: active client sessions on these servers are lost, although the state associated with them might still be intact in a failed machine's memory.

We developed a recovery approach that exploits hardware and software redundancy in Internet service installations to reuse active clients' session state after OS failures (http://discolab.rutgers.edu/bda). Our lightweight, application-independent system provides both failure detection and recovery, for use with complex, multi-tier Internet services. The core of the system is the novel Backdoors (BD) architecture,[1] which uses commodity programmable network interface cards (NICs) with specialized firmware and OS extensions to provide remote access to lightweight application and OS state in a machine's memory without relying on its OS or processors. Using BD, machines in an Internet server cluster can cooperatively observe each other's health, detect failures, and take over client sessions from failed nodes.

In this article, we describe the BD architecture and our OS extensions for monitoring and recovery of service sessions. We have implemented a prototype in the FreeBSD 4.8 kernel, using Myrinet Lanai-XP programmable NICs (www.myri.com). The results from our experiments with the Rice University Bidding System (Rubis; http://rubis.objectweb.org), a cluster-based

 Published by the IEEE Computer Society 1089-7801/05/$20.00

The Backdoors Architecture

According to the Telecom Glossary 2000 ANSI standard, a backdoor is "a hidden software or hardware mechanism, usually created for testing and troubleshooting."[1] The Backdoors architecture goes a step further by providing an alternate path into a system to enable automated remote healing (recovery or repair) operations.

To implement BD, we use intelligent (programmable) network interface cards (I-NICs) with remote direct memory access (RDMA) capability, which allows a machine to access another's memory for reading and writing without involving its processor(s). RDMA makes BD nonintrusive to system activity during normal operation, and robust to OS failures.

Figure A shows how our model partitions a computer between *front-door* components, which are under the OS's control, run OS or application code, and interact with the outside world, and *backdoor* components, which are involved in monitoring and recovery operations. Our crucial assumptions are that the backdoor hardware remains available after OS failures and memory contents remain valid and accessible over the system bus.

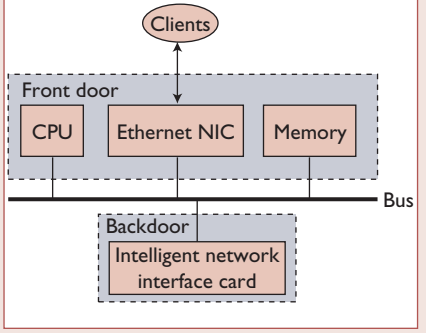

Figure A. System equipped with a backdoor intelligent network interface card. The backdoor I-NIC can access its host system's resources without using the machine's processor(s).

Reference

1. *Telecom Glossary 2000 T1.523-2001*, Am. Nat'l Standard Institute, 2001; www.atis.org/tg2k.

multi-tier Internet auction service modeled after eBay, indicate that our approach is nonintrusive and effective. Indeed, the system can recover all service sessions from failed nodes in both the front-end and middle tier within 25 milliseconds.

Motivation and Approach

Today's Internet services employ servers organized in clustered multi-tier architectures in which multiple nodes perform processing for a given client session:

- front-end nodes handle HTTP requests,
- mid-tier nodes implement application logic, and
- back-end nodes run database servers.

Machines in all tiers run commodity, general-purpose OSs, which typically cannot tolerate failures caused by OS bugs or misconfiguration.

An OS failure renders an entire system unusable because applications depend on core OS services for memory allocation, process management, and I/O. For noncritical or stateless services, rebooting would be sufficient for recovery. Moreover, if the service is *idempotent* — generating the same outcome in response to multiple copies of a given request — clients can recover by simply reissuing their requests. Yet, the reboot approach presents at least two problems:

- reboots are destructive to currently executing transactions, forcing the clients to reissue them, and
- reboots are disruptive, incurring downtime for both the service provider and clients.

While most applications and their clients can tolerate the side effects of a reboot, such an approach can be unacceptable for the critical, transaction-oriented services. Depending on a server's load-balancing and admission policies, clients are not guaranteed readmission to resume their sessions. In addition, the service might provide guarantees that include uninterrupted delivery — at least to the extent the network permits.

We designed the BD architecture to support *remote healing*,[1,2] using remote memory access to detect failures and perform automated recovery actions. BD relies on a specialized network interface that allows external access to a computer's resources (memory, I/O devices, and so on) without involving its processors or OS. This allows the recovery actions to be deferred until after the failure has occurred, adds only negligible overhead during normal (failure-free) operation, and provides fast recovery. (For an overview, see "The Backdoors Architecture" sidebar.)

Remote Recovery with Backdoors

Remote recovery with BD addresses system-hang failures, in which a server cannot execute useful work because the OS is unresponsive. Such failures are caused by faulty OSs — for example, due to drivers that leave interrupts disabled, deadlocks, or misplaced panics. Our goal is to reliably detect

