Nonintrusive Remote Healing Using Backdoors*

Florin Sultan, Aniruddha Bohra
Department of Computer Science
Rutgers University, Piscataway, NJ 08854-8019
{sultan, bohra}@cs.rutgers.edu

Iulian Neamtiu, Liviu Iftode
Department of Computer Science
University of Maryland, College Park, MD 20742
{neamtiu, iftode}@cs.umd.edu

ABSTRACT

In this paper, we propose a remote healing approach for computer systems based on *backdoors*, a system architecture that supports monitoring and repair actions on a remote operating system or application memory image without using the processors of the target machine. A backdoor can be implemented using the remote memory communication technology provided by communication standards like Virtual Interface Architecture or InfiniBand, specifically its support for remote DMA read and write operations. We discuss the potential and challenges of backdoor-based remote healing and describe a preliminary prototype we developed as proof of concept.

1. INTRODUCTION

As computer systems become more complex, tolerance to failures and recoverability without compromising performance have emerged as guiding principles for system design [22]. The need for such features is exacerbated by the increasing demand for performance critical, highly available services.

Self-healing systems are becoming increasingly interesting because they present the aforementioned features. They have been studied in various contexts (software engineering, artificial intelligence, machine learning, etc.) [33]. Computer vendors have already launched initiatives and even market systems with built-in support for hardware and firmware fault detection and containment [16, 32]. From a system viewpoint, to support healing, a system must be capable of at least two functions: *(i)* monitoring, for detecting exceptional events (failure, intrusion, policy violations, etc.), and *(ii)* taking action in response to these events by recovery, repair, fault containment, etc.

Past and recent operating systems research has examined problems related to today's self-healing systems. On the monitoring side, OS-level monitors were used for run-time adaptation of an OS [26, 28]. Fault detection and containment was specifically addressed in extensible operating systems [27, 25], using clever forms of encapsulation to protect against faulty or malicious code. Despite obvious advantages like instant access to the entire system state, observing a system from within has limitations: *(i)* it has limited effectiveness, e.g., if the machine runs out of resources or certain components fail; *(ii)* cannot perform integrity checking on a system if the integrity checking code itself is bad or corrupted; *(iii)* cannot detect intrusion from within an already compromised system; *(iv)* cannot recover state from a system unless checkpointed externally to some stable device, with the incurred consistency and performance problems. Passive OS monitoring using virtual machine technology was proposed for intrusion detection and analysis, or automated failover support [4, 11]. However, its use is limited to specific problems and may incur high overhead and/or cost.

On the action side, the common approach is to force the crash/reboot of a system that develops a problem, without any attempt at saving its state. Some recent designs have introduced support into the OS for *replacing* an OS component dynamically [28]. However, this involves complete re-design of the OS, and, even with it, no system we know of is capable of repair *and* recovery actions after a failure.

Although the above approaches could be used to provide support for self-healing, they critically rely on resources of a system that may have already failed (e.g., a faulty processor, a deadlocked or crashed OS, etc.), or use simple destructive actions (e.g., reboot) to recover a machine, discarding useful state that could still be repaired and/or recovered. This is not particularly helpful, especially if the machine runs software critical to its users.

In this paper, we propose an alternative approach to self-healing called *remote healing*, where monitoring, repair and recovery actions are performed nonintrusively from a remote system. To support remote healing, a computer system must be equipped with a *backdoor* (BD), a specialized network interface that allows external access to its resources (memory, I/O devices, etc.) without involving its processor(s). A backdoor enables intervention on a system even when conventional wisdom would declare it "dead" (e.g., due to a system-wide freeze that does not allow any program or OS code to execute) or otherwise inaccessible (e.g., due to heavy load, DoS attack, etc.).

To perform remote healing, a BD-based architecture requires generic support from the OS. Basic building blocks of the system must be specifically provided with *channels* for remote access to system memory. Remote access channels allow such systems to provide rich functionality for remote healing that current systems lack and cannot implement without major redesign, or simply cannot support: low-overhead remote monitoring, extraction and recovery of OS and application state out of a failed system or application, on-line repairing or patching OS/applications, etc.

Backdoors can be implemented using *remote memory communication* (RMC), a technology originally developed to lower the overhead of communication by reducing OS involvement [2, 14]. In addition to low-overhead send/receive operations that require no OS intervention, RMC provides remote DMA (RDMA) primitives

*This work is supported in part by the National Science Foundation grant CCR 0133366.

that allow external access to the memory of a host without using its processor(s). RDMA read and write primitives are present in industrial standards like Virtual Interface Architecture (VIA) [12] and InfiniBand [17]. Until now, RMC has been used mainly as a high-performance cluster interconnect [24, 6, 34, 9, 18, 23], and to provide efficient support for fault tolerance [35, 30].

This paper takes a novel approach on the use of RMC in system design. We argue that the nonintrusive communication capability of RMC is a viable and powerful building block for remote healing systems (RHS). To our best knowledge, this is the first attempt at leveraging RDMA capabilities of modern RMC for nonintrusive remote healing. Previous work done in the 1980's on the Titan system [21] has used custom hardware with limited RMC capabilities (memory controllers equipped with Ethernet interfaces) to perform remote read/write memory operations for remote debugging and software/data patching without rebooting the kernel [20].

Intrusive remote memory access to a computer system has been used before, e.g., for remote debugging of system software. In this case, the access is usually performed over a serial interface, the debugged system is passive and, in the case of a crash, the best another system can do is a post-mortem inspection and analysis of the memory of the crashed system. The technique uses processor cycles on the inspected machine and specifically requires that critical hardware components of the possibly crashed machine are in good shape. In contrast, BD-based remote healing allows repair or recovery actions to be performed even when the processors of the target system are not available.

The remainder of the paper is structured as follows. Section 2 discusses the design principles of a BD architecture. Section 3 provides background on RMC. Section 4 describes a BD implementation using RMC. Section 5 discusses BD design challenges. Section 6 describes a prototype system we have built using the BD idea. Section 7 reviews related work.

2. BACKDOOR PRINCIPLES

In order to support remote healing a BD-based architecture must satisfy the following design principles:

Bidirectional access. BD must allow both input (read) and output (write) operations on a remote target system. Read operations are used mainly for monitoring while write operations are necessary to perform healing actions.

Remote memory access. BD must support at least remote memory access operations. Additional operations such as atomic remote memory access and remote I/O are desirable in order to support more complex healing actions.

Availability. BD must be available even after a failure of the target system. A failed OS or hardware component that is not used in implementing the BD should not impair its continued operation.

Nonintrusiveness. BD must enable remote access to resources of a computer (memory, I/O controllers, etc.) without involving processors of the target system. The processors/OS may however be involved in BD initialization operations.

Transparency. Remote access through a BD must not be visible/detectable on the target system by direct means, e.g., by accessing an I/O port. Indirect detection might be possible through fine-grained and accurate system bus performance measurements.

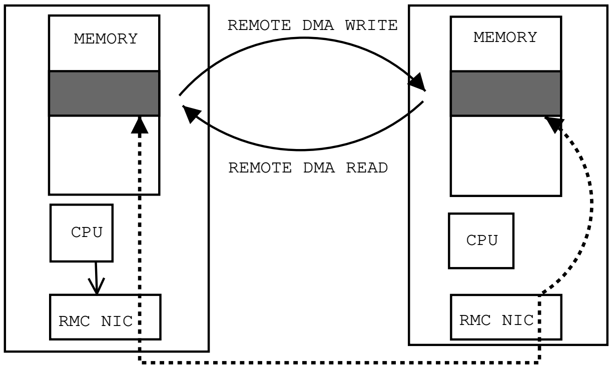

Figure 1: ***Remote DMA read/write operations in RMC.***

Access control. BD must provide a mechanism to allow the monitor and target operating systems to agree on access permissions when monitoring begins. However, the target should not be able to close the BD or change access permissions afterwards.

Tamper resistance. The target system must not be able to change the result of a remote access performed through the BD. The content made visible externally must not be different from the one transferred from/to the memory or an I/O device.

