Using Remote Memory Communication for Self-Healing Systems*

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

Iulian Neamtiu, and 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 novel self-healing approach for a computer system based on Back-Doors (BD), a system architecture that allows monitoring and repair actions to be performed on a remote operating system or application image without using remote CPU cycles. The key ingredient of the Back-Door architecture is the Remote Memory Communication (RMC) technology provided by standards like Virtual Interface Architecture (VIA) or InfiniBand, specifically its support for remote DMA read and write operations. In this paper, we discuss the potential and challenges of the Back-Door approach to self-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 [19]. The need for such features is exacerbated by the increasing demand for performance critical, highly available services.

Self-healing systems (SHS) are becoming increasingly interesting because they present the aforementioned features. They have been studied in various contexts (software engineering, artificial intelligence, machine learning, etc.) [18]. Computer vendors have already launched initiatives and even market systems with built-in support for hardware and firmware fault detection and containment [14, 26]. From a system viewpoint, an SHS 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 exceptional events by recovery, fault containment, repair, etc.

Past and recent operating systems research has examined problems related to today's SHS. On the monitoring side, OS-level monitors were used for run-time adaptation of an OS [22, 25]. Fault detection and containment was specifically addressed in extensible kernels [23, 21], using clever forms of encapsulation to protect against faulty or malicious code. Despite obvious advantages like instant access to all the state of the system, observing a system from within has limitations: *(i)* 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 guarantee intrusion detection from within an already compromised system; *(iv)* cannot recover state from a system, unless checkpointing state 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, 9]. However, its use is limited to specific problems and may incur high overhead and/or cost. Remote monitoring and control through standardized protocols is routinely used in the Internet management infrastructure [12]. However, the control interface has a coarse granularity and is limited to on-off actions on full subsystems or systems, or to configuration changes.

On the action side, the generic 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 [25]. 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 system support for an SHS, they critically rely on resources of a system that may have already failed (e.g., a faulty CPU, 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 to equip systems with *back-doors* (BD) that allow access to a system's memory without requiring CPU cycles on the target machine. A BD architecture would allow intervention on a system and to perform operations on it even when conventional wisdom would declare the system "dead" (e.g., due to a system-wide freeze that does not allow any program or OS code to execute). Such a mechanism would provide a powerful tool in designing systems for automated monitoring, healing, recovery and repair.

Fortunately, the tool that makes this idea possible already exists. *Remote memory communication* (RMC) is a technology originally developed to lower the overhead of communication by reducing OS involvement [2, 11]. In addition to low-overhead send/receive operations that require no OS intervention, RMC provides remote

*This work is supported in part by the National Science Foundation under the NSF CAREER CCR 0133366.

DMA (RDMA) primitives that allow external access to the memory of a host, without using its CPU. RDMA read and write primitives are present in industrial RMC standards like VIA [10], InfiniBand [15], and have been used as building blocks in other standards like DAFS [16].

RMC has been used previously in system design as a fast and reliable communication channel decoupling (sub)system functionality. For example, TCP Servers [20] offloads TCP processing in a cluster over RMC channels. Infiniband [15] uses RMC to decouple device from host processing. Intra-cluster or client-server communication exploits the low overhead and the good performance typical of current RMC hardware [6, 27, 17]. In one instance [28], RMC has been used for providing application-level fault tolerance.

This paper takes a novel approach on the use of RMC in system design in the context of SHS. We argue that the silent communication capability of RMC is a viable and powerful building block for self-healing systems. To our best knowledge, this is the first attempt at leveraging RMC capabilities in this domain.

An analogy for our proposed use of RMC is with a rudimentary remote memory access method that is commonly used, 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. This technique uses CPU cycles on the inspected machine and specifically requires that critical hardware components of the possibly crashed machine are in good shape. In contrast, one of the directions that our RMC-based design exploits goes beyond static passive analysis. Silent access to memory allows, for instance, extraction of useful state from a dead system (even when "dead" may mean loss of CPU or other critical hardware component) and even changing the behavior of the system by online modifications to its memory.

To be functional, the BD design 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 self-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 system/applications, etc.

The remainder of the paper is structured as follows. Section 2 gives some background on RMC. We present the characteristics of an SHS in Section 3. We discuss design challenges in Section 4. We describe a case study for a SHS in Section 5. Section 6 reviews related work.

2. 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 CPU is still involved in such transfers. In contrast, a remote DMA operation completely bypasses the CPU on the remote host. For this, the remote NIC performs a silent DMA to/from the host memory.