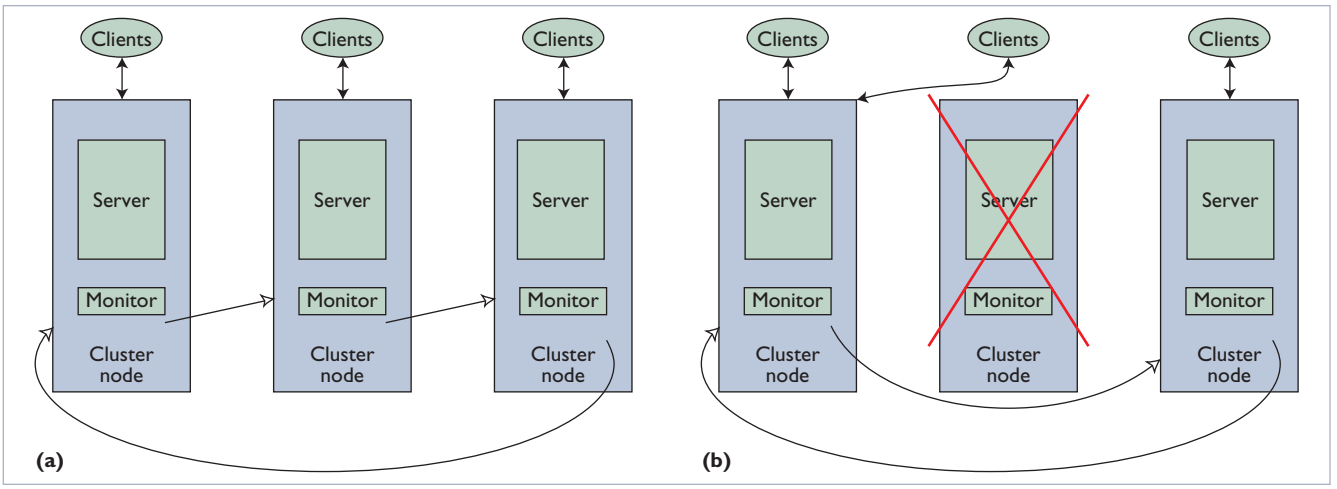

Figure 1. Monitoring and recovery in an Internet service's front-end tier. (a) Identical front-end servers run on nodes that also perform mutual monitoring in a virtual ring. (b) A monitor detects a node failure, relocates client sessions, and reconfigures the ring.

these failures and salvage the critical software state residing in the OS memory. However, the OS's unavailability prevents the execution of monitoring and recovery actions on the affected machine.

The BD system detects node failures via mutual monitoring between nodes in a service tier. As Figure 1a illustrates, each node runs a monitor process that inspects the OS state of its successor in a virtual ring.

Upon detecting a failure, a recovery node in the same tier takes over the client sessions serviced by the failed node. It extracts the critical state from the failed node and injects it into its own memory. As all server nodes run the same application, the recovery node can seamlessly use the extracted state to continue client service (Figure 2b). This also requires server nodes to have access to the same resources (external file servers, databases, and so on).

Operating System Support

To facilitate remote operations over BD, we designed two extensions to an existing OS to support efficient monitoring and extraction of state from a failed system.

Monitoring

For system-hang failure detection, BD uses the *sensor box* (SB), an OS mechanism we developed in previous work.[2] The SB allows a monitor to observe a target system's "liveness" using only remote memory reads.

An SB is a region in the target's OS memory, structured in records called *sensors* — configurable tuples <*ID*; *C*; *L*; *V*>. *ID* is a unique identifier; *C* is a class of sensors; *L* is a limiting threshold that depends on the sensor class; and *V* is a scalar (the actual sensor). Each monitor machine's OS provides an API for remotely reading (fetching) a target SB through BD; the monitoring function can thus be implemented through a user-level process.

We can define several sensor classes for detecting anomalous events in a system.[2] In particular, *progress* sensors (monotonically increasing health meters, which a live OS is supposed to continuously update) are best suited to detecting failures. Upon creating a progress sensor, a target system specifies its detection deadline *L*. The target OS then signals to its monitor that it is still alive by increasing the sensor's value *V* at intervals smaller than *L*. The monitor periodically fetches the target's SB and scans it, comparing the current and previous values for each progress sensor. If a critical sensor's value does not change within the deadline interval *L*, the target system is considered failed.

The monitor can ensure that all activity ceases on a failed node by performing a remote OS-locking operation on the suspected node before initiating recovery. Remote OS-locking stops scheduling and interrupt handling in the target system, effectively freezing it. This eliminates side effects from potential false positives in failure detection.

Session Recovery

Recovery of live sessions is challenging because Internet services are usually structured as multiple communicating processes running on multiple machines. Moreover, their states are highly

dynamic and they operate under heavy client loads. To support recovery, we developed the *continuation box* (CB), an OS abstraction that encapsulates fine-grained state associated with a service session. The CB concept relies on the observation that most of the software machinery involved in providing services (OS, servers, applications, file and database access mechanisms, and so on) is already replicated on the other peer nodes in a tier. This means that, after a failure, we must salvage only the "essential" application and OS state for the individual client sessions. The CB leverages our previous work on *service continuations* for efficient migration of live service sessions between equivalent servers.[3]

A server application can use CBs to record lightweight snapshots (LWSs) of session state. An LWS must completely describe a well-defined point from which a server process can safely resume an ongoing service session after failure. In addition to maintaining LWSs, CBs track state for the server application's communication channels (interprocess and client-server). For example, a CB for a client performing a static HTTP transfer will contain an LWS with the file name and the offset reached in the transfer, and will track the client-server TCP connection's state. Figure 2 shows the OS-specific components and application-specific LWSs in a CB that spans two server processes communicating over interprocess communication (IPC) channels. LWSs are opaque to the OS and the CB extraction protocol.