3. REMOTE MEMORY COMMUNICATION

Remote memory communication (RMC) is a communication technique that aims to significantly reduce the communication overhead typically associated with TCP/IP networking. The basic idea in RMC is to bypass the operating system in the common send/receive path while providing a protected channel for communication. The RMC architecture supports two communication models: send/receive and remote DMA (RDMA).

In the send/receive model, the communicating parties can bypass the OS and use the network interface controller (NIC) directly for communication. However, the host processor is still involved in such transfers. In contrast, an RDMA operation completely bypasses the processor on the remote host. For this, the remote NIC performs a silent DMA to/from the host memory.

RDMA write is the most common RMC operation, supported by most RMC implementations [14, 2, 13, 19]. With RDMA write, the sender can write into a remote memory buffer without remote processor intervention. The completion of the RDMA write can be determined by checking a completion queue in the network interface or through an application specific flag in the area to be updated. RDMA read is a more complex operation which, until recently, has not been supported in RMC implementations, although part of the standards [12, 17]. With RDMA read, a node can initiate a transfer from a specified remote memory buffer without involving the remote OS or processor.

Figure 1 illustrates the RDMA read/write operations between two machines. For initialization, both endpoints execute a one-time register operation notifying their NICs of the memory buffers involved in RDMA. The processor of the left node initiates the RDMA operation (vertical solid arrow). The solid horizontal

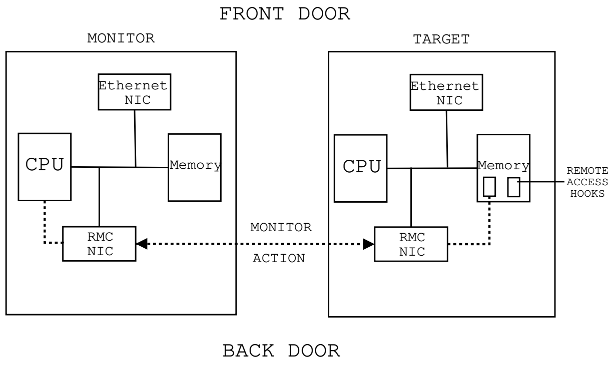

Figure 2: ***An RMC-based backdoor.***

arrows show the logical data path of the RDMA operations, while the dotted line follows the physical data path. An RDMA write silently transfers the contents of the source into the destination buffer. To achieve this, the NIC bypasses the processor at the destination and does a DMA of data it receives from the network into the registered destination buffer. In an RDMA read operation, the remote NIC bypasses the host processor to directly perform a DMA operation from the remote memory.

4. BACKDOOR WITH RMC

In this section, we describe a backdoor implementation using remote memory communication (Figure 2). RMC is a good candidate to implement a backdoor because current RMC implementations satisfy most of the BD principles outlined in Section 2.

4.1 Monitoring Using RMC

To provide support for monitoring, a system must be instrumented to enable detection of anomalous state in the operating system or applications running on it. Instrumentation state may include liveness or performance statistics, integrity invariants, etc. We next discuss monitoring alternatives and how RMC can improve on them.

Local Monitoring. Introspection and local monitoring have been used in extensible operating systems and in hot-swapping operating systems to adapt to varying load/resource profiles [26, 28]. Local monitoring has the advantage of creating/using perfect and instant knowledge about a system, since it can directly access data structures in system memory.

Local monitoring has two major drawbacks. First, it is intrusive to the monitored system by competing for its resources (processors, memory, disk). Second, it is as good as the monitored system: in the event of OS/hardware failure, overload, or attack, a local monitor may not be able to execute and the system may lose all its monitored state. In a distributed system, local monitoring is not capable of accurately detecting such events.

Remote Monitoring. Monitoring can be performed remotely by a different system using communication over a network. This can be done either in *push mode*, where the monitored node sends information (heartbeats) to its monitoring node, or in a *pull mode*, where the monitoring node periodically requests information from the monitored node. The monitored node cooperates by generating state data through introspection and externalizing it for remote access. Remote monitoring has two major benefits. First, assuming a reliable network, it can detect catastrophic failures due to hardware or software faults. Second, it enables system-wide decisions in a distributed system, for reconfiguration or resource provisioning.