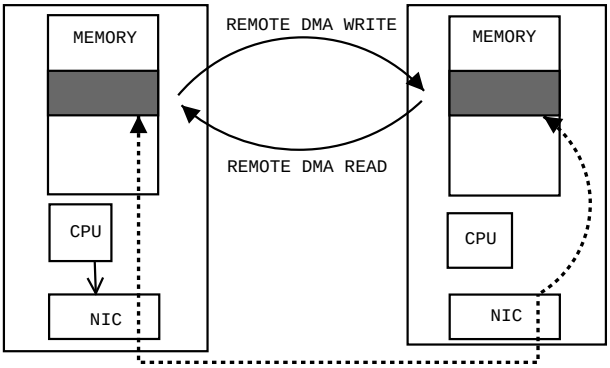

Figure 1: ***Remote DMA read/write with RMC.***

RDMA write is the most common RMC operation, practically supported by all RMC implementations [11, 10, 15]. With RDMA write, the sender can write into a remote memory buffer without remote CPU 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 standard [10, 15]. With RDMA read, a node can initiate a transfer from a specified remote memory buffer without involving the remote OS or CPU. The data read from the remote memory is stored in a local buffer.

Figure 1 illustrates the RDMA read/write operations between two machines. For initialization, both communicating parties execute a one-time register operation notifying their NICs of the memory buffers involved in RDMA. The CPU of left-side node initiates the RDMA operation (vertical solid arrow). The solid horizontal 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 CPU 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 CPU to directly perform a DMA operation from the remote memory.

3. SELF-HEALING WITH RMC

Features of self-healing systems have been identified and studied in various contexts [18]. In this paper, we describe the functions and architectural features that a system (OS) equipped with Back-Doors (Figure 2) must provide in order to support self-healing software. First, an SHS must be able to detect anomalies in its behavior using the available state information. Second, on detection of a problem, the system must choose an appropriate action for recovery towards a "correct state" and enact the action. We denote these two major functions that the system support must perform in an SHS as *monitoring* and *action*. For an OS designer, monitoring maps into observing the state of the OS and applications running on it. Depending on the observed anomalies, action may

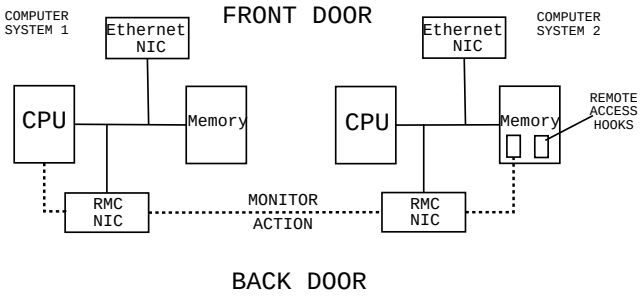

Figure 2: ***Monitoring and action with RMC.***

map into recovery of good system state, repair of damaged state, containment of a fault, repelling intrusion, etc.

3.1 Monitoring using RMC

To provide support for monitoring, an SHS must be instrumented by identifying statistics of interest for a healing action, and collecting those statistics during execution. For example, the number of context switches over time may provide information about system liveness, the number of lingering half-open connections in the TCP/IP stack may hint at a SYN flood attack, etc. We next discuss the alternatives available to a system for monitoring, focusing on their relative benefits and drawbacks.

Local Monitoring

Introspection and adaptation have been used in extensible operating systems [21] as well as hot-swapping operating systems [25]. Such systems collect statistics and use them to adapt to load and resource profiles. Local monitoring has advantages like creating/using perfect knowledge about the system and allowing for easy adaptation. Monitoring a system locally does not rely on message exchange over a network and therefore has instantaneous and perfect knowledge of the data extracted from the monitored system. In addition, a monitor that executes locally has access to the system memory, data structures and other configuration parameters that are not accessible remotely by traditional means. This allows systems to better adapt themselves to the current load profile.

Local monitoring also has two major drawbacks. First, the monitor running locally competes with other tasks running on the system for resources like CPU, memory and disk. On a heavily loaded system, the monitoring overhead might be too high. Ideally, monitoring should be transparent and consume no resources on the monitored system.

Second, local monitoring is useful as long as the system does not suffer a crash or a hardware failure, or is not compromised by an attacker. If a catastrophic failure occurs, the local monitor itself cannot execute and the system loses all its (potentially recoverable and useful) state data. Similarly, a compromised system cannot be trusted with correctly monitoring itself. Moreover, such events are of utmost importance in a distributed system, and must be reported.