The CB provides the following API, which allows server processes to export and import state snapshots:

```
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 session (an OS-specific identifier), `conn` is the client connection, and `state_buffer` is an application memory buffer that holds the LWS of the client session state in a process. The server process that accepts the client connection creates a CB using `cb_create`. During execution, all server processes use `cb_export_state` to save LWSs to a CB. In case of a failure, a recovery node in the same tier extracts the CB from the failed machine's memory. Server processes on the recovery node then use `cb_import_state` to retrieve LWSs and continue service.

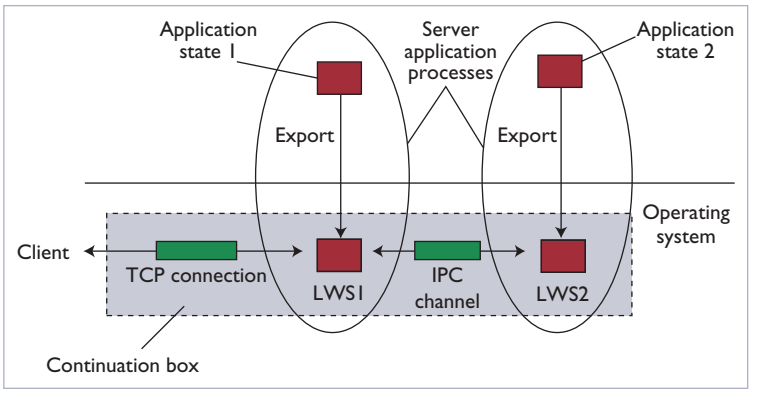

Figure 2. Continuation box. A CB maintains application-specific state components (in red) and the state of communication channels (in green) associated with one client session. While servicing the client, server processes periodically export lightweight snapshots (LWSs) of application state to the CB.

The CB API establishes a contract between a server process and the system. A process must export LWSs periodically (during service) and import the last LWS at the recovery node (during recovery). In exchange, the OS of the recovery node that takes over the client session extracts and reinstates the CB locally and then synchronizes the server processes' state with that of their communication channels. (see the "Log-Based Continuation Box Synchronization" sidebar.)

Case Study: A Recoverable Auction Service

We have implemented the CB mechanism, including support for TCP connections and OS pipes, in the FreeBSD 4.8 kernel. A CB is associated with the socket corresponding to a client and contains pointers to state components (LWSs and communication logs) and log control information (read/write pointers).

For remote monitoring and state extraction, we modified the Myrinet GM 2.0 library to provide in-kernel remote memory read/write operations. To ensure atomic access to a system's state and safely fail-stop a suspected system, BD provides a remote OS-locking operation, which blocks all scheduling and interrupt processing in the remote OS.

A front-end recovery node extracts an established client TCP connection from a failed machine and uses IP address takeover to make the fail-over transparent to the client's TCP. Once the recovery node has extracted the TCP endpoint, it injects this state into the local TCP/IP stack as a new connection placed on a server socket's accept

Log-Based Continuation Box Synchronization

A recovery node's operating system uses a limited form of log-based rollback recovery to synchronize session state in server processes with the current state of the related communication channels.[1] The OS maintains logs of communication activity and uses them to replay a process's communication with peer processes (via bytestream interprocess communication channels) or with the client (over TCP). Logging incurs no CPU overhead because it uses the communication channels' data buffers, which the OS maintains already.

When extracting the continuation box (CB), the recovery node's OS first determines what portion of the failed node's log is necessary for application-level replay of receive and read operations. During the replay, server processes read data from logs and change the session state to match the failed node's state before failure. Similarly, the OS computes the number of bytes already received at the other end of a communication channel and discards duplicate bytes generated during the replay of send and write operations.

Figure B1 shows an example execution replay in which a reader process *R* was behind a writer process *W* in exporting its lightweight state snapshot (LWS) to the CB before the failure occurred. The vertical bar represents a virtual infinite buffer abstracting the channel over which the two processes communicate. The moment a process exports an LWS, the snapshot is labeled with the sequence number of the first byte it will read or write after the snapshot. For *R*, *lsr* denotes the first byte to read after taking its last snapshot LWS_r. Similarly, *lsw* denotes the first byte to be written by *W* after its last snapshot LWS_w. Given that *lsr* < *lsw* in Figure B1, *R* will issue reads for data that *W* can no longer supply upon resuming service at the recovery node.

To support the replay of data read from a communication channel before a failure, the system extracts data from the failed node that the writer at the recovery node cannot generate. In the case depicted in Figure B1, the synchronization log includes only the [*lsr*; *lsw* – 1] region of the buffer — from the *lsr* byte up to, but excluding, the *lsw* byte (the first to be regenerated by *W*'s replay). The recovery OS can thus ignore data for sequence numbers less than *lsr* because *R* will not need them after restart. The recovery OS can also ignore data in the interval [*lsw*; *failure*] because *W*'s replay will regenerate it after restart.

Figure B2 shows an example in which the reader was ahead of the writer in taking its last snapshot before failure (*lsr* > *lsw*). During replay, *R* will read only from sequence numbers (starting with *lsr*) that *W*'s replay can regenerate. Therefore, the recovery OS does not need to extract read logs from the failed node (when *R* took its snapshot at *lsr*, the system actually discarded any previous log). During replay, the recovery OS discards any write issued by *W* in the interval [*lsw*; *lsr* – 1].

Reference

1. F. Sultan, A. Bohra, and L. Iftode, "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions," *Proc. Symp. Reliable Distributed Systems* (SRDS), IEEE CS Press, 2003, pp. 177–186.

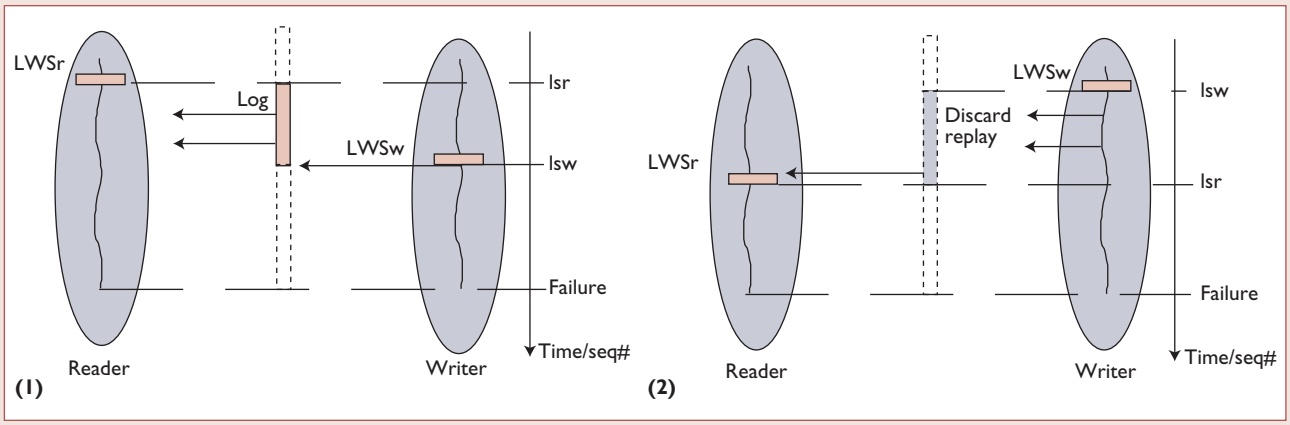

Figure B. Synchronization of communication channel endpoints. Synchronization during recovery replay depends on the relative positions of reader and writer processes upon resuming service from their last lightweight snapshots (LWSs) of state. (1) If the reader was behind the writer before failure, the OS replays reads. (2) If the writer was behind the reader, the OS discards duplicate writes.

queue. A listening server process accepts it, imports the LWS from its associated CB, and resumes service to the client.

We have used our BD-based system for session recovery with Apache and Flash Web servers and the Icecast audio-streaming server (www.icecast.org). To test the system, we chose the Rubis online auction service. This complex application uses a multi-tier architecture to support item selling and bidding, user accounts, and user ratings and com-

ments. The typical workload is a mix of browsing and updating of persistent data.

Base System

We run Rubis in a three-tier architecture with Web servers on front-end nodes (FEs), application servers in the mid-tier nodes (MTs), and a transactional database system (DB) on back-end nodes. All FE and MT nodes run FreeBSD 4.8 with our SB and CB extensions. We run Apache 2.0.47 on FEs; the Tomcat 4.1 servlet container and JBoss 3.2.2 Enterprise JavaBeans (EJB) server on MTs; and MySQL 4.0.15 as the DB server. Client requests enter the system at the FE, pass from Apache to the specified application servlet running on the MT in the Tomcat container, and then on to JBoss, where Rubis EJBs implement the application logic. From there, the application queries the DB server.

In Rubis, bidders' typical behavior (placing last-moment bids) makes their requests critical in the moments before an auction closes. If a node on the request path fails, reissuing a client's request risks missing the deadline, generating a duplicate transaction, or failing to be readmitted to an overloaded system.

Recoverable Rubis

To make Rubis recoverable, we ran it on a cluster in which we installed BD on the FEs and MTs. We assume the back-end nodes achieve fault-tolerance through well-known methods such as database replication. We modified Apache and the Rubis beans to use the CB API, adding only 500 lines of code to Apache and 30 lines to Rubis, compared to code bases of 15,000 and 36,000 lines, respectively. The programmer identifies points of nondeterminism and then exports nondeterministic state changes. To implement our changes, we had to understand the way the servers and Rubis application work. Programmers who are knowledgeable about their servers' and applications' internals should be able to integrate BD recovery support with minor effort.

A client interacts with the service by sending HTTP requests to an FE node. When a client's request enters the system, an FE tags it with a globally unique request ID, which identifies a given session's CB-encapsulated state. On an FE node, a session's LWS contains the request, its ID, and the offset reached in the output stream sent to the client. On an MT node, where the request is translated into a DB transaction, the LWS contains the request ID, transaction identifier, and transaction result (one database record). These LWSs average only 99 and 44 bytes in FE and MT nodes, respectively.

Table 1. Cost of continuation box (CB) system calls and state extraction.

State size (Kbytes)	Export (μs)	Import (μs)	CB extraction (μs)
1	11	8	158
5	20	10	258
10	28	24	358

If an FE node fails, its monitor notifies a recovery FE node 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 transaction status (`abort` or `commit`) from the database and retrieves the transaction result from the CB. The MT node uses this information to decide whether to reissue the transaction, or to rebuild the reply to the client if the transaction has already been executed.

This scheme relies on simple DB support for reconnects to preserve DB transactions while (re)generating replies. To implement it, we modified MySQL to support database reconnects and queries for transactions' status. Alternatively, we could have modified the database schema to include a table of transaction IDs along with the status. The application would have had to maintain the transaction status in this table during normal execution and query it during recovery.

Implications

Extracting critical state from a failed system using BD is a last-resort recovery action, suited for the most severe system-hang failures. However, our approach is limited to failures that do not corrupt memory: if a CB is corrupted, we cannot recover its corresponding session. Yet, the evidence from field-error data,[4] synthetic fault-injection tests,[5] and problem-report databases for open-source kernel development (www.freebsd.org/cgi/query-pr-summary.cgi) suggests that memory corruption is a fairly rare and localized event when an OS fails.

Our fine-grained recovery model is potentially more robust to propagation of such corrupted (bad) state than heavy-weight approaches, which recover large amounts of unstructured state (checkpointing, process migration, virtual-machine migration, hot backups, and so on).

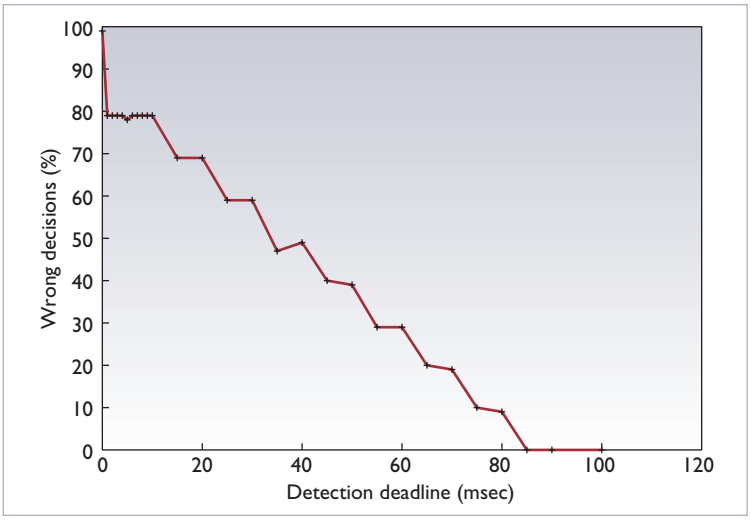

Figure 3. False positives in failure detection. Context-switch counters with detection deadlines less than the scheduler time slice (100 milliseconds) cause monitor nodes to generate false positives in error detection; their frequency increases at shorter deadlines.

Smaller state components enable recovery of "good" state by identifying and filtering occurrences of bad state. For example, computing checksums over application LWSs can detect corrupted state and prevent the injection of a single bad CB into another healthy system. In contrast, moving a whole process context or virtual machine indiscriminately reinstates all the bad state it might encapsulate.

Experimental Evaluation

We evaluated our system's performance and correctness in a multi-tier setup consisting of two FEs, two MTs, and one back-end node, all of which were Dell PowerEdge 2600 2.4-GHz dual-processor machines with 1-Gbyte RAM, interconnected by 1 Gbit/sec Ethernet, and running FreeBSD 4.8. The FEs and MTs run our BD prototype, implemented with the Myrinet Lanai-XP NICs.

Nodes in each tier monitor each other and detect failures using progress sensors that track the number of interrupts and context switches. We injected OS failures in one node in each tier and recovered sessions serviced by the victims on the alternate nodes in the same tier. For fault injection, we introduced bugs in Ethernet network drivers to cause system crashes. We also subjected victim nodes to synthetic failures (processor halt, disabling the interrupt controller, or disabling device interrupts) or froze them by entering the kernel debugger from the console at random moments in time.

API Overhead and Extraction Cost

We measured call latency to evaluate the runtime overhead incurred by updating the SB and using the CB API. Updating sensor values in the SB is extremely lightweight as it simply involves writing integer values to memory. Table 1 shows the cost of `export` and `import` CB calls for three different CB state sizes. It shows that monitoring and recovery support should be lightweight on server nodes because the CB API imposes little runtime overhead for updating and retrieving CB state.

The last column in Table 1 shows the cost of extracting a CB, consisting of a TCP connection and an application LWS, from an FE node. For comparison, the raw latency of a remote memory read was roughly 16 μs for 4-byte payloads and 118 μs for a 10-Kbyte payload.

Monitoring Overhead

A monitor node's overhead includes the monitoring cost (reading the local view of the monitored SB, comparing counter values, and so on) and the cost of transferring the remote SB from the monitored node. We measured a monitor process's CPU usage while varying the sampling rate of a remote SB with 100 progress counters. In the worst case, when sampling the SB in an infinite loop, CPU usage was 46 percent. Sampling at the timer's lowest granularity (10 milliseconds), CPU usage was roughly 5 percent, and it dropped below 1 percent for 100-millisecond sampling intervals. This shows that monitor nodes can perform fast failure detection while maintaining low overhead.

Detection Deadlines and False Positives

To avoid false positives at a monitor — wrongly declaring a healthy node to have failed — we must carefully choose each progress counter's detection deadline L to match the counter's expected behavior.

A trade-off exists between fast failure detection and a low rate of false positives (ideally, 0). To illustrate, we artificially induced false positives using a progress counter based on the number of scheduling decisions. A remote monitor sampled the counter at intervals equal to its detection deadline while a CPU-bound task ran on the node. The task does not block; thus, no scheduling decision can occur within a time slice. When the detection deadline is smaller than the time slice (100 milliseconds), the counter could stall and lead the monitor to declare the node faulty.

Figure 3 shows the fraction of false positives

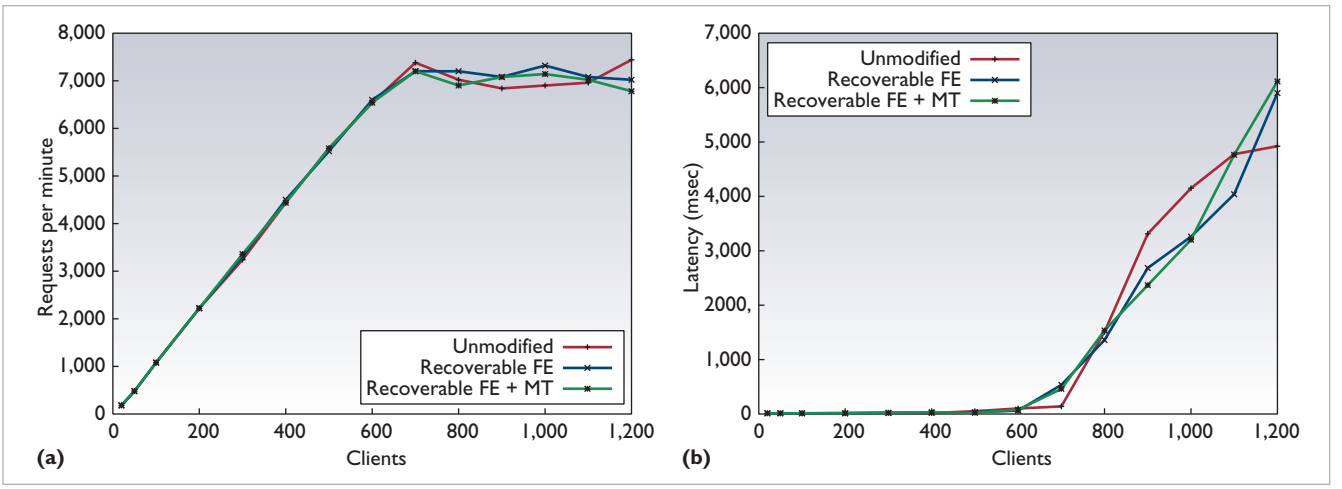

Figure 4. Backdoors overhead. Backdoors' recovery support does not affect (a) throughput or (b) latency in the recoverable version of the Rice University Bidding System (Rubis).

that occurred as we increased the detection deadlines — in turn, increasing the chance that the OS would make a scheduling decision before the deadline. Deadlines of less than the scheduler time slice induced incorrect detection, whereas scheduling activity in the system eliminated false detection for deadlines of more than 85 milliseconds. This shows that the system is sensitive enough to expose errors in failure detection caused by unrealistic detection deadlines.

Failure-Free Overhead

To estimate the impact of BD's recovery support on client-perceived performance, we ran load experiments on Rubis using both an unmodified "base" system and a recoverable version in which servers (Apache in FE and JBoss in MT) used the CB API. Using a methodology similar to that used by Cecchet, Marguerite, and Zwaenepoel[6] and *think times* (intervals between successive requests) as specified by the Transaction Processing Performance Council Web benchmark (TPC-W; www.tpc.org/tpcw/), we created a workload that emulates client browsers.

Figure 4 shows the aggregate throughput and latency perceived by clients for the base case, recoverable FE, and recoverable FE and MT. All three systems have identical behavior: the curves overlap when the system is underloaded and show small variations due to nondeterministic system behavior after saturation, which demonstrates that recovery support imposes no performance penalty.

Fail-Over Correctness

We evaluated the system's correctness through crash-test runs under a load of 200 clients generating a heavy synthetic workload. Each request performs a database transaction consisting of multiple queries and an update on the same table. After each run, we checked session fail-over correctness via two types of tests:

- *Communication-path integrity* tests ensure that every client request is correctly matched by its expected reply.
- *Database integrity* tests ensure that the database includes no missing or duplicate transactions.

All runs validated our system's correctness, as each client request received the correct reply and every database transaction completed properly.

Fail-Over Latency

To evaluate the impact of failure detection and recovery on client-perceived performance, we ran crash tests under a workload of 200 clients. Each client generated browse transactions in 90-sec runs with normally distributed think times with 7-sec mean and a slow-down factor of 0.5.[6] With this workload, CPU utilization is about 45 percent on FE nodes and 15 to 30 percent on MT nodes. We used the number of network interrupts and context switches as progress counters with a failure-detection deadline of 10 milliseconds (the timer granularity), and emulated crashes in FE, MT, and both.

Figure 5 shows the event timeline from crash detection to the end of recovery for the last recovered session. We define the end of recovery for a session as the moment the FE sends the first

Related Work

Operating system support for fast recovery helps improve the availability of systems and services. Previous research has employed three techniques to achieve fast recovery: protection from crashes through fault isolation, reuse of in-memory state across crashes, and fast reboot of failed software components.

In Recovery Box, Baker and Sullivan use a stable OS memory area that can survive a crash to restore OS subsystems' live state after reboot.[1] Their approach relies on transactional semantics to ensure application recovery with remote client sessions. Yet, recovery is not transparent to clients, which must reconnect and resubmit requests. The system does not support complex applications consisting of multiple communicating processes.

Nooks uses code interposition and virtual memory techniques to sandbox faulty kernel-loadable modules.[2] It can detect faults that occur in extensions and involve memory accesses, bad call arguments, or live lock (when the machine spends an inordinate amount of time executing the extension code, at the expense of other processes). In contrast, our Backdoors system cannot detect memory-access failures unless they crash the system. However, it can detect failures regardless of where they occur in system code and handle failures triggered by factors other than system software.

In the Microreboot approach to fast failure recovery, a software system, such as a Java application server (JBoss), is augmented to selectively discard and refresh the state of discrete failed software components in the middleware.[3] The approach relies on efficient external state stores to save components' volatile state across microreboots.

Liao and colleagues use a Myrinet network interface card (NIC) to monitor the performance of a shared virtual memory system.[4] Zhou, Chen, and Li use RDMA to mirror the address space of processes on other nodes in a cluster,[5] which makes user-level checkpointing and fail-over faster than using disks as stable storage. Yet, because the system concentrates only on checkpointing user-level state, without considering the state of active communication channels, it cannot be used for fail-over of Internet services.

Rio uses software fault isolation to protect the file-system cache against corruption during crashes. The system uses the cached data to warm-reboot the file system.[6] Backdoors takes a similar approach in using the system memory as storage for recovery data, additionally providing the support to seamlessly reuse this data on other healthy systems. Unlike Rio, however, the current Backdoors implementation does not provide fault isolation, which makes it vulnerable to memory corruption in a crash.

When an OS failure affects an Internet server with open TCP connections, the connections must survive the crash transparently to the clients. Several research systems focus on TCP servers' reliability. Fault-Tolerant TCP uses TCP wrapping to mask server failure and restart,[7] but its use of heavyweight single-process checkpointing for recovery makes it impractical for Internet services. Snoeren, Andersen, and Balakrishnan employ fine-grained fail-over

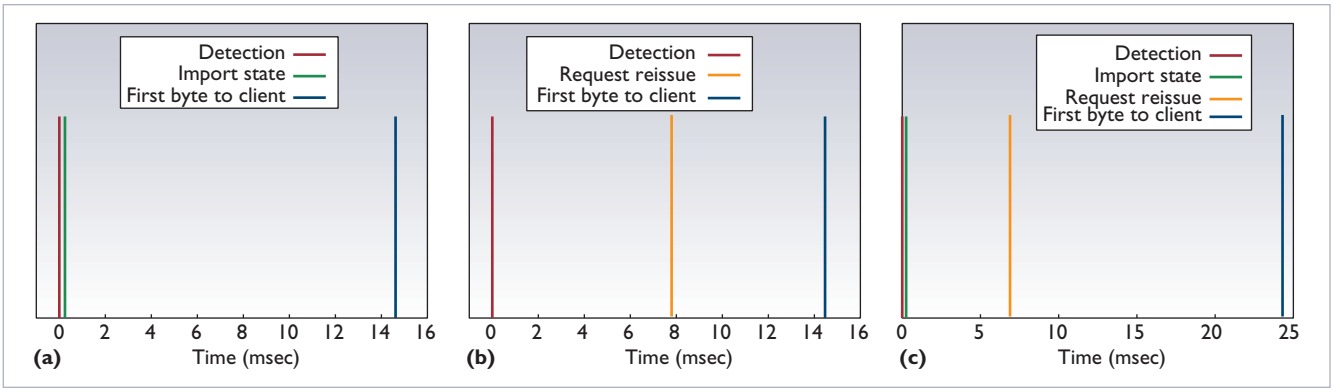

Figure 5. Fail-over event timeline. The (a) front-end and (b) the mid-tier nodes require less than 15 msec from crash detection to resumption of service to the last-recovered session. (c) A crash on both the front-end and mid-tier nodes exhibits the worst recovery latency, which is still less than 25 msec.

byte to the client after fail-over. Detection latency is limited by our choices of detection deadlines and sampling period. The worst-case recovery latency is less than 25 milliseconds when both nodes fail, indicating that our fail-over mechanism should have no practical effect on client-perceived performance.

The low fail-over latency compares well to the effects of packet loss on normal client-server TCP traffic over the Internet. Recovery introduces a "gap" in the outgoing server bytestream comparable to round-trip times between two systems

Related Work, cont. from p. 32

using connection migration in cluster-based Web servers.[8] However, their scheme is limited to static HTTP transfers and relies on broadcasts of recovery state inside the cluster. Bressoud and Schneider use virtual machine monitors to intercept and back up the entire state of a system, which requires dedicated machines and imposes high performance penalties.[9]

Primary-backup schemes use active remote logging[10] or passive traffic tapping at the link-layer[11,12] to mirror TCP servers' communication and computation state on other machines. These schemes require fully dedicated nodes as backups and use interposition techniques that add overhead and increase communication latency, thus affecting the failure-free execution performance. They do not tolerate loss of the backup unless it uses some form of logging support that is available after failure.[10]

References

1. M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment," *Proc. Summer Usenix Conf.*, Usenix Assoc., 1992, pp. 31–44.
2. M.M. Swift, B.N. Bershad, and H.M. Levy, "Improving the Reliability of Commodity Operating Systems," *Proc. 19th Symp. Operating Systems Principles* (SOSP), ACM Press, 2003, pp. 207–223.
3. G. Candea et al., "Microreboot: A Technique for Cheap Recovery," *Proc. 6th Symp. Operating Systems Design and Implementation* (OSDI 04), Usenix Assoc., 2004, pp. 31–44.
4. C. Liao et al., "Monitoring Shared Virtual Memory Performance on a Myrinet-Based PC Cluster," *Proc. ACM Int'l Conf. Supercomputing*, ACM Press, 1998, pp. 251–258.
5. Y. Zhou, P.M. Chen, and K. Li, "Fast Cluster Failover Using Virtual Memory-mapped Communication," *Proc. 13th Int'l Conf. Supercomputing*, ACM Press, 1999, pp. 373–382.
6. P.M. Chen et al., "The Rio File Cache: Surviving Operating System Crashes," *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), ACM Press, 1996, pp. 74–83.
7. L. Alvisi et al., "Wrapping Server-Side TCP to Mask Connection Failures," *Proc. IEEE Infocomm*, IEEE CS Press, 2001, pp. 329–337.
8. A.C. Snoeren, D.G. Andersen, and H. Balakrishnan, "Fine-Grained Failover Using Connection Migration," *Proc. 3rd Usenix Symp. Internet Technologies and Systems* (USITS), Usenix Assoc., 2001, pp. 221–232.
9. T.C. Bressoud and F.B. Schneider, "Hypervisor-Based Fault Tolerance," *Proc. 15th ACM Symp. Operating Systems Principles* (SOSP), ACM Press, 1995, pp. 1–11.
10. D. Zagorodnov et al., "Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 03), IEEE CS Press, 2003, pp. 393–402.
11. S. Mishra, M. Marwah, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 03), IEEE CS Press, 2003, pp. 373–382.
12. R.R. Koch et al., "Transparent TCP Connection Failover," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 03), IEEE CS Press, 2003, pp. 383–392.

on the Internet (tens to hundreds of milliseconds over WANs), from which TCP derives retransmission timeouts for unacknowledged packets. Recovery's impact, as perceived by remote clients, should be no worse than when server packets are lost in the Internet.

Backdoors is an architectural approach to improving the survivability of computer systems. We believe this approach is applicable not only for recovery of Internet service sessions, but also for continuously monitoring and understanding system behavior in large installations. We are currently exploring software emulation of BD for virtual infrastructures, such as VMWare (www.vmware.com) and PlanetLab (www.planet-lab.org), to let us monitor multiple OS instances on a given system without additional hardware. Our ultimate goal is to involve multiple systems equipped with BDs to perform automated monitoring and recovery without semantic knowledge of the applications or OS, and without involving human operators.

Acknowledgments

This work is supported in part by the US National Science Foundation under NSF CCR-0133366 and ANI-0121416. Pascal Gallard and Iulian Neamtiu worked on the project as visiting graduate students at Rutgers University. The authors thank Christine Morin (IRISA/INRIA) and Arati Baliga for their feedback to help improve the article.

References

1. F. Sultan et al., "Nonintrusive Remote Healing Using Backdoors," *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, ACM Press, 2003, pp. 69–74.
2. A. Bohra et al., "Remote Repair of OS State Using Backdoors," *Proc. Int'l. Conf. Autonomic Computing*, IEEE CS Press, 2004, pp. 256–263.
3. F. Sultan, A. Bohra, and L. Iftode, "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions," *Proc. Symp. Reliable Distributed Systems* (SRDS), IEEE CS Press, 2003, pp. 177–186.
4. M. Sullivan and R. Chillarege, "Software Defects and their

Impact on System Availability: A Study of Field Failures in Operating Systems," *Proc. 21st Int'l Symp. Fault- Tolerant Computing* (FTCS-21), IEEE CS Press, 1991, pp. 2–9.
5. P.M. Chen et al., "The Rio File Cache: Surviving Operating System Crashes," *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), ACM Press, 1996, pp. 74–83.
6. E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," *Proc. 17th Conf. Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 2002, pp. 246–261.

Florin Sultan is a research staff member at NEC Laboratories America. His research interests include operating systems, networking, distributed systems, and system support for fault-tolerant and self-healing systems. Sultan received a PhD in computer science from Rutgers University. He is a member of Usenix and the ACM. Contact him at sultan@nec-labs.com.

Aniruddha Bohra is a PhD student in computer science at Rutgers University. His research interests include storage networks, operating systems, distributed systems, fault tolerance, and availability in Internet services. Bohra received an MS in computer science from Rutgers. He is a student member of Usenix, the IEEE Computer Society, and the ACM. Contact him at bohra@cs.rutgers.edu.

Stephen Smaldone is a PhD student in computer science at Rutgers University. His interests include operating systems, distributed systems, file systems, and storage networks. Smaldone received a BS in computer science from Rutgers. He is a student member of the ACM, the IEEE Computer Society, and Usenix. Contact him at smaldone@cs.rutgers.edu.

Yufei Pan is a software engineer at AskJeeves. His research interests include self-healing systems, operating systems, and networking. Pan received an MS in computer science from Rutgers University. Contact him at ypan@ask.com.

Pascal Gallard is a research engineer at IRISA/INRIA, Rennes, France. His research interests include operating systems, supercomputing clusters, and communication architectures. Gallard received a PhD in informatics from University of Rennes. Contact him at Pascal.Gallard@irisa.fr.

Iulian Neamtiu is a PhD student in computer science at the University of Maryland, College Park. His research interests include programming languages, operating systems, dynamic software upgrades, and software engineering. Neamtiu received a BEng. in computer science from the Technical University of Cluj-Napoca, Romania. He is a student member of the ACM. Contact him at neamtiu@cs.umd.edu.

Liviu Iftode is an associate professor in the Department of Computer Science and head of the Laboratory for Network Centric Computing (Discolab) at Rutgers University. His research interests include operating systems, distributed systems, pervasive computing, and mobile networks. Iftode received a PhD in computer science from Princeton University. He is vice chair of the IEEE Technical Committee on Operating Systems (TCOS), as well as a member of Usenix, the ACM, and a senior member of the IEEE. Contact him at iftode@cs.rutgers.edu.