SYSTEM ARCHITECTURES BASED ON FUNCTIONALITY OFFLOADING

BY ANIRUDDHA BOHRA

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Liviu Iftode

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2008

ABSTRACT OF THE DISSERTATION

System Architectures Based on Functionality Offloading

by Aniruddha Bohra

Dissertation Director: Liviu Iftode

Offloading to hardware components that support the primary task of a system enables separation of concerns and allows both the primary and offloaded components of a system to be easy to understand, manage, and evolve independent of other components.

In this dissertation, we explore the software mechanisms required to effectively offload functionality to idle processing elements. We present the design, implementation, and evaluation of three system architectures – TCPServers, Orion, and FileWall, which offload functionality for improving performance (TCPServers), improving availability (Orion), and for extending functionality (FileWall). We explore software mechanisms to offload functionality to a subset of processors in an Symmetric Multiprocessor (SMP) system, a programmable network interface, and an interposing network middlebox to realize the three system architectures.

TCPServers is a system architecture that offloads network processing to a subset of processors in an SMP system. Network processing imposes direct and indirect overheads on server systems. It directly affects system performance since it executes at a higher priority than application tasks and prevents other components of the system from executing simultaneously on the processors. It indirectly affects performance by causing cache pollution and Translation Lookaside Buffer (TLB) flushes, which lead to degraded memory system performance. Through offloading, TCPServers isolates the network processing and eliminates the direct overheads. We also present mechanisms for increasing network stack concurrency and connection

scheduling, which significantly improve the performance in a multi-threaded network stack.

Orion is a system architecture that enables *Remote Healing*, where the monitoring and healing actions are performed externally, by offloading such functionality to external hardware. Fine grained monitoring of computer systems for performance anomalies, intrusion detection, and failure detection imposes significant overhead on the target system. Performance critical systems, where such fine grained monitoring is essential, cannot be subjected to these overheads. Orion offloads the healing functionality to a programmable network interface, which provides an alternate path to the memory of the target system. Using Orion, monitoring and healing actions can be performed nonintrusively, without involving the processors of the target system. We present the design and implementation of mechanisms for remote monitoring and remote repair of damaged OS state using Orion.

FileWall is a system architecture that enables extension of network file system functionality. Network file system evolution is constrained by the tight binding between clients and servers. This binding limits the evolution of file system protocols and hinders deployment of file system or protocol extensions. For example, current network file systems have limited support for implementing file access policies. We propose *message transformation* as a mechanism to separate the client and server systems for protocol enhancement. FileWall offloads message transformation and policy enforcement to an interposing network element on the client-server path. We have used FileWall to implement file access policies and present experimental results showing policy enforcement at FileWall imposes minimal overheads on the base file system protocol.

The main conclusion of our research is that system architectures based on functionality offloading can be realized simply and effectively through efficient software mechanisms, using only commodity off the shelf hardware. With the availability of resources at idle processing cores in a multiprocessor system, intelligent peripherals, and unused nodes in a cluster, offloading is a practical solution for improving performance and introducing new functionality in computer systems and networks.

Acknowledgements

This dissertation would not be possible without the help, support, and guidance of my family, teachers, and friends - I want to thank them for all their efforts. I want to thank my advisor, Liviu Iftode, and my committee members - Ricardo Bianchini, Rich Martin, S. Muthukrishnan, and Jason Nieh, for valuable suggestions that made this dissertation better. The last-minute help from Rich actually made the dissertation possible. Several professors at Rutgers, whose courses I have taken and whose talks I have attended, helped me understand how little I know and how much I need to learn. I thank them all for their efforts and help through the years.

My advisor, Liviu Iftode, has been a firm believer in me and my work, more than I ever was. He has been a source of energy and ideas, many of which were wasted on me, but he has never given up. Through the years, Liviu has been a part of hundreds of arguments with me, some where I had something useful and productive to contribute, others where I was argumentative and foolish. I thank him for never discouraging me and helping me grow as a researcher and as a person.

Ricardo has been as much a professor and teacher to me as a friend and mentor. He has given valuable advice whenever I went up to him (and it was often). His enthusiasm and focus on the soccer field as much as in research has been an inspiration. I cannot imagine staying on in graduate school without his support and help.

Over the years, I have worked on projects with many people. At the beginning of my graduate study, Florin Sultan took me under his wing and made me realize the importance of patience, perseverance, and attention to detail. I still hear his words, “Don’t jump!”, whenever I am chasing bugs and getting nowhere. To him I owe more than I can thank him for. Stephen Smaldone has been a friend and my partner in the research through the last six years. He never lets me forget what the goal is in work as well as in life. It has been a pleasure working with Steve and I am honored to be his friend. Members of Discolab, especially Murali Rangarajan,

Cristian Borcea, Pascal Gallard, Iulian Neamtiu, Yufei Pan, Porlin Kang, and Arati Baliga, who over the years have contributed ideas, participated in discussions, and even helped with building systems and running experiments, have been a crucial part of my years at Rutgers. Lu Han and Tzvika Chumash gave valuable feedback and help during the final days before my defense. The long nights in the lab were always an interesting time due to all of them.

I have been lucky to have many friends who made the long time away from home enjoyable and interesting. Unfortunately, I cannot possibly thank each one of you here. I have known Budha and Hari for more than a decade and have always had a great time with them discussing science, cricket, history, politics, and everything else under the sun. I also want to acknowledge the colleagues at NEC Labs, who were understanding and supportive of me continuing to work on my dissertation while working there.

This dissertation would not have been possible without Kavitha, who through her support and continuous encouragement has kept me going. She is also responsible for introducing me to the pleasure of having pets, and for enriching my life with Mishki, Puff, Ollie, and Nicky. Together they have suffered my absences and handled the ups and downs of this journey. Finally, my parents and my brother are responsible for all that is good in me, the faults are all mine. Thank you all!

Dedication

To my parents

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Chapter 1

Introduction

1.1 Thesis

This dissertation investigates the role of functionality offloading in architecting high performance, highly available, and extensible systems. We believe offloading presents a practical solution to improve performance and introduce new functionality in computer systems by utilizing idle hardware resource. We demonstrate software mechanisms for functionality offloading through three system architectures that improve network performance, enable continuous monitoring, and extend network file system protocols. With the availability of resources at idle processing cores in multicore systems, intelligent peripheral devices, and idle nodes in a cluster, offloading presents an important and practical design principle for future system architectures.

1.2 Offloading Functionality in Computer Systems

Functionality offloading in computer systems can be roughly defined as identifying self-contained, computation intensive sub-tasks of all processing, and executing them on external hardware. This results in freeing up CPU cycles that can be used for other applications. Offloading reduces the overheads of main memory accesses and the ensuing cache pollution on the host system by placing memory on the external hardware. Finally, specialized hardware can be designed to ensure that offloaded tasks are processed faster than at the general purpose host processor.

For offloading to be beneficial, three conditions must be satisfied. First, the computer system must have hardware that can execute the tasks offloaded to it by the host CPU. Second, the host CPU must have other tasks to perform even after some functionality has been offloaded. Third, the overheads of offloading the functionality, which arise due to *data communication*

	Hardware	*Software*
Coprocessor	Floating Point Unit (FPU), Crypto accelerators	Emulated FPU, Programmable Graphical Processing Units [117]
Watchdog	Hardware sensors [55], AVIO [116], Copilot [147], GigaScope [57]	Defensive Programming [157], Recovery Oriented Computing [146], Cluster Based Services [46, 26]
Interposition (Internal)	TCP Offload Engines [10, 5, 25, 219], User level networking [23, 64, 67, 95], Network Attached Secure Disks [79]	Recoverable software [201, 158, 49], Virtual Machine Monitors [66, 214, 209, 21], Asymmetric OS [192, 128]
Interposition (External)	Active Networks [203], EarlyBird [182], Packet Filters [35]	Firewall [74], NAT [85], Overlay networks [164], Content Distribution Networks [211, 106], Internet Services [133, 40]

Table 1.1: Comparison of offloading architectures

and *state synchronization* between the host CPU and external hardware, must be low.

In this dissertation, we focus on offloading in the context of network server systems, which are the basic building block of Internet services. Servers that support Internet services are expected to *(i)* serve a large number of clients without performance degradation, *(ii)* be highly available and tolerate hardware and software failures, transparent to the clients, and *(iii)* be extensible to incorporate any new features required to support evolving application workloads and client expectations.

1.3 Offloading Architectures

System architectures have continuously evolved with application workloads to offload common tasks. Early system designers identified the inefficiencies in floating point arithmetic and designed coprocessors to improve its performance. Recently, cryptographic accelerators have been introduced to handle the increasing complexity of cryptographic operations in the growing number of security aware applications. However, computer systems are no longer used only for computation. For desktop systems today, high performance network access, large data storage, and entertainment and graphics are arguably more important. For server systems today, the ability to support a large number of concurrent clients while maintaining continuous service is of primary importance.

Offloading architectures can be classified into three broad categories based on the target

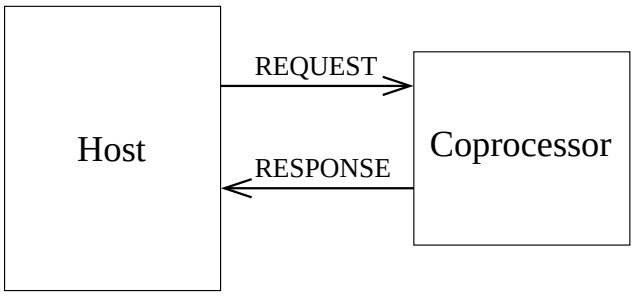

Figure 1.1: Coprocessor based offloading architecture

functionality - *(i)* Coprocessor, *(ii)* Watchdog, and *(iii)* Interposition. These architectures have been realized in hardware as well as software and are applied in a wide array of systems from desktops to collections of servers in Internet services and Content Distribution Networks. Table 1.1 presents a summary of instances of these architectures and we discuss them in more detail in the remainder of this section.