Remote monitoring has several drawbacks. First, it suffers from imperfect knowledge: network unreliability, resource contention in the network and processor contention for executing protocol software at the monitored node may cause inaccurate monitoring. Second, it incurs overhead at both monitored/monitor nodes. Third, it is limited in scope, since a remote monitor does not have direct access to the state of the monitored node.

Best of Both Worlds: Nonintrusive Remote Monitoring. A BD-based architecture can use remote access channels provided by RMC to perform nonintrusive remote monitoring. Figure 2 shows a monitor node accessing the memory of a target (monitored) node through RMC.

Besides leveraging the benefits of remote monitoring, RMC alleviates or eliminates its penalties. Monitoring becomes nonintrusive to the target system, to the extent that this cannot directly detect that it is being monitored (according to the transparency principle). Furthermore, the high performance and reliability typical of RMC hardware help with accurate monitoring. Finally, subject to BD access control policies, RMC could enable access to the entire memory of a system.

There are two clear benefits of monitoring via nonintrusive remote access channels: *(i)* In practice, it does not affect the performance of the target system as it does not compete for its processor(s). The only contended resource is the system bus. However, in order for this to impact performance, a memory-intensive workload on the target system must compete with an unrealistically aggressive monitoring (e.g., generated by continuous memory dumps via RMC). *(ii)* It allows retrieval of state still present in system memory after a failure, if the RMC interface and the memory are available (according to the availability principle). The entire physical memory of the system is potentially accessible, even after some other critical components of the system have failed.

4.2 Action Using RMC

Upon detecting anomalies in the monitored node, a healing action must be executed to bring it to a desired configuration or state. The action component is semantically rich (depending on the trigger event) and may affect the system at the OS or application level.

Changing a system from *within* has been the focus of research [26] on automatic adaptation in extensible kernels. However, this may not always work due to resource problems (exhaustion, critical failures, etc.), or simply because the integrity of the module performing the action has been compromised. Some of these problems have found solutions in the context of virtual machine monitors (VMM). VMMs were used to build backup state on different nodes for recovery after a crash [4], and to provide a secure execution environment for system loggers to help with intrusion detection and analysis [11]. Theoretically, these approaches may be used in RHS but in practice they are too expensive and/or not general enough. Recently proposed designs like [28] provide built-in support for complex *online* reconfiguration, where whole OS subsystems can be hot-swapped. Hot-swapping may be ineffective on a heavily loaded system, or impossible on a dead system that may not have cycles available to execute even the hot-swapping code.

In contrast with the above approaches, a BD-based remote healing architecture can exploit RMC to implement remote actions of fine-grained control at the OS or application level: *(i)* repair data or code on a live system; *(ii)* extract data from a dead system, to be recovered on another healthy system. Moreover, an RHS can perform these actions even after other conventional access paths have failed due to hardware of software faults.

To support remote healing, the OS must provide *remote access hooks* (shown in the target node in Figure 2). Their role is to provide an interface for enforcing actions on the OS or the applications running on it. The remote access hooks must be registered with the RMC for remote access (read or write) by another system that runs recovery or repair code.

The actual specification of a hook depends on the domain of its intended action. For example, a recovery hook can be an indirection data structure for retrieving references to critical data structures within the target OS, for easy remote extraction or modification. Such hooks may use OS abstractions specifically designed for external manipulation of state by a specialized extraction protocol.

Repair hooks may control remote accesses to certain regions of OS memory and can be used by specialized repair threads executing on the monitor node. Through the repair hooks, the repair threads can build a local view of the target system, analyze/diagnose the problem, and fix damaged state through remote writes. Similarly, application-defined repair hooks can enable remote direct writes into an application's memory to patch or clean-up the state of a faulty or corrupted application.

While capable of performing nonintrusive repairs through remote memory access, the BD can also allow a remote node to opportunistically use the processors of the target machine, if available, to execute healing actions. In this case, the repair hooks can be regions of memory where specialized rescue handlers may reside. Handlers may be replaced remotely if needed, without interfering with operation of the target system, and the system forced into executing them. A rescue handler may implement complex actions, e.g., testing/diagnosis of devices and drivers.

5. CHALLENGES

In this section, we discuss several challenges and open questions that confront the proposed BD architecture for remote healing.

5.1 Monitoring Issues

The main problems faced by monitoring a system with BD are: *(i)* semantics of externalized state, *(ii)* synchronization-free access, *(iii)* fault propagation, and *(iv)* erroneous detection.

Semantics of externalized state. Detecting that a monitored system is in a bad state by using only its externalized state can be difficult due to complex state semantics. "Bad state" may range from application-specific (e.g., deadlock in a multithreaded program) to system-specific or system-wide state (e.g., a faulty hardware component). A monitor must interpret it appropriately to detect a failure.

For example, application specific monitoring requires an understanding of application semantics, such as application specific invariants. Consider an application that attempts to lock a mutex it already holds. Accurate detection of this error requires knowledge of the expected behavior (e.g., cannot lock a mutex again in the same thread), and cooperation on behalf of the application (e.g., providing information about locks currently held and the next locking action). The code that generates such application invariants (at the target) and the code that verifies them (at the monitor) can be either written by the programmer or automatically generated by compiling a set of constraints expressed in a specialized language [10].

Correct interpretation of monitored state also depends on its correctness. For example, an OS crash or an attack may corrupt monitored state and potentially obscure the fault to a monitor, thereby compromising failure detection. System support is needed for protection against corruption of critical monitored state and/or to detect corruption after its occurrence.

Synchronization-free access. Remote monitoring of a live system may run concurrently with updates performed on monitored data structures. Modification of a data structure while a monitoring access is in progress may result in inconsistent information for a monitor. Mutual exclusion (e.g., using locks) cannot be used to solve the problem, as it violates the BD nonintrusiveness principle. Such complications require careful definition of monitored state and its access protocols.

Fault propagation. For monitoring to be meaningful, the faulty component of the monitored system must be accurately identified. However, this is not always possible, as the fault may be propagated to other subsystems that have dependencies to the faulty one, to other processes in an IPC chain with a faulty process, etc. For example, a disk error may manifest itself as memory corruption in an application, making it impossible for a monitor to identify the fault.

Erroneous detection. In some situations, the monitor may not be able to differentiate between normal behavior of the system and an error condition. For example, when monitoring the "liveness" of a server application, idleness when there are no connected clients may be erroneously interpreted as application failure. A conservative monitor may ignore significant errors while an aggressive monitor may cause false positives. It is a challenging task to carefully engineer a system and program the monitoring in order to minimize such errors.

5.2 Action Issues

The main issues for performing healing actions via BD are: *(i)* intrusive actions, *(ii)* correctness of repair actions, *(iii)* correctness of recovered state, and *(iv)* control of BD actions.

Intrusive actions. While monitoring should be nonintrusive, remote healing actions may *necessarily* be intrusive for a "good" purpose. For example, to retrieve consistent state for recovery, a recovery hook may have to stop an application; or, to repair an OS under DoS attack, a repair hook may have to kill an offending process. The challenge is how to implement such necessary intrusive actions on a live system.

Correctness of repair actions. Injecting incorrect state into a system may cause irreparable damage and defeat the purpose of remote healing. Definition and enforcement of correct repair actions on a target system should be automated using a language-based framework (similar to that of [10]), and by providing OS abstractions to implement the repair actions.

Correctness of recovered state. A recovery action after a fault may incorrectly use data potentially corrupted during the fault. For example, an OS fault may generate wild writes that damage in-memory state used for recovery. A system could include support for preventing such integrity violations (e.g., protection mechanisms) or for detecting them (e.g., by maintaining checksums over critical data, to allow remote recovery procedures to identify and filter out invalid state). Despite such precautions, it might be possible that only partially correct state can be recovered from a failed system. In such cases, a recovery procedure may at best restore the system to a state consistent enough to permit continued operation.