Remote Monitoring

Remote monitoring allows a system to be monitored by an external system through an exchange of messages over a network. This can be done either in *push mode*, where the monitored node sends information over the network periodically (heartbeat) or on an event, or in a *pull mode*, where the monitoring node periodically requests information from the monitored node. The monitored node cooperates by *(i)* generating state data through introspection, and *(ii)* externalizing the state data for remote access.

Remote monitoring allows detection of fatal faults in a monitored node through simple non-receipt of heartbeats or responses to a query. Remote monitoring has two major benefits: it can detect catastrophic failures due to hardware or software faults, and it allows for global decisions. In a distributed system, remote monitoring enables complex system-wide decisions for reconfiguration or resource provisioning.

Unfortunately, this approach also has some drawbacks. First, it may suffer from imperfect knowledge: since the monitor communicates with the monitored node over a network, network delays and unreliability may cause unpredictable monitoring. Note that, from a system viewpoint, both link contention in the physical network and CPU contention for executing protocol software at the monitored node contribute to unpredictable monitoring. Second, remote monitoring incurs overhead at both the monitoring node and the monitored node, therefore it is more expensive than local monitoring. Third, it is restricted by the inherently limited access, since a remote monitor does not have direct access to the memory of the monitored node.

Best of Both Worlds: Remote Monitoring over RMC

A Back-Door architecture (Figure 2) uses remote access channels provided by RMC to perform remote monitoring. A monitoring node (the left-side node in Figure 2) can silently access the memory of the monitored node through such a remote access channel.

Besides leveraging the benefits of remote monitoring outlined above, RMC alleviates or eliminates its penalties. As monitoring accesses are performed without the intervention of the CPU/OS of the monitored node, they do not impose extra burden on the node, to the extent that the node may even be unaware that is being monitored. Furthermore, the high performance and reliability typical of RMC hardware help with accurate monitoring. Finally, at least in principle, RMC allows remote access to the whole memory of a system.

There are two clear benefits of monitoring via silent remote access channels: *(i)* It does not affect the performance of the monitored system under normal operation as it does not compete for CPU or memory resources. The only contended resource is the system bus, for which modern processors provide techniques to mask the latency of contention. *(ii)* It allows retrieval of state still present in system memory after a failure, if the RMC interface and the memory are available. The entire physical memory of the system is potentially accessible, even after some other critical components of the system have failed.

3.2 Action using RMC

An SHS must execute a certain action towards a desired configuration or state when the monitoring component detects anomalies. The action component is semantically rich (depending on the trigger event) and may affect the system at the OS or application level. In this section, we argue for externally performing actions on a target system, as local actions may be ineffective or even impossible to achieve.

Changing a system from the *inside* has been the focus of research [21] on automatic adaptation in extensible kernels. However, altering a system from within may not always work due to resource problems (exhaustion, critical failures, etc.), or simply because the module performing the action has been corrupted or its integrity compromised. Some of these problems have found solutions in the context of virtual machine monitors (VMM). The VMM-based approaches passively observe a system as a whole while it runs as a "guest" (OS + applications) inside a VMM, by logging interactions of the guest OS with the 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 [9]. These approaches may be used in SHS architectures but they are either too expensive or not general enough. Other recently proposed designs like [25] provide built-in support for complex *online* reconfiguration, where whole OS subsystems can be hot-swapped. The disadvantage is that they must be built from scratch with a modular, object-oriented design so that replacing live OS components does not affect active requests. Also, 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 previous approaches, the Back-Door architecture exploits RMC to implement actions of fine-grained control at the OS or application level: *(i)* inject data or code in a live system; *(ii)* extract data from a dead system, to be recovered on another healthy system. Moreover, the Back-Door can perform these actions even after other conventional access paths have failed due to hardware of software faults.

To support self-healing, an OS with Back-Door must provide *remote access hooks* (shown in the right-side node in Figure 2). Their role is to provide an interface for enforcing actions on a system or the OS itself. The remote access hooks must be registered with the RMC for remote access (read or write) by another system that runs repair or recovery code.

The actual specification of a hook depends on the domain of its intended action. We point to several possible domains, while not describing here implementation solutions. For instance, a repair hook can be a region of memory where specialized salvage handlers may reside. Handlers may be injected/replaced remotely, without interference with the system's execution, and the system forced into executing them. If a system hangs in an infinite loop with interrupts disabled, remote code re-writing [24] of a jump instruction using the repair hook may take it out of the loop to a rescue handler. Similarly, specialized application-aware repair hooks can enable remote direct writing into an application's memory. Such hooks can be used remotely to "patch" the state of a faulty or corrupted application.

An example of a recovery hook is specialized code (built-in or dynamically injected) for retrieving references to critical data structures within the OS, for easy remote extraction or modification. Such hooks may even implement OS abstractions specifically designed for external manipulation of state by a specialized extraction protocol.

4. CHALLENGES AND LIMITATIONS

In this section we discuss several challenges and open questions that confront the proposed Back-Door mechanism of self-healing. Healing an operational system poses problems specific to the *monitoring* and the *action* components of the architecture.

4.1 Monitoring Issues

The three main problems faced by monitoring a system using Back-Doors are: *(i)* understanding the semantics of the externalized state, *(ii)* fault propagation, and *(iii)* erroneous detection.

Semantics of externalized state

Detecting that a monitored operational system is in a bad state by using the state externalized by the system is difficult. The 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).

Application specific monitoring requires an understanding of application semantics, application specific invariants, and a cooperative application that provides such information to the remote monitor. For example, consider an application that attempts to lock a mutex it already holds. Detecting where the error is in order to take corrective action is impossible without knowledge of the expected behavior (e.g., cannot lock a mutex again in the same thread), and the cooperation of the application (e.g., providing information about locks currently held and the next locking action).

Similarly, to identify a system-wide failure, e.g., a hardware component failure, requires the monitored OS to externalize a "liveness" state for that component. This state can be a monotonically increasing counter or a simple flag which is set on failure. The monitor must interpret it appropriately and detect the failure.

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 a server application, idleness when there are no clients connected may be erroneously interpreted as no progress. 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 to minimize such errors.

4.2 Action Issues

A self-healing action may involve retrieving state from a crashed system, or repairing state through the Back-Door. The two main issues for performing self-healing actions via Back-Doors are: *(i)* correctness of the healing action (including correctness of the mechanism and integrity of the new state injected in the system), and *(ii)* control of the Back-Door actions.

Correctness

Injecting incorrect state in a system may cause irreparable damage to the system and defeat the purpose of self healing. This is why if the action involves replacing or introducing code into a system, the new code must be checked for correctness. For example, it must not access resources not present on the monitored system, must not

make illegal references to memory, and, most importantly, must preserve the role of the replaced component in the system.

Control of Back-Door Actions
Protection of a remote access channel is important both during monitoring and while performing self-healing actions over it. With RMC, a channel provides an intruder with opportunities to attack at both ends.

At the monitor end, the monitor should be the only trusted entity and the only entity allowed to access the back-door channel. Protecting against unauthorized write access (e.g., for repair/healing 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 the monitor and register the whole memory.

A more insidious attack may occur at the monitored end, where an attacker may try to "close" the back-door in order to hide its activities from a remote monitor. Closing the back-door 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 may be prevented if the NIC provides simple support for disabling accesses after its (safe) initialization.

Finally, availability of the back-door channel even after a system crash in order to retrieve useful state requires the NIC to be programmed to leave the back-door open in such events.

5. CASE STUDY

We have applied the ideas and guidelines outlined here to develop a prototype SHS that enables self-healing and fine-grained recovery in a server cluster of nodes with similar functionality. The system detects failures at three *failure levels* (OS, application, and client session), and salvages affected sessions.

In our system, a "failure" in a cluster node 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., CPU, 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.

In case of a failure, the *recovery action* is to save affected sessions by dynamically migrating them out of their current server node to another healthy node in the cluster. 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 node, reconfigure the cluster, etc.

We have designed and implemented in our system the OS support needed for remote monitoring of a node and for session recovery.

The monitoring component of the system is responsible for failure detection. It is implemented by *monitor processes* that run on the same cluster the service runs on. Monitor nodes (denoted by *M*) run a failure detection algorithm based on observation of state dynamics in monitored nodes (denoted by *m*).

Monitored entities running on *m* (OS or applications) communicate indirectly with their monitor(s) *M* via a per-node OS abstraction of *m* 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. To assert its liveness, an entity running on an *m* allocates a progress counter in the PB, then performs introspection and voluntary reporting by updating its value. An *M* reads the PB of *m* using RDMA read operations. If *M* detects that a progress counter has not been updated within its detection deadline, *M* considers that the corresponding entity has failed and decides what recovery action to take.

Use of RMC along with the PB abstraction for monitoring ensures that: *(i) m* is not directly involved in the observation process, i.e., observation of *m*'s state by *M* is essentially a 0-cycle operation on *m*. *(ii) m* does not interact with *M*; in general, it may be unaware of its existence. *(iii)* Participation of *m* in monitoring is voluntary, by local state introspection and reporting.

The recovery component of our system provides fine-grained recovery support for client service sessions after a failure. It is responsible for dynamic relocation (migration) of service sessions off of a monitored node *m*, following a failure detection by a monitor *M*.

To achieve fine-grained recovery, the system uses a per-client *State Box* (SB), a new OS-based mechanism that supports dynamic migration of live client sessions between communicating multi-process servers. Client-server communication takes place over TCP while server processes communicate over IPC channels. The SB model assumes that a service maintains well-defined *fine-grained* state for a client session and that client sessions are independent of each other. In this case, SB is an abstraction for encapsulation of both OS and application state associated with a client session. Application processes use a small SB API to save state that describe the point they reached in servicing the session, without forcing synchronization between processes or with the client. In case of failure, another node in the cluster reads the SB-encapsulated state from the failed node and uses it to locally reinstate the session state. The SB then synchronizes the application state with the OS state of the (TCP and IPC) communication channels to ensure consistent service to client.

Support of recovery via SB and transfer of SB state over RMC ensures that the recovery component: *(i)* is light-weight, imposing only minimal overhead during the failure-free execution for recovery support operations, and low cost during recovery; *(ii)* is silent during failure-free execution, i.e., it creates no background traffic in the system; *(iii)* does not involve failed nodes for state extraction during recovery, thereby it can recover a session even if its node is impaired by severe hardware/OS faults.

6. RELATED WORK

Recent initiatives in industry [14] and academia [19] seek to move the thrust of future computing systems from performance to self-managed, self-configuring, self-healing systems.

Several middleware solutions for software rejuvenation [13], self-healing [3], problem determination in complex systems [7] have been proposed. The techniques used in such systems are: gathering statistics at run-time through monitoring, detecting anomalies using these statistics, restarting or replacing the software components of a system. Such middleware is complementary to our work as the systems provide statistics through introspection, externalize the execution state, and provide components that can be replaced to repair the state at the application level. We propose similar techniques to be applied for repairing state while executing, and retrieving useful state after failure in a complete computer system.

In [5], an "all or nothing" approach is taken and the system is rebooted to bring it back to a consistent/working state following a failure. Our approach addresses a missing link: we reuse useful state in the system memory through repair and recovery rather than lose it by rebooting.

Self-adapting operating systems have been studied in the context of extensible OS research [21] where behaviour is modified by extending OS functionality with user code downloaded into the OS kernel. More recently, [25] 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.

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. The reliable file cache of [8] protects the cached 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. We also rely on OS or application state being available in system memory after a failure. However, we propose uniform mechanisms for controlled recovery from existing state and for repair of damaged state. The protection mechanisms explored in the above systems can also be used in our architecture.

In [28], RMC is used for recovery support in a cluster by mirroring the virtual address space of an application process on a remote node. This is the only work we are aware of in which RMC is used as support for fault-tolerant computing. In contrast, our approach involves RMC in all stages of automated reaction to failure: detection, recovery and repair. To our best knowledge, our work is the first to apply the RMC techniques to building self-healing systems.

7. 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 *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 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)*, 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 Server. Technical Report DCS-TR-427, Department of Computer Science, Rutgers University, November 2000.

[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. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[9] 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.

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

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

[12] D. Harrington, R. Presuhn, and B. Wijnen. RFC 3411: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks, December 2002.

[13] 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.

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

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

[16] J. Katcher and S. Kleiman. An Introduction to the Direct Access File System. White Paper, June 2000.

[17] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proc. USENIX Annual Technical Conference*, San Antonio, TX, June 2002.

[18] M. Mikic-Rakic, N. Mehta, and N. Medvidovic. Architectural Style Requirements for Self-Healing Systems. In *Proc. 1st Workshop on Self-Healing Systems*, Nov. 2002.

[19] 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.

[20] M. Rangarajan, K. Banerjee, J. Yeo, and L. Iftode. MemNet: Efficient Offloading of TCP/IP Processing Using Memory-Mapped Communication. Technical Report 485, Rutgers University, Sep 2002.

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

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

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

[24] E. G. Sirer, R. Grimm, A. J. Gregory, N. Anderson, and B. Bershad. Distributed Virtual Machines: A System Architecture for Network Computing. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sept. 1998.

[25] 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.

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

[27] 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. Symp. on Computer Architecture (ISCA)*, May 2002.

[28] 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.