1.3.1 Coprocessor

Coprocessors are offloading targets that provide execution support to the host processor. CPU intensive tasks are isolated and executed externally at the target. Figure 1.1 shows a coprocessor based offloading architecture. Data and execution control is transferred from the host to the coprocessor through explicit requests. The host waits on the coprocessor to returns the results, synchronously through polling, or asynchronously through an interrupt.

Coprocessors are used to improve system performance by delegating tasks to dedicated, possibly specialized execution elements. For example, early microprocessors did not natively implement floating point arithmetic. Software emulation of floating point operations on these systems led to significant CPU overheads. Using a specialized coprocessor as the floating point unit reduces these overheads and improves overall system performance. Other examples of coprocessors include cryptographic accelerators, memory management hardware, etc., which are commonly used in desktop and server systems today.

Apart from specialization, programmable coprocessors can be used to implement functionality beyond their primary role in the system. For example, programmable Graphical Processing Units (GPUs), originally designed for 3D rendering, have been used to implement parallel processing engines kernels for general purpose computations [117].

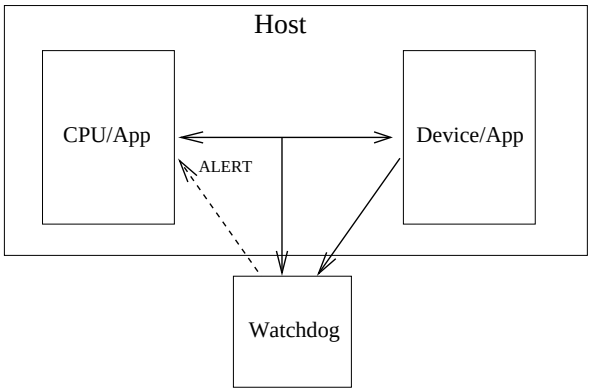

Figure 1.2: Watchdog based offloading architecture

Remote displays and graphical terminals [172, 165] use servers as coprocessors and offload all computation to them over a local area network. The host system simply performs graphical display and rendering, while the server handles all processing. User actions, e.g., keyboard and mouse events, are sent to the servers and the updates to the display are returned to the client. Recently, thin clients have been used over the wide area network to support seamless mobility and stateless computation at clients [20, 108].

1.3.2 Watchdog

Watchdogs are offloading targets that are programmed to monitor a subset of system state, execute independent of the host processor, and generate alerts on detecting unexpected behavior through interrupts or system messages. The watchdog functionality is an independent task and the host does not control execution of the watchdog, which can operate even when the CPU is not available due to OS crash or deadlock. Figure 1.2 shows a watchdog based offloading architecture. In the figure, the watchdog observes the communication between the CPU and a device, and triggers an alert to the CPU when unexpected behavior is detected.

Watchdogs are used to trigger tasks on the host system, which adapt the host behavior in response to external or unpredictable changes in the execution environment. For example, failure of a fan in the chassis of a system leads to increase in the CPU operating temperature, which may result in loss of expensive hardware. On detecting this event, a watchdog monitoring the fan would generate an alert and the OS may shut down the system to protect against failures. Watchdog functionality is an integral part of all computer systems today. Apart from

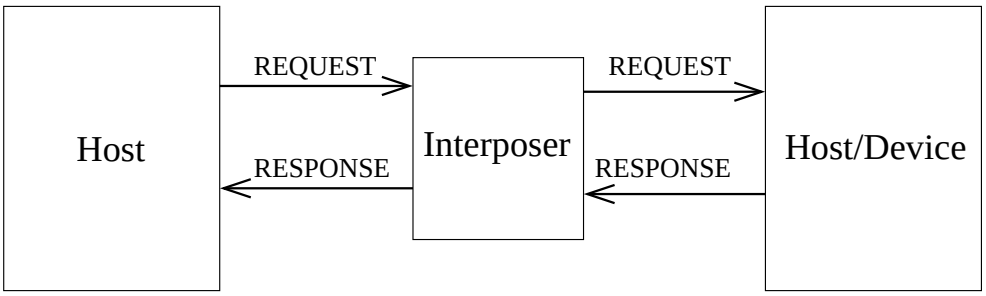

Figure 1.3: Interposition based offloading architecture

hardware sensors, common hardware watchdogs include device health monitors, e.g., RAID array monitors, and PCI bus monitors. Recently, watchdogs that monitor memory traffic have been introduced to detect leaking sensitive information [226], to design workload aware page replacement policies [223], detecting atomicity violations in programs [116], etc. Copilot [147] uses an external host as a watchdog to monitor the integrity of in-memory OS data structures to identify and detect incorrect or malicious modifications.

Watchdogs have been used in software to improve the resilience of applications. Defensive Programming [157] introduces a programming methodology for designing server applications to avoid denial of service attacks. In this system, sensors are introduced through code annotation and an external monitor observes the evolution of these sensor values. On detecting a violation of normal or expected behavior, an alert is generated and a specialized recovery handler is invoked to repair the system. In the context of Internet services, several systems have used watchdogs to continuously monitor client requests and responses, to monitor client perceived performance [139, 140], identify failures [48] or misconfigurations [134], and to initiate recovery actions through migration [199, 122, 102, 171], rollback [38], or recursive reboots [40, 49]. Watchdogs have also been used in cluster based systems to reconfigure the cluster in response to workload changes. These systems define observable metrics, e.g., response time, power dissipation, CPU and device idle times, etc., to define policies that are continuously evaluated by the watchdog. Alerts are generated for reconfiguration through request rerouting [11], and load balancing or unbalancing [88, 150, 46, 225].

1.3.3 Interposition

In an interposition architecture, offloading targets are placed on the interface boundary between two system components, e.g., the CPU and device controllers, to observe and police the communication between them. Figure 1.3 shows an interposition based offloading architecture. While in a coprocessor architecture, the host communicates explicitly with the offloading target, in an interposition architecture, *two* end-points communicate. Interposition can be realized *internally*, where all communication is within a host, and *externally*, where two autonomous hosts communicate over the network.

Hardware implementations of internal interposition architectures typically involve interposing on communication between the host CPU and one or more peripheral devices. The goal in such architectures is to offload device management to specialized hardware, closer to the device. Examples of internal interposition include protocol offload engines [10, 5, 25, 219, 76], which interpose between the host CPU and the network interface to eliminate network processing overheads at the host. User-level networking [23, 64, 67, 95] completely bypasses the OS and enables direct memory-to-memory communication over the network. Network Attached Secure Disks (NASDs) [79] offload query processing closer to the disks to reduce the overheads of data transfers across the I/O bus.

Software implementations of internal interposition monitor the interface across software components. Valgrind [61] annotates function calls and monitors the process address space to identify memory leaks, data races, and array bounds violations. Tripwire [207], virus scanners [202], and software firewalls [86, 155] are commonly used today to protect desktop systems from unwanted traffic, viruses, and worms. Rx [158] interposes on all memory accesses, identifies failures, and uses execution replay while modifying the environment to avoid bugs due to data races, scheduling priorities, signal delivery, etc. Nooks [201] interposes between device drivers and the OS and protects OS data structures against corruption due to bugs in the device drivers. Buffering and replaying transactions across this interface enables transparent restarts that would otherwise require a full system reboot. Infokernel [17] interposes between the application and the OS for implementing user defined policies without modifying the underlying OS. Infokernel relies on external observations and request transformation

to modify buffer cache replacement, network congestion control, and even process scheduling. Hypervisors [37] or Virtual Machine Monitors [66, 214, 209, 21] interpose on the OS-hardware boundary and in different instances verify, modify, or emulate hardware instructions to create a *virtual* execution environment for efficient resource sharing and to improve resilience of a wide variety of OS and application software.

In multiprocessor systems, software based internal interposition has been used to *offload* network processing to a subset of host processors. Such offloading isolates network processing from application processing. The Enhanced Transport Architecture (ETA) [161, 162] uses dedicated packet processing engines to isolate network processing from the rest of the OS. ETA also proposes the use of I/O Memory Management Units, which enable DMA across different memory regions to reduce the data copying overheads within the OS. AsymOS and Piglet [128, 192] partition a multiprocessor OS into device and host kernels that communicate using shared memory. The device kernel has its own scheduler and handles all device interactions while the host kernel executes applications.

External interposition architectures involve two autonomous hosts communicating over the network. Interposers sit on the client-server network path and are often called *middleboxes* [210]. These middleboxes are used to implement functionality that would require modification of the clients and/or servers. They are also used to protect end-points against unwanted network traffic [74], identify intrusions [213, 182], reduce network address requirements [85], virtualize end-points to support failover [122] and improve performance [144], and migrate processing within the network [199]. Middleboxes are the basic building block of overlay networks [7], which have been applied in almost all areas of distributed services, e.g., monitoring [222, 190], content distribution [106, 211], distributed storage [107, 131, 13, 194, 58, 167], content sharing [51, 39, 105, 44], resilient routing [7], multi-path transport protocols [221, 104], etc. In the context of Internet services, middleboxes have been used to improve client-visible performance and to maintain continuous service through load balancing [144] and client transparent failover [200, 197]. Such interposition has also been used to replay client requests and for replication of client requests to multiple instances of the same service for debugging and maintenance [134].

1.4 The Offloading Debate: Opportunity and Challenges

The growing use of the Internet has led to a large population of users who expect high quality, continuous service from the system infrastructure. Simultaneously, the increasing complexity of the applications provided over the network requires additional processing power. Offloading is attractive, as it frees up cycles on the host to perform complex application processing, handle a large number of simultaneous clients, and continuously monitor the health of the critical system infrastructure. Offloading is challenging, as it partitions the state of the system into multiple components that are maintained independently at the host and the offloading target. Maintaining consistent state across the host and the target requires data communication and synchronization, which in turn may require modifications to existing software. Communication and synchronization introduce overheads, while software modifications increase complexity and reduce maintainability of code.