Control of BD actions. Protection of a remote access channel is important both during monitoring and while performing healing actions over it.

At the monitor end, the monitor process/thread should be the only trusted entity allowed to access the channel. Protecting against unauthorized write access (e.g., for repair actions) to the memory of the remote system requires strict and careful registration of each remote access hook. This may reduce flexibility in actions. The alternative is to fully trust a monitor node and register the whole memory.

At the monitored end, a faulty system may, intentionally or not, try to "close" the backdoor, thus making monitoring and/or healing impossible. Closing the backdoor may amount to just brute-force deregistration of the local endpoint of a remote access channel, or may take subtler forms (e.g., substituting the legitimate channel by a fake one that gives a monitor the appearance of a healthy system). Such attacks can be prevented if the BD provides simple support for disabling accesses after its (safe) initialization. This feature is not currently supported by existing RMC network interface controllers.

6. CASE STUDY

We have implemented a prototype RHS for Internet services by modifying the FreeBSD kernel and using VIA Giganet cLAN [13] to implement the BD[1]. The system runs equivalent servers on multiple machines that use VIA communication to perform monitoring, detect failures at three *failure levels* (OS, application, client session), and salvage affected sessions.

In our system, a "failure" is defined as the impossibility of continuing service to one client, to all clients serviced by a given server application, or to all clients serviced by the node. Under this definition, a failure may occur at one of the above three levels, may reflect various degrees of "illness" and may have multiple causes, including but not limited to hardware faults: *(i)* a faulty hardware component, e.g., processor, network interface, local hard disk, interrupt controller, etc. *(ii)* a faulty software component in the OS that leads to a system-wide freeze, e.g., system hanging due to a locking error, a misplaced panic, etc.; *(iii)* a faulty software component in the application, e.g., a deadlock that prevents the application from servicing all or part of its clients; *(iv)* a wrong operator command or a misconfiguration that causes the node/OS/application to halt or stop making progress while servicing clients. The memory contents and the RMC hardware are assumed available after a failure.

The system currently implements a single action in response to a failure – recovery of affected sessions. Recovery is achieved by dynamically migrating individual sessions out of their current server to another healthy server. Session migration is transparent to client, and does not affect the consistency and correctness of the service. In addition, depending on the failure level, the system may take other steps, e.g., reconfigure/restart a faulty application in case of session/application failure, reboot the faulty server system, etc.

The monitoring component of the system is responsible for failure detection. It is implemented by *monitor processes* using an OS abstraction of a target node called *Progress Box* (PB). A PB provides monitored entities and their monitor with a basic meter for liveness called *progress counter*, a monotonically increasing scalar value that has an associated failure detection deadline. The PB is accessed through an API from both the target and the monitor node. To assert its liveness, an entity running on the target node uses the PB API to allocate a progress counter in the PB, then performs introspection and voluntary reporting by updating its value. The monitor process reads the PB using RDMA read operations. If it detects that a progress counter has not been updated within its detection deadline, the monitor considers that the corresponding entity has failed and decides what recovery action to take.

The recovery component of our prototype provides fine-grained recovery support for client service sessions after a failure is detected. The recovery solution consists in migrating the active client sessions to another server using Migratory-TCP [31]. The session state is embedded as a service continuation (SC) [29], an OS mechanism that we have implemented to encapsulate OS and application state associated with a client session. Upon failure detection, the new server reads the SC-encapsulated state from the failed server and reinstates the session state.

7. RELATED WORK

Recent initiatives in industry [16] and academia [22] seek to shift the thrust of future computing systems from performance to self-managed, self-configuring, self-healing systems.

Middleware solutions have been proposed for software rejuvenation [15, 5], self-healing [3], and problem determination in complex systems [7]. These systems gather statistics at run-time through monitoring, detect anomalies using these statistics, restart or replace software components to bring a system back to a consistent/working state. In contrast, BD focuses on operating system support for monitoring a computer system, including application and system software, healing it and/or retrieving useful state after failure. Our approach addresses a missing link: we reuse useful state in the system rather than lose it by reboot or restart.

Self-adapting operating systems have been studied in the context of extensible OS research [25] where behavior is modified by extending OS functionality with user code downloaded into the OS kernel. More recently, [28] proposed an OS that allows hot-swapping components at run time. However, these systems are built from scratch to provide an extension interface, or an object-oriented design to allow easy replacement of hot swappable components. Both systems monitor their state locally and adapt to current load conditions. Neither can perform recovery after an OS or a hardware component failure. In contrast, we describe a solution using a slightly modified OS using remote monitoring where we can replace components, repair state, as well as extract useful state from a system *after* failure.

[1]For lack of RDMA read support in our hardware, we have emulated it through RDMA writes executed by a dedicated processor on an SMP machine.

System/application state preserved in nonvolatile memory has been used in [1, 8] to survive crashes. In [1], a stable region of memory called Recovery Box is used to store system state and retrieve it after crash. If the state is corrupted, the system falls back to recovery by hard reboot. In [8], a reliable file cache protects file system data during a crash and allows to warm reboot the file system from it. Both systems focus on protection against unauthorized accesses during a crash through controlled interfaces, integrity checks and hardware support. BD may also rely on OS or application state being available in system memory after a failure. However, we propose a uniform architecture for controlled state recovery and for repair of damaged state.

Language support for automatic error detection and repair of data structures is explored in [10]. A BD-based architecture can be integrated with such support to define detection/repair algorithms on system and application data structures. It can also circumvent local failures by being capable to detect errors and perform repairs remotely.

8. REFERENCES

[1] 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.

[2] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[3] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, Self-Awareness and Self-Healing in OpenORB. In *Proc. 1st Workshop on Self-healing Systems*, Nov. 2002.

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

[5] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *Proc. HotOS-VIII*, May 2001.

[6] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-based Network Servers. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 2001.

[7] M. Chen, E. Kiciman, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proc. DSN 2002*, June 2002.

[8] 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. 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1996.

[9] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proc. 2nd USENIX FAST Conference*, Mar. 2003.

[10] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proc 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[12] D. Dunning et al. The Virtual Interface Architecture. *IEEE Micro*, 18(2), Mar. 1998.

[13] Emulex, Inc. http://www.emulex.com.

[14] E. Felten, R. Alpert, A. Bilas, M. Blumrich, D. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proc. 23rd Annual Symposium on Computer Architecture (ISCA)*, May 1996.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation : Analysis, Module and Applications. In *Proc. 25th IEEE Intl. Symposium on Fault Tolerant Computing (FTCS)*, June 1995.

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

[17] The Infiniband Trade Association. http://www.infinibandta.org, Aug. 2000.

[18] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, Apr. 2003.

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

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

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

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

[23] M. Rangarajan, S. Gopalakrishnan, A. Arumugam, R. Sarker, and L. Iftode. Federated DAFS: Scalable Cluster-based Direct Access File Servers. In *Proc. 2nd Workshop on Novel Uses of System Area Networks (SAN-2)*, Feb. 2003.

[24] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. In *Proc. of The Annual Linux Showcase, Extreme Linux Workshop*, Oct. 2000.

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

[26] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In *Proc. HotOS-VI*, May 1997.

[27] E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

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

[29] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. Technical Report DCS-TR-508, Rutgers University, Nov. 2002.

[30] F. Sultan, T. D. Nguyen, and L. Iftode. Scalable Fault-Tolerant Distributed Shared Memory. In *Proc. of Supercomputing Conference*, Nov. 2000.

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

[32] Unisys ES7000 server. http://www.unisys.com/hw/servers/es7000.

[33] ACM SIGSOFT Workshop on Self-Healing Systems., Nov. 2002. http://www-2.cs.cmu.edu/~garlan/woss02/.

[34] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI Communication for Database Storage. In *Proc. 29th Intl. Symposium on Computer Architecture (ISCA)*, May 2002.

[35] Y. Zhou, P. M. Chen, and K. Li. Fast Cluster Failover using Virtual Memory-mapped Communication. In *Proc. 13th Intl. Conference on Supercomputing*, June 1999.