In the following, we identify the technology trends and application requirements that create an opportunity for offloading while illustrating the challenges in realizing such architectures in the same context.

1.4.1 Offloading Network Processing

In current operating systems, the CPU implements the network protocol stack. As a result, network servers spend a significant fraction of CPU cycles processing packets within the network stack. Ideally, the system should spend as little time as possible processing packets and spend more time running applications.

Past research has identified that packet processing time is dominated by CPU stalls on main memory accesses, not by the execution of the instructions involved in the processing [50]. Historically, the data touching operations - checksum and memory copies, were identified as the primary source of memory accesses and avoiding them has been the focus of researchers for the past two decades. As a result of academic and industrial efforts, most Network Interface Cards (NICs) support checksum offloading, and DMA controllers, which eliminate the use of programmed I/O in the OS. Modern operating systems also support zero-copy send and receive interfaces [145] that avoid data copies across the user space and OS kernel boundary. User

level networking completely bypasses the OS from the network processing path and enables direct memory to memory communication [23, 64]. Recently proposed Remote Direct Memory Access (RDMA) [67, 95] standards and commercially available NICs implement memory to memory communication eliminating OS involvement in the critical path of network processing.

The techniques described above completely eliminate the checksum and data copy overheads in the modern network stacks. However, network processing also involves accesses to in-kernel data structures, for example, protocol control blocks (PCBs), shared queues, routing tables, etc., which result in main memory accesses. Interrupt driven network stacks cause disruption of application processing for handling network events leading to loss of locality of reference, cache misses and TLB flushes. In extreme cases, this may lead to receive livelock [125], where no useful processing can be performed by the system.

Data independent main memory accesses are difficult to avoid since the network protocols are layered and in some cases stateful. With a few thousand concurrent connections, these accesses overwhelm the CPU caches and result in main memory stalls. In modern processors, a main memory access may lead to a stall for up to 200 clock cycles and this overhead is getting worse with growing processor speeds. Compared to the typical packet processing time of 3000 clock cycles, it is easy to see that a few memory accesses may result in significant performance degradation.

In the past, the rapid increase in CPU frequency with the relatively slow growth in network bandwidth has allowed server systems to handle the demands of protocol processing. Today, we are witnessing a dramatic increase in the network bandwidth, both at the end-host, through faster NICs, and end-to-end, with growing access and core network bandwidth. Consequently, the Internet is being used to access applications, which have high bandwidth requirements and place significant processing load on the server systems. Internet games, virtual worlds, video and voice traffic, require servers to handle high network volumes and large numbers of clients while performing complex computation.

Offloading network processing in server systems satisfies the requirements of freeing up CPU cycles for use by other applications. It is also promising for reducing the memory accesses and improving cache efficiency. Finally, specialized hardware can be design to execute the protocol processing tasks faster than the general purpose host processors.

Despite the promise of performance benefits, network protocol offloading has its problems. Its benefits have been regarded by some to be *elusive* [181] and others to call it a *dumb idea* [124]. This skepticism is rooted in three primary observations. First, protocol offload only benefits workloads, which have a balance of both network and application computation. For others, offloading yields significantly lower improvements, which are offset by the increased complexity of the resulting system. Second, the global state shared by all network consumers, for example, port numbers and IP routing tables, may be updated *external* to the network protocol. Therefore, the corresponding state, maintained at the external hardware must be updated frequently. These accesses typically involve traversing high latency I/O buses leading to a degradation of performance. Third, the external hardware has fixed limited resources, which may cause it to become the bottleneck in the network processing path. Even in scenarios where the CPU can provide the extra cycles required to handle protocol processing, it is difficult to reconfigure the system at run-time to utilize them.

The issues raised by the opponents can be addressed today by using software mechanisms for offloading to idle general purpose processors. With the avaiability of multiprocessor and multicore systems, it is possible to dedicate a subset of processors for protocol processing. Therefore, the processing capability of the offloading hardware is identical to the host processors, the software can be developed, debugged, and maintained similar to the base OS, and the resources dedicated to network processing can be dynamically reconfigured based on the observed load.

In the context of using a subset of processors in a multiprocessor system as the offloading target, several systems have recently been proposed [161, 162, 128, 192]. The availability of multiprocessor servers has led to the performance bottleneck becoming the limited concurrency afforded by traditional interrupt-driven OS. Parallelization of network processing allows the system to take advantage of the multiple processing elements - hardware threads, independent cores on a processor die, or multiple processors in a Symmetric Multi Processing (SMP) system. Offloading network processing to a subset of processors isolates the network protocol processing. The system can be reconfigured at run-time to dynamically adapt to the load experienced by the system. The processors that handle network processing use shared memory to

communicate with the host, reducing the overheads of maintaining consistent state. The software that implements the offloaded functionality is a part of the host operating system and can process packets as fast as the host processor. In summary, offloading overheads are low, the modifications to existing software are limited, and the source code can be easily maintained.

In Chapter 2 of this dissertation, we present the design and implementation of TCPServers, a system architecture that offloads protocol processing to a subset of processors in a multiprocessor system. We also present mechanisms to improve the concurrency of the protocol stack using TCPServers as the basic building block of the system architecture.

1.4.2 Offloading System Monitoring

In large and complex system architectures commonly deployed today, identifying a failed component, fast and with high accuracy, is crucial. Large data centers routinely host hundreds of thousands of computer systems [81]. The problem is further compounded when the system is distributed geographically, e.g. in Content Distribution Networks (CDNs) like Akamai [4] and research networks like Planetlab [24]. At such large scales and distribution, hardware as well as software component failures are routine and occur frequently. Continuous maintenance and monitoring involving humans in these scenarios is becoming prohibitively expensive. Therefore, self-management of computer systems has become more and more the focus of systems research [146, 157, 158].

Traditionally, monitoring has been performed *within* the system, by executing a special monitoring task or daemon that observes the target application or a subsystem through statistics that represent the behavior of the system. Continuous monitoring of a large complex target uses system resources, for example, CPU, memory, disk, etc. which may not be available. Monitoring from within the system relies on CPU resources of the system and cannot detect or identify failures like crashes or deadlocks which render the system unusable. Finally, monitoring from within cannot reliably detect intrusions, since the monitoring task itself may be compromised.

Previous research has proposed observing the target system from an *external* monitor to overcome the limitations of internal monitoring [83]. External monitors can identify unresponsive or crashed systems through coarse grained periodic heartbeats. Fine-grained external

monitoring can be performed by a cooperative monitor and target system. The target system gathers statistics locally and exports this information to the monitor, either on-demand or continuously. The monitor provides resources to retrieve, store, and analyze the monitoring data. Unfortunately, external monitoring still relies on the target's resources to gather monitoring data. Moreover, the monitor must trust the data provided by the target, which may not be accurate if the target system is malicious or may be stale if the target system is overloaded. For example, a malicious target may deliberately provide incorrect data to the monitor to prevent detection, while an overloaded system may be unable to reply to the external monitor, leading to incorrect decisions.

Ideally, a monitor must observe all state changes at the target system, without consuming any resources at the target processor. The monitor must be transparent and the target system or application must not be able to detect, observe, or otherwise affect the operation of the monitor. Unfortunately, in complex systems, realizing an ideal monitor is impossible. System designers have focused on monitoring a subset of state changes by identifying *critical state* in system memory and defining a thin interface through which applications access this critical state. The monitors can then *(i)* interpose on this interface to observe and modify state updates, or *(ii)* observe the changes to critical state externally, identify anomalies, and make out-of-band modifications to recover or repair the critical state

Offloading monitoring functionality to external hardware enables the monitor to be isolated from the target system. With offloading, the monitor does not consume any target resources, can store monitoring data in local memory not visible to the target, can access target memory through DMA even if the target OS is crashed, hung, or is otherwise unresponsive, and can be protected against modification or tampering by the target system.

Despite the promise of strong isolation and low overheads, offloading system monitoring has its problems. These problems arise due to three fundamental limitations of an offloading based monitoring architecture. First, the external monitor does not interpose on the operations that access critical state. Therefore, the monitoring data generated is, at best, *delayed* and cannot prevent the illegal modifications to the state. Second, there is an inherent tension between the monitoring coverage and the resource requirements at the monitor. If the critical state being monitored is large or it generates a large volume of updates, the monitor requires

significant memory and CPU resources to process the data. Limited resources at the monitor lead to losing data, ignoring portions of the data, or limiting the processing. Conversely, by focusing on a small subset of system state, the monitor may ignore critical information outside the monitored state. In both cases, the monitor may generate false alarms or it may ignore critical events. Third, the monitor has access to system state through an alternative path to memory. Therefore, an insecure monitor may compromise the integrity of the target system by exposing memory contents and in some cases lead to incorrect or inconsistent state if monitor initiated modifications are permitted to critical state.

Offloading for monitoring has been studied in several contexts: sensors for monitoring temperature, fan speeds, etc., within a single system [55], monitoring applications [115, 157] and system software on a single node [201, 195, 158, 147, 66], monitoring cluster based applications [133, 163, 40, 146] or nodes [111], Internet scale monitoring of distributed services [182, 65], and the core IP network backbone [57]. In all cases, the goal is to generate a sequence of updates to the state, maintain a history of updates through logging, and use this historical log with a baseline profile of the system to identify anomalous behavior. The baselines are specified by developers and administrators, or they may be generated by creating a statistical model during system operation. The monitor generates alerts on detecting anomalies and these alerts can then be used to recover or repair the damaged state of the target system.

In Chapter 3, we present the design and implementation of Orion, a system architecture that offloads monitoring functionality to a programmable network interface or a cooperating monitor node in a cluster. Orion enables an alternative path to system memory for fine-grained monitoring of system and application state. We also present mechanisms for failure detection and remote repair of damaged OS state through Orion.

1.4.3 Offloading Protocol Extensions

In enterprise networks today, applications increasingly rely on key network services, for example, domain controllers, directory services, and network file services. These networks are well-managed, are protected from external accesses, and are expected to provide much higher performance and availability than that provided by the Internet. Internally, enterprise network services are accessible by all clients and typically provide unrestricted access to resources.

Therefore, managing enterprise services is a critical challenge for administrators today.

Administrators must exercise control over access to enterprise services, including monitoring performance, validating access rights, and implementing organizational policies. They must also continuously evolve their deployments to meet the needs of changing workloads, dynamic user communities, and evolution of legislative and organizational policies. The challenge is maintaining control and evolution without disrupting service to a large population of users. While the transport and network protocols are standardized and implement the least common denominator of functionality, application protocols, e.g., network file systems, must be extended to incorporate new functionality specific to an enterprise network. These extensions may fix an implementation bug, improve monitoring functionality, improve performance through non-standard optimizations, or extend functionality to implement policies beyond those supported by standardized protocols.

Traditionally, protocol extensions have been implemented by modifying the client and server systems. Unfortunately, such modification is intrusive and requires changes not only to a large and diverse client software ecosystem, but also to server software that may not be available.

Recently, network middleboxes have been used to extend and modify network protocols over the Internet [210]. Middleboxes interpose on the client-server network path and provide a platform for system designers to offload functionality for extending network protocols. Interposing middleboxes, for example, Firewalls [74] and Network Address Translators (NATs) [85] are an integral part of the current network architecture. Packet filters [35], network intrusion detection systems [213, 182], protocol proxies [106, 211], load balancers, etc., are increasingly being used to extend existing network protocol functionality.

Despite the advantages of transparent protocol extension and separation of concerns between the base and extended protocols, offloading application protocol functionality to middleboxes has its problems. First, such offloading breaks end-to-end semantics [169], which has been the guiding principle of network protocol design throughout its history. Intelligent network elements limit the control of the end points on protocol behavior. They also introduce possibilities of incorrect implementation or failures, which violate the assumptions made by the

communicating end points. Second, offloading protocol extensions requires middleboxes to understand protocol semantics. Since the state is maintained at the clients and servers, transparent extension requires the middlebox to infer the end-point state from the message streams. Out of band modifications of the end point state cannot be identified at the middleboxes, leading to incomplete or inconsistent knowledge of the protocol state. Moreover, any modification to the message streams may violate these semantics and lead to disruption of service. Finally, enterprise network protocols are performance critical and an additional interposing network element adds latency and may become the bottleneck. With network speeds approaching 10Gbps in the near future, performance limitations at the middleboxes may significantly degrade application performance.

In this dissertation, we focus on extension of network file systems using functionality offloading. In the context of network file systems, offloading to middleboxes has been used in Slice [11] to virtualize cluster-based file systems to scale up to large volumes of data storage and number of clients, and to provide quality of service guarantees to file system clients. Offloading authentication is provided by Kerberos [191] and various active directory protocols, e.g. LDAP, NIS, etc. In the context of wide area file systems, offloading has been used to provide content addressable caching [13] at middleboxes to reduce the number of round trips over the wide area network (WAN). Such offloading enables client visible file system performance over the WAN to be similar to that over a local area network. Extensions to the file system protocols at middleboxes to support monitoring [68], reordering requests [69], federating namespaces [204], and even repairing failed file servers [224] have also been proposed.

In Chapter 4 of this dissertation, we present the design and implementation of FileWall, a system architecture for offloading file system extensions to a network middlebox. FileWall uses message transformation to virtualize network file system endpoints, file system objects, and the file system namespace and allows implementation of complex file access policies. We use FileWall to implement a wide array of file access policies including monitoring, access control, and client transparent failover policies.

1.5 Summary of Dissertation Contributions

This dissertation has three main contributions in the area of system architectures using functionality offloading, which were published in the *Proceedings of the 3rd International Symposium on Network Computing and Applications, 2007* [159, 33], *Proceedings of the 1st International Conference on Autonomic Computing, 2004* [32], *Proceedings of the 3rd International Symposium on Dependable, Autonomic, and Secure Computing, 2007* [185], and *Rutgers University Computer Science Technical Report 569, 2005* [31].

We present the design, implementation, and evaluation of three system architectures – TCPServers [159], Orion [32, 31], and FileWall [33, 185], which offload functionality for improving performance (TCPServers), improving availability (Orion), and for extending functionality (FileWall) respectively.

TCPServers is a system architecture that offloads network processing to a subset of processors in an SMP system. Network processing imposes direct and indirect overheads on server systems. It directly affects system performance since it executes at a higher priority than application tasks and prevents other components of the system from executing simultaneously on the processors. It indirectly affects performance by causing cache pollution and Translation Lookaside Buffer (TLB) flushes, which lead to degraded memory system performance. These effects are even more prominent in Symmetric Multiprocessor (SMP) and Chip Multiprocessor (CMP) systems, where there are additional processing components that are idle due to limited concurrency afforded by traditional operating system architectures. We also present Receive Queues (RQs), OS data structures that enable early demultiplexing of incoming packets and connection aware scheduling of network processing. These mechanisms allow the system to support a large number of simultaneous connections while maintaining high throughput for each connection.

Orion is a system architecture that enables *Remote Healing* [198], where the monitoring and healing actions are performed externally, by offloading such functionality to a programmable network interface. Fine grained monitoring of computer systems for performance anomalies, intrusion detection, and failure detection imposes significant overhead on the target system.

Performance critical systems, where such fine grained monitoring is essential, cannot be subjected to these overheads. Moreover, healing from within may not be possible if the system state is corrupted, the system is hung or deadlocked, or is otherwise unable to execute the healing actions. Orion offloads the healing functionality to a programmable network interface, which provides an alternate path to the memory of the target system. Using Orion, monitoring and healing actions can be performed nonintrusively, without involving the processors of the target system. We present *Sensor Box* (SB), an OS data structure that collects monitoring data and exports it to the external monitor. Applications and OS subsystems can participate in the Orion monitoring framework by updating the SB through an API. We present two case studies, *(i)* failure detection, and *(ii)* remote repair of OS state damaged by resource exhaustion, which demonstrate the use of Orion mechanisms for Remote Healing.

Orion is an extension of the *Backdoor Architecture (BDA)* vision, where intelligent network interfaces are used to design automated management of computer systems without human involvement. In BDA, the focus is identifying critical system and application states and, on failure, restore this state on a healthy replica in a local area network. Monitoring in BDA is designed to identify failed nodes and is not extensible. In contrast, Orion takes a *holistic* view of monitoring OS subsystems and applications and designs mechanisms to allow applications to participate in the monitoring system. Unlike BDA, Orion also uses a local intelligent device for monitoring and is not limited to cooperative monitoring within a cluster.

FileWall is a system architecture that enables administrator governed extension of network file system functionality. Network file system evolution is constrained by the tight binding between clients and servers. This binding limits the evolution of file system protocols and hinders deployment of file system or protocol extensions. For example, current network file systems have limited support for implementing file access policies. We propose *message transformation* as a mechanism to separate the client and server systems for protocol enhancement. FileWall offloads message transformation and policy enforcement to an interposing network element on the client-server path. Through primitive FileWall policies, we demonstrate the mechanisms of attribute transformation, flow transformation, and flow coordination, to enforce a range of file access policies, from performance monitoring to client transparent failover. As a case study for complex policy enforcement, we present a Role-Based Access Control policy implemented

using FileWall without modifying either clients or servers. Through a Virtual Control Namespace, FileWall also eliminates the need for interface changes or specialized software agents that enable users to participate in the access control protocol. We present experimental results showing policy enforcement is accomplished by FileWall with minimal overheads on the base network file system protocol.

1.6 Contributors to the Dissertation

The following is a list of my colleagues who co-authored papers from which I used material in this dissertation, along with their contributions: Florin Sultan, Pascal Gallard (INRIA/IRISA), contributed to the design and implementation of the Orion system architecture. Iulian Neamtiu (University of Maryland), Stephen Smaldone, Yufei Pan, and Arati Baliga implemented applications for the Orion system. Stephen Smaldone implemented various policies for FileWall and contributed to the FileWall evaluation.

1.7 Organization of the Dissertation

This dissertation is organized as follows. Chapter 2 describes TCPServers, a system architecture that offloads network protocol functionality to a subset of processors in a multiprocessor system. Chapter 3 describes Orion, a system architecture that offloads system management functionality to a programmable network interface. Chapter 4 describes FileWall, a system architecture that offloads file system protocol extensions to a network middlebox. Finally, Chapter 5 concludes the dissertation.

Chapter 2

TCPServers: Offloading for Improved Network Performance

2.1 Problem Statement

Network processing performance has been the focus of systems research for more than two decades. At different times, especially when the bandwidth supported by the network interfaces has increased significantly, e.g., from 10Mbps to 100Mbps and from 100Mbps to 1Gbps, offloading network processing to the network interfaces has been proposed to alleviate the CPU overheads. Fortunately, the CPU speeds have correspondingly increased over the years to keep pace with the increased bandwidth, prompting some researchers to question the motivation of offloading and that of the benefits it offers [124, 181].

At present, we are witnessing another jump in the network bandwidth with NICs supporting up to 10Gbps. However, this increase is not accompanied by the optimism of ever increasing processor speeds due to gate complexity and temparature limitations in modern microprocessors. To counter the slowdown in the processor speed, CPU vendors are now providing multiprocessor hardware, where one or more execution contexts are present on the same processor die. While the increased parallelism provides an opportunity for scaling to higher network bandwidths, such parallelism is difficult to harness. The synchronization, interrupt, and memory overheads prevent existing network stacks to scale up to support available network bandwidths.

In this chapter, we present TCPServers, a system architecture to offload network processing to a subset of processors in a multiprocessor system. TCPServers takes advantage of available processors in a multiprocessor system to offload network processing while retaining the advantages of simplicity of interfaces and resource management, along with ease of deployment and maintenance, which have been the primary criticisms of protocol offload [124].

The primary benefit of offloading within a multiprocessor system using TCPServers is *isolation* of network and application processing. Isolation of network processing prevents application-network cohabitation that leads to cache and TLB pollution and involuntary context switches. We present mechanisms that take advantage of isolated network functionality to reduce the synchronization overheads and to increase concurrency of network processing. We introduce *Receive Queues (RQ)*, an OS abstraction, that enables scheduling network processing at the priority of the connections. RQs also enable early demultiplexing and graceful degradation of performance even when the system is overloaded.

2.2 Network Processing Overview

To support network processing, the OS must *(i)* provide applications with an interface to send/receive data, *(ii)* control the network devices to transmit data from memory to the interface and from the interface to the system memory, *(iii)* perform protocol processing, e.g. IP, TCP, etc. In the following, we trace the path of a packet in an operating system.

To simplify the discussion, we focus on the TCP/UDP/IP protocol suite, and on widely deployed and implemented BSD-derived network stacks. We do not describe protocol details since our focus is the end-host processing from the Operating System perspective.

2.2.1 Sockets

Sockets are data structures that support communication between user applications and the OS kernel. Sockets perform two primary functions in the network stack: *(i)* provide applications an interface to send and receive network data, and *(ii)* virtualize the implementation details of individual protocol stacks. The kernel representation of sockets consists of state variables, e.g., reference count, socket state, non-blocking status, etc., send and receive queues, a reference to the protocol implementation, e.g. IPv4, and a reference to the protocol control block representing the connection,.

A mapping between the kernel socket representation and the user interface is maintained through a socket descriptor, which is returned to the application when the socket is created. The system call interface exports the sockets as file descriptor and enables the application to

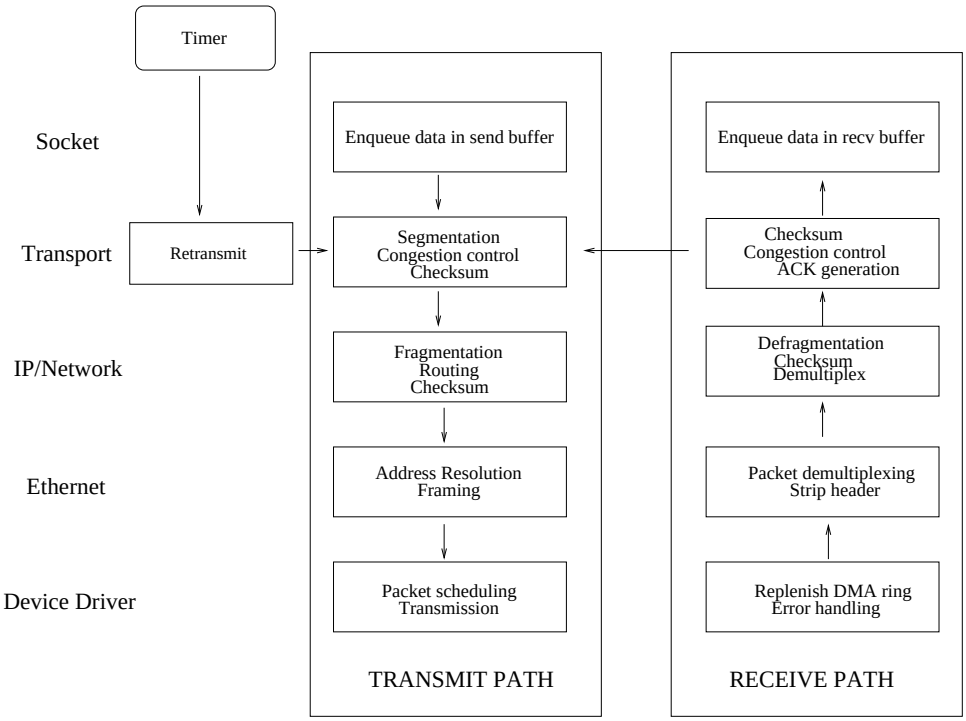

Figure 2.1: Transmit and receive paths in an OS network stack

communicate with the Operating System kernel.

2.2.2 Network Buffers

Network data is maintained in the OS in chains of memory resident kernel buffers, called mbufs in *BSD systems and skbufs in Linux. These buffers represent a *packet* that contains application data and protocol headers to construct a self-contained unit of network transmit and receive data. Network buffers are small size contiguous regions of memory that are reference counted to allow efficient sharing and manipulation through different components of the network stack. In the following discussion, we use mbufs as the representative name for the network buffers.

2.2.3 Transmit Path

Figure 2.1 shows the transmit path through the OS networking stack. Application data is copied from application buffers into newly allocated mbufs in the socket send queue using the `send` system call.

For datagram sockets (UDP/IP), the mbufs are handed to the UDP output routine, which

constructs the UDP datagrams, adds the transport header, and calls the IP transmit routine for downstream processing.

For TCP/IP sockets, the OS checks the connection state to compute the available capacity on a connection using its congestion control algorithm. If the connection can accomodate data, application data is divided into segments and a transport header is created for each segment. TCP segments are cloned and sent to the IP transmit routine for downstream processing. In addition to the application send context, the TCP transmit functionality is invoked when acknowledgements are received and when retransmission is required. In all cases, TCP segments are sent to the IP output routine for further processing.

During IP processing, application data is broken up into maximum transmission unit (MTU) sized fragments, the IP address next hop neighbour is identified, and an IP header is added to each fragment. The OS maintains a global routing table, which is used to identify the IP address of the next hop in the network path. A routing table entry contains the next hop IP address, and the Medium Access Control (MAC) addresses of the outgoing network interface and the next hop neighbor. The MAC address of the next hop neighbour is determined on demand, using the Address Resolution Protocol (ARP), and is added to the global routing table. Once the MAC addresses are added to the IP datagram, the packet is complete and can be transmitted over the network.

If the network interface is idle, the packet is queued for a DMA transfer to the NIC. If the interface is busy, the packet is queued in a software interface queue to be transmitted when the network interface is ready to accept new packets for transmission.

The OS performs the above tasks in the context of the application process unless the interface is busy and the packet is queued on the interface transmit queue. The packets on the interface queue are transmitted in the software interrupt context when the NIC indicates to the OS that it is ready to send new packets.

2.2.4 Receive Path

From the OS perspective, network receive processing starts with the NIC generating an interrupt indicating that packets have been received. Historically, each packet required at least two interrupts, first to indicate the packet arrival to prompt the OS to initiate a DMA transfer from

the NIC to the host memory, and second to indicate completion of the transfer. Modern NICs can initiate the transfer without involving the host CPU. In the following, we assume the OS maintains a ring of pre-registered DMA descriptors and the NIC generates an interrupt only when the packet has been transferred to the host memory.

Interrupt service routines (ISRs) execute at the highest priority in the system and preempt the currently executing process on the CPU to borrow its context. Therefore, all other activity in the system is suspended while interrupts are being handled, and the ISR must not perform CPU intensive tasks and defer these tasks for the software interrupt context.

Typical network interrupt handlers perform the following essential tasks : acknowledge the interrupt and perform book-keeping. Since each packet transfer consumes a DMA descriptor, the interrupt handler also replenishes the depleted DMA descriptors and registers them with the NIC. Most network receive processing is deferred for the software interrupt handlers, which execute at a higher priority than all user processes.

In the software interrupt handler, the packets are classified and the protocol specific input handler is invoked. For IP packets, the IP input handler defragments the packets and forwards them if the destination is not the current host.

IP packets destined for the current host are further classified according to the unique 5-tuple connection identifier, $< srcIP, dstIP, srcPort, dstPort, Protocol >$, where srcIP and dstIP are the source and destination IP addresses respectively, and srcPort and dstPort are the source and destination port numbers respectively. Protocol identifies the transport layer protocol, which handles this packet. The 5 tuple uniquely identifies the connection and consequently the socket to which the packet belongs and invokes the transport layer protocol handler referenced in the socket.

For the UDP/IP protocol, the packet is queued directly in the socket receive queue. If an application is blocked waiting for receive on the socket, it is woken up and a transfer is initiated to copy the data to the application.

For the TCP/IP protocol, packets are queued to the socket input queue in sequence. If the packet is out of sequence, it is maintained in a reorder queue until the gaps are filled by subsequent packets. TCP protocol handlers are also responsible for handling and generating acknowledgements and updating the congestion control information. The data is copied to the

application in the context of a `recv` system call issued on the socket.

2.2.5 Network Performance Overheads

Network processing overheads can be classified into per-byte and per-packet overheads. Per-byte overheads are due to operations that access each byte transmitted or received over the network, for example, checksum calculation and verification, data copies across the user-kernel boundaries, etc. Per-packet overheads arise due to operations performed by the OS on receiving and sending a packet, for example, interrupt processing, DMA setup and teardown, transport layer processing, etc. Since a packet typically contains thousands of bytes, the per-byte operations contribute to the majority of the CPU cycles consumed by network processing. Per-byte overheads can be significantly reduced or avoided altogether using zero-copy protocols, and previous research, discussed in Section 2.7, has demonstrated both hardware and software mechanisms for the same. Unfortunately, even with low per-byte overheads, per-packet operations contribute significantly to the processing time, which gets worse for multiprocessor implementations of the network stack.

Network processing imposes direct overheads by consuming CPU cycles and limiting OS concurrency. Software interrupt handling executes at a higher priority than application processing and directly prevents other tasks from executing in the system. This significantly reduces the performance observed by applications running on the system, and, in extreme cases, leads to receive livelock [125]. In multiprocessor systems, global data structures, for example connection tables, packet queues, memory allocators, etc., are protected by expensive synchronization that further limits network stack concurrency, reducing overall system performance.

Indirect overheads of network processing arise due to the cohabitation of application and network processing and are manifest as diminished locality of reference and increased context switches. The loss of locality of reference results in higher cache and Translation Lookaside Buffer (TLB) misses, which reduce the throughput of the system. Similarly, a context switch requires the OS to take a snapshot of the process state in memory and restore it when execution resumes, which increases the time spent in the system scheduler. Context switches also cause a large number of cache and TLB misses and pipeline flushes.

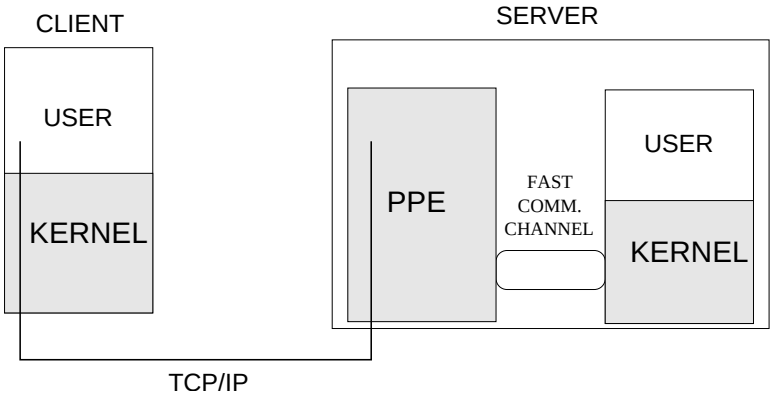

Figure 2.2: TCPServers Architecture.

2.3 TCPServers Architecture

TCPServers is a system architecture for offloading network processing from application hosts to dedicated processors in a multiprocessor environment, nodes in a cluster, or intelligent devices in a cluster of intelligent devices (CID). TCPServers partitions the network processing into application and dedicated packet processing engines (PPEs). Application processors (nodes) are separated from the PPEs to alleviate the inefficiencies introduced by cohabitation of the network processing.

Figure 2.2 shows the TCPServers architecture. The application processors and the PPEs communicate over fast communication channels. For multiprocessor environments, the communication is over shared memory, for cluster and CID environments, the communication between application processors and PPEs is over a high-bandwidth, low-latency interconnect, e.g. VIA or Infiniband.

In a multiprocessor system, which is the focus of this dissertation, PPEs handle the asynchronous events, e.g., NIC interrupts, network timers, deferred send processing, and receive processing. Since the majority of the send processing is performed in the context of the sending process and does not generate significant overhead when using zero-copy send interfaces, communicating with PPEs through shared memory and Inter-Processor-Interrupts (IPIs) is avoided. Therefore, PPEs are invoked in the send path only if the network processing is deferred, e.g. waiting for interface queue to be drained, or for ARP processing.

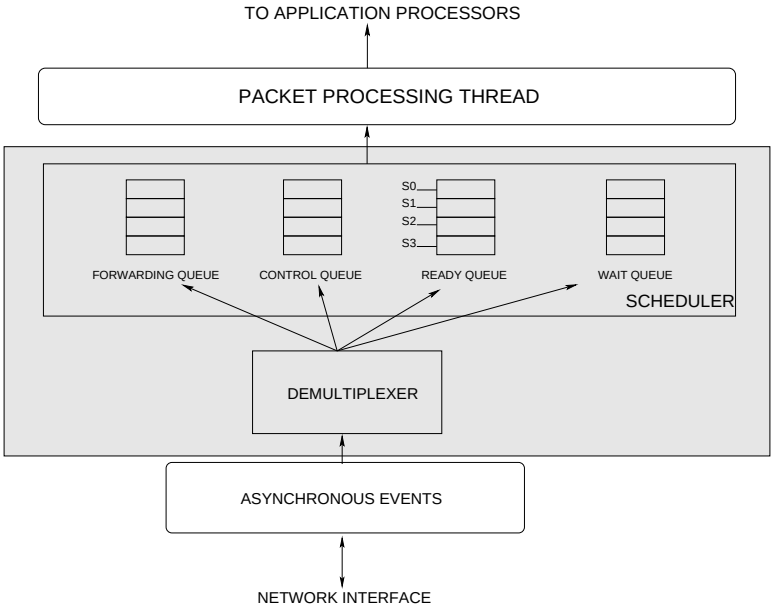

Figure 2.3: TCPServers Packet Processing Engine

Receive processing for TCP/IP protocol requires asynchronous protocol processing to generate ACKs, to send any pending data in response to received ACKs, and to retransmit packets on loss detection or on timeouts. These tasks cannot be delayed to be performed in the context of the receiving process, therefore PPEs handle majority of the receive processing. The receiving process context is used only to perform the copy from the kernel to user buffers.

2.4 TCPServers Design

Figure 2.3 shows the components of a TCPServers Packet Processing Engine (PPE). Each PPE has three main components: an asynchronous event handler, a scheduler, and a Packet Processing Thread (PPT). In the following, we describe the design guidelines and the design of the individual components in the TCPServers software architecture.

TCPServers is designed to minimize application and network processing cohabitation. Such cohabitation leads to increased involuntary context switches, cache and TLB misses, and unfair resource allocation to network processing due to higher priority of interrupts in the system. However, a naive isolation of network and application processing may *reduce* the network server performance.

2.4.1 Protocol Processing Threads and Asynchronous Event Handling

TCPServers performs protocol processing in a process context instead of the software interrupt context. Kernel-resident processes, called protocol processing threads (PPTs), performs network processing and are bound to each packet processing engine. An active PPT never yields the processor since network receive processing is non-blocking and runs to completion. Moreover, the scheduler does not schedule any other process on PPEs. During startup, a PPT is created for each processor in the system and is woken up using an inter-processor interrupt on activation. The enables TCPServers to dynamically reorganize the PPE and application processor sets in a system.

TCPServers uses an adaptive polling mechanism for handling NIC generated events. The NIC generates an interrupt on the network activity edge. The first received packet generates an interrupt, which is followed by a sequence of polls to handle received packets. During polling the NIC interrupt mechanism is disabled. When polling generates no new events, interrupts are re-enabled at the NIC.

TCPServers assigns a statically configured Master Packet Processing Engine (MPPE) to handle all interrupts. All interrupts are diverted to the MPPE using the I/O APIC redirection mechanism. Polling is performed in the software interrupt context (softirq), which executes at the MPPE at a higher priority than the PPT.

The choice of a single processor to handle all interrupts appears to be limited. However, isolating the interrupt handler and the softirq at the master PPE reduces cache pollution at the application processors as well as on other PPEs. Moreover, since a single processor handles packets from the NIC, packets are read off in the order they are received at the NIC, eliminating packet reordering. Finally, even if additional processors were used to handle the interrupt, only one softirq thread could execute at a time in the system, therefore yielding no additional parallelism than afforded by our architecture.

2.4.2 Receive Queues

We introduce Receive Queue (RQ), an OS data structure used by TCPServers to enqueue protocol processing requests to PPTs or application processes. An RQ is a mailbox associated

with each socket, where received packets are queued by the asynchronous event handler and are dequeued by the PPT. The RQ is addressed by the connection identifier that identifies a socket.

Receive queues are implemented as fixed sized arrays, which are protected by per-CPU atomic operations, for example test and set or compare and swap operations. Since there is one producer (asynchronous event handler) and one consumer for each queue (PPT), the atomic operation based mutual exclusion is sufficient. This eliminates not only expensive synchronization operations but also the contention on the globally shared IP queue.

Receive queues are associated with a socket when the socket's address is identified on one of the following events: *(i)* when local ports are allocated implicitly or through a `bind()` system call, *(ii)* the OS creates a new socket on a successful connection, *(iii)* when sockets are bound to the same UDP multicast group. Globally shared queues are created at startup for packets received for unknown connections or IP fragments that do not contain the transport headers.

2.4.3 Early Demultiplexing

Traditional networking stacks handle hardware interrupts and queue the received packet in the IP queue for processing in the software interrupt context. TCPServers eliminates the IP queue. Instead, the asynchronous event handler demultiplexes the incoming packets early into per-socket receive queues.

Packet demultiplexing in the asynchronous event handler uses the Ethernet header to identify the target network protocol (ARP, IP, PPP, IPX etc.). IP packets are further classified using the five tuple flow identifier to uniquely identify the connection. Packets are queued into the corresponding receive queue only if there is an empty slot and are dropped otherwise. IP fragments that do not contain a transport layer header are placed in a globally shared receive queue for future reassembly.

Through early demultiplexing and fixed sized per-socket receive queues, TCPServers creates a feedback mechanism. When applications cannot handle the packets at the incoming rate, receive queues overflow and further packets are dropped. On the other hand, well-behaved applications that can handle the rate of incoming packets enjoy greater stability under load.

2.4.4 Scheduling Network Processing

TCPServers requires a mechanism to schedule network protocol processing in the PPT context. Integrating the PPT with the system-wide scheduler defeats the purpose of offloading. Therefore, we need a mechanism independent of the system scheduler, which prioritizes network event processing across receive queues. Network processing has several unique characteristics which affect the design of such a scheduling algorithm:

1. Scheduling a *network processing task* does not involve a context switch. Instead, it is a function call and therefore does not incur any additional overhead.

2. Network processing in the PPT always runs to completion. The tasks do not block waiting for other tasks or OS resources, e.g. IPC or disk. Therefore, no additional state must be preserved across executions.

3. Network processing priority is determined not only by the resources spent on servicing the socket, but also by the network *protocol semantics*. For example, IP forwarding packets must be handled at a higher priority than the packets destined for the host; in a busy server, packets that terminate connections (FIN, RST) must be handled as soon as possible to reclaim system resources.

Packet Classes and Scheduling Queues

The PPT must handle packets for IP forwarding, packets received for active sockets (connected TCP and bound UDP sockets), connection requests for new connections, and packets belonging to multiple protocols, e.g. TCP/UDP, ICMP, ARP, etc.

We classify network packets which are processed by the PPT into the following categories in decreasing order of priority.

- *Forwarding Packets:* Packet forwarding systems, e.g., software routers, must handle packets that must be forwarded at a higher priority than packets for the host itself. Such requests are rare or are disabled altogether for server systems.

- *Control Protocol Packets:* ICMP, ARP, and RARP protocols are essential for other network communication to continue. Therefore, these packets have a higher priority than

other higher layer protocols. Classifying ICMP packets further, ICMP error messages, e.g. for MTU errors, are higher priority than the requests, which can be delayed further.

- *Connection Teardown Requests:* Packets that cause a connection teardown free up resources at the server and are therefore highest priority packets destined for connected sockets.

- *Packets for Existing Sockets:* Data packets belonging to a connected socket are assigned a higher priority than requests for new connections (SYN) packets. New connection requests are used to admit new clients for service, and a busy server, which is unable to keep up with existing clients, would not want to spend system resources on new requests.

The packet classes above are identified by the asynchronous event handler at the time it performs early demultiplexing into receive queues. We maintain four classes of queues : *(i)* Forwarding Queue, where receive queues for forwarding are maintained, *(ii)* Control Queue, where receive queues for control protocols and for sockets where a connection teardown request has been received are maintained, *(iii)* Ready Queue, where all receive queues with packets pending for service are maintained, and *(iv)* Wait Queue, where the remaining receive queues are maintained.

The packets in a higher priority queue are serviced before moving to the next queue. Except the Ready Queue, packets in all other queues are of identical priority, therefore a simple round robin scheduling mechanism is used to service those queues in the PPT. When more than one processor is used as a packet processing engine, the master PPE handles all queues, while other processors handle only the Ready Queue.

The Receive Queues in the Ready Queue are organized further into priority groups based on the responsiveness of the application and the number of pending packets received for the socket. The Receive Queues belonging to the same priority group have similar priorities. The TCPServers scheduling algorithm tries to ensure each priority group receives its fair share of the PPE CPU resources.

Scheduling Priority

The Receive Queue priority captures the responsiveness of the parent application (socket). Applications that are unresponsive have data waiting in the socket queues for `receive` system calls. Therefore, packets received for these applications, can be delayed. On the other hand, applications with pending send buffers require urgent service, since these applications are waiting on the network subsystem to send out the bytes already present in the socket buffers. Finally, if there are neither pending sends or receives, applications that have received more packets require service before those that have received only a few packets, since the system has invested more resources in handling them.

The scheduling priority or weight that captures the above criteria allows the TCPServers scheduling to ensure prioritized packet processing for active and responsive applications. We define the scheduling priority as a linear combination of the weights assigned to the pending send and pending receive packets at the socket, and the pending packets in the RQ denoted by W_s, W_r, and W_q respectively. These weights are defined for a socket i as

$$W_s(i) = \frac{PendingSendBytes(i)}{SegmentSize(i)} \tag{2.1}$$

$$W_r(i) = \frac{MaxRcvBytes(i) - PendingRcvBytes(i)}{SegmentSize(i)} \tag{2.2}$$

$$W_q(i) = NumPendingPackets(i) \tag{2.3}$$

$W_s(i)$ and $W_r(i)$ are approximate number of pending send and receive *packets*. In the above, $MaxRcvBytes(i)$ represents the socket's receive buffer size, and $SegmentSize(i)$ is the size of a segment of data that is sent out over the connection. While this measure is not exact, it approximates the number of packets pending attention and the number of packets that can be served for this socket.

Unfortunately, the equations above ignore the applications that set a very large receive socket buffer. In that scenario, $MaxBytes(i)$ dominates $W_r(i)$ and cannot be easily compared against other sockets. To overcome this limitation, we normalize the weights and redefine the

weights as

$$\widehat{W}_s(i) = \frac{W_s(i) \times MaxSndBytes}{MaxSndBytes(i)} \quad (2.4)$$

$$\widehat{W}_r(i) = \frac{W_r(i) \times MaxRcvBytes}{MaxRcvBytes(i)} \quad (2.5)$$

$$\widehat{W}_q(i) = \frac{W_q(i) \times MaxRQSize}{MaxRQSize(i)} \quad (2.6)$$

To avoid fractional weights, we scale the normalized weights by the maximum allowed send and receive buffer sizes, and redefine the scheduling weight as

$$W(i) = \widehat{W}_s(i) + \widehat{W}_r(i) + \widehat{W}_q(i) \quad (2.7)$$

TCPServers Scheduling Algorithm

Algorithm 1 TCPServers scheduling algorithm

```
1: TCPServerSchedule
2: S = TCPServerGroupSchedule(G_i)
3: W_i = W_i + 1
4: if i < g and (W_i+1)/(W_i+1+1) > φ_i/φ_i+1 then
5:     i = i + 1
6: else
7:     i = 1
8: end if
9: return S
```

TCPServers uses a proportional share scheduling algorithm derived from the recently proposed Group Ratio Round-Robin (GR^3) [43] CPU scheduling algorithm. The GR^3 algorithm provides constant fairness bounds on proportional sharing accuracy with O(1) scheduling overhead.

Figure 1 shows the algorithm used by TCPServers to schedule network processing at the PPT. ϕ is the weight assigned to each group G, while D is the deficit maintained within a group to perform a deficit round robin scheduling; $curpos$ is the current position in the group. The $TCPServerSchedule$ subroutine picks the next group to schedule (G_i) and passes a pointer to this group to the $TCPServerGroupSchedule$, which returns the socket whose packets are processed.

The key idea is to define groups of RQs (clients) with closely related priorities and keep the groups in sorted order, while keeping the RQs (clients) within the group in an unsorted list. Scheduling across groups uses the ratio of group weights (sum of weights of all members of the group) to determine which group to select. Since the groups are kept in a sorted list, this operation compares only two group weights at a time yielding a constant time decision. Within the group, RQs are scheduled using a simple Deficit Round Robin algorithm, where the RQ is scheduled for time slots proportional to its normalized weight within the group. Fractional shares are deferred for the next round where the *deficit* is added to the client's weight.

Groups are defined for RQs with closely related weights. The *closeness* in weights is defined as a logarithmic order, where a group of order σ, contains all clients whose weights ($W(i)$) follow

$$2^{\sigma} \leq W(i) \leq 2^{\sigma+1} - 1 \tag{2.8}$$

Using the logarithmic relationship reduces the number of groups and therefore the maintenance overheads for the sorted group list. Moreover, all clients within a group have their priorities within a factor of two.

Unlike in GR^3, the RQ weights change frequently, on receiving a packet and on sending packets. TCPServers recalculates the priority for all RQs on which packets are received before returning from the asynchronous event handler. The RQs are removed from their current group and are re-inserted into the group which satisfies Equation 2.8. Removal and insertion of an RQ is an $O(g)$ operation, where g is the number of groups in the system since a linear scan is required to identify the group for the modified RQ. However, even for large 32 bit weights, there are a small number (32) queues in the system and this scan is not too expensive.

In a server system, where there are few applications, and there are few outstanding send bytes in the socket buffer, the number of packets received in a burst is the only source of modification to the priority. All updates for a receive queue due to packets received in one invocation of the asynchronous event handler are batched, therefore the updates represent only a fraction of the overall CPU overheads.

2.5 Prototype Implementation

We implemented TCPServers in FreeBSD-7 Current, which implements a multithreaded network stack and uses fine-grained synchronization based on mutual exclusion primitives: mutexes, locks including read-write locks and spinlocks, and condition variables. This operating system is multithreaded and scales well for medium sized multiprocessor systems. It has support for adaptive polling API and uses a deferred task-queue to handle asynchronous events.

We use the fast interrupt handler for handling the network interrupts instead of the default interrupt thread based implementation. The driver implements a deferred task-queue based interrupt handling, where the interrupts are acknowledged and the task-queue is responsible for initiating the DMA and replenishing the DMA descriptors.

We implement three versions of the TCPServers, *(i)* TCPServers, where the PPT is bound to one processor of the multiprocessor, *(ii)* TCPServers-Early, where we implement early demultiplexing through Receive Queues in addition to the base TCPServers implementation, and *(iii)* TCPServers-Sched, where we implement our socket aware scheduling algorithm in addition to the TCPS Early implementation.

TCPServers is implemented as a loadable kernel module and requires minor modifications to the default OS. The OS modifications are limited to the scheduler and to the device driver. The TCPServers module installs a private network stack, which has minor modifications to the default FreeBSD stack.

The TCPServers stack registers as a consumer for the netgraph ethernet node, `ng_ether` [153], which hands over all network packets to the TCP server asynchronous event handler in the interrupt context. A new protocol family, (`PF_TCPS`), is used to invoke the system calls defined in the TCP server stack for application sockets to allow easy debugging and to localize changes within the module. These changes are not required if the default stack includes the non-intrusive modifications.

2.5.1 Dedicating Processors for Network Processing

To implement TCPServers functionality, we create kernel resident processes (*kthreads*), which execute the packet handling code for the receive part of the network processing. We disable

pre-emption and execute an infinite loop within the kthreads to isolate the CPU for network processing.

To enable dynamic reconfiguration, we create a *kthread* for all processors in the system and bind them to a CPU using the scheduler thread *bind* API. Specific threads are woken up by the inter-processor-interrupt (IPI) mechanism, which allows a processor to generate a trap on one or more processors in the system. The trap handler executes a function specified in the IPI.

All kthreads are initialized in the sleeping state. The system maintains two bitmaps : *(i)* AllowedPPE represents the subset of processors that are *candidates* for being a dedicated PPE. This bitmap prevents all processors being assigned to network processing, preventing all other activity in the system, and *(ii)* ActivePPE represents the set of processors currently dedicated to network processing.

During initialization, a Master PPE is chosen statically, the kthread assigned to its CPU is woken up, the ActivePPE bitmap is updated, and interrupts are rerouted. We modified the I/O APIC interrupt control infrastructure to enforce routing all *network* interrupts to a dedicated processor (Master PPE) and disable all other interrupts from being delivered to this processor. We also bind the network interrupt task queue (softirq) handler to the Master PPE to ensure all asynchronous events are delivered to the Master PPE. To ensure that the scheduler does not schedule threads on the PPEs, the ActivePPEs are removed from the candidate processor set.

2.5.2 Receive Queues

Receive Queues are implemented as fixed sized arrays and are protected using atomic variables instead of expensive locking primitives. These queues are First In First Out (FIFO) data structures that store the packets received by the asynchronous event handler and accessed by the TCP server PPTs. The PPT queues any data to be delivered to sockets after IP and transport (TCP/UDP) processing in the default socket receive buffers. The data is copied out to the process address space in the context of a receive system call.

Receive Queues are associated with sockets, and each socket has a unique RQ. Receive Queues are assigned a unique addresses based on the socket's network address 5-tuple. The queues are stored in a hash-table indexed by this address and a reference is kept in the socket data structure.

RQs are created when the application calls the `bind` system call or when a stream based socket is connected. Additional Receive Queues are created for out of order IP fragments that do not have transport headers. Separate kernel threads and receive queues are created for handling control protocols like ARP, RARP, and ICMP. RQs are destroyed when the parent socket is closed, or the TCP connection is disconnected. Receive Queues for the control protocols are never destroyed.

2.6 Evaluation

We perform all experiments using an SMP Dell Poweredge 2600 server with two 2.8 GHz Intel Xeon processors, each with 512KB L2 cache. The system has 3GB RAM, and two Intel 82544 gigabit Ethernet adapters connected over a 66MHz/64bit PCI-X bus interface. We use two identical clients connected back-to-back with the server interfaces using cross-over cables. In all our experiments, the client systems are not overloaded and the performance is determined solely by the server system. The server runs a modified version of the FreeBSD-Current as of September 20, 2006, while the clients run Linux Fedora Core 3 with a 2.6.18 kernel.

Benchmark: Most existing network performance benchmarks are not designed for a large number of simultaneous connections and typically measure the peak bandwidth using a single connection. To overcome this limitation, we implemented a benchmark program using the libevent event driven programming library [156]. The benchmark operates in two phases. During the startup phase, it establishes the requested number of connections and, in the measurement phase, the server sends data to the client over each of the connections. The data is obtained from an in-memory file using the zero copy `sendfile` interface. We set the send and receive buffers for all sockets as 64KB, as this yields the maximum performance in our experiments. The benchmark runs for 3 minutes (180 seconds) and the data is continuously transferred for the duration of the test in fixed sized blocks of 8KB. The clients discard all received data and send an acknowledgement at the end of each block. On receiving the acknowledgement, the server queues the next block for transfer to the socket. For fine-grained measurements, we use the hardware performance monitoring counters exported by the `hwpmc` driver in the FreeBSD kernel.

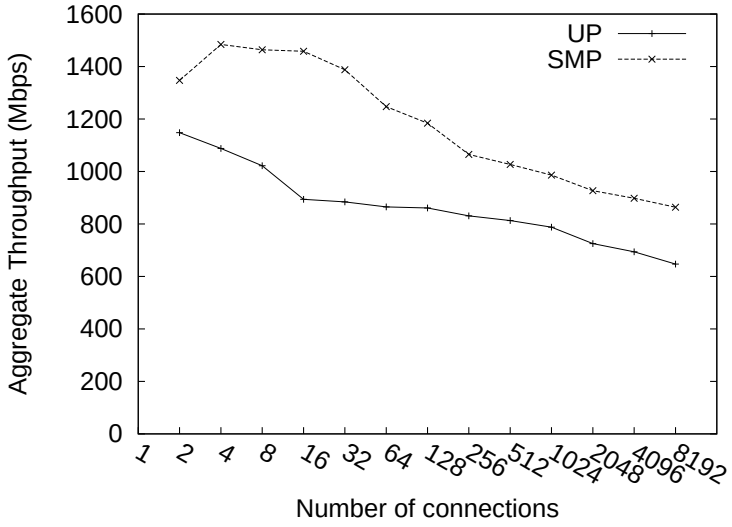

Figure 2.4: Aggregate throughput for default SMP network stack.

2.6.1 Network Stack Characterization

We first characterize the network stack performance using our synthetic benchmark. Figure 2.4 shows the aggregate throughput observed across all connections, as the number of connections are increased for the uniprocessor (UP) and a multiprocessor (SMP) kernel. We observe that, while the UP performance degrades as the number of connections increases, the SMP performance initially increases and then degrades. The initial increase is due to the additional parallelism being exploited by the multiprocessor kernel.

The performance of both systems is much lower than the theoretical maximum. A breakdown of the CPU utilization is shown for the SMP kernel in Figure 2.5. We show the idle time, time spent in the scheduler and synchronization, the network stack, and the user-level processing. We observe that as the number of connections increases, the synchronization and scheduling overhead increases significantly, reducing the time spent in the network stack. This leads to a drop in performance as the number of connections increases.

To further analyze the performance, in Figure 2.6 we show the effects of lock contention. In a multiprocessor system, a contended lock leads to the executing thread being suspended and woken up later when the current owner releases the lock. Therefore, contended locks lead to a higher synchronization, as well as, context switch overhead. To illustrate the effect, we focus on the global connection table lock, which is acquired on every send and receive

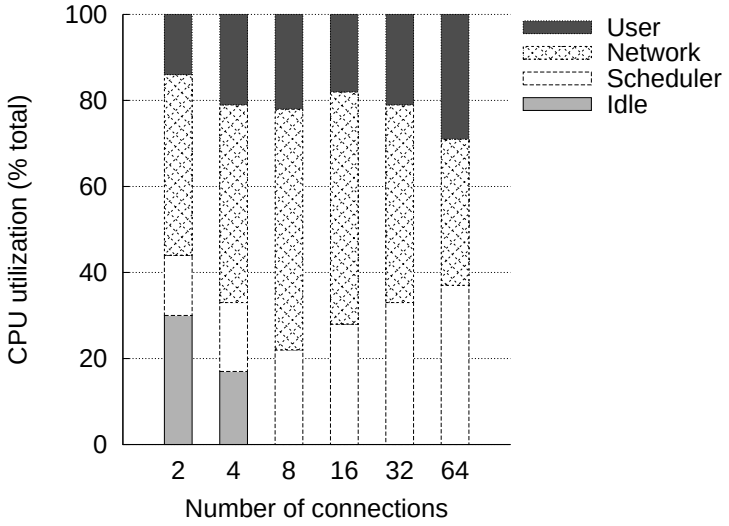

Figure 2.5: CPU utilization breakdown for default network stack.

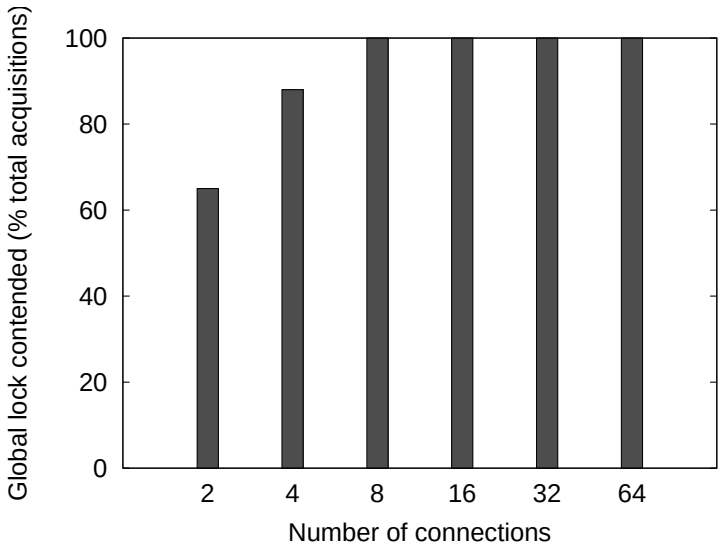

Figure 2.6: Lock contention in default network stack.

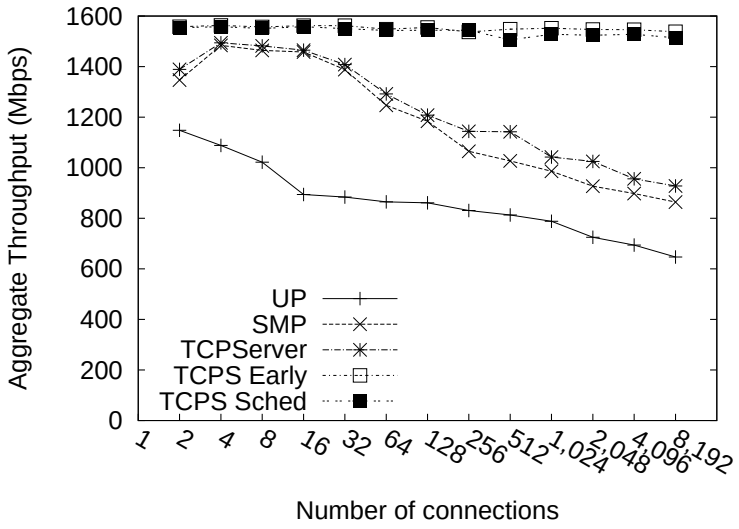

Figure 2.7: Aggregate throughput comparison across network stack variants.

operation to locate the corresponding connection structure from the list of all connections. We see that the contention for the global lock increases and beyond 8 simultaneous connections, the lock is always contended. From the above, we conclude that in order to improve network performance, we must reduce the lock contention as well as scheduler overheads from the network processing.

2.6.2 Performance

Figure 2.7 shows the performance of three variants of TCPServers compared against the default SMP performance. The uniprocessor (UP) performance is included as a baseline reference. We observe that for up to 16 connections, the SMP performance is comparable to the base TCPServers. This variant of TCPServers reduces the scheduler overheads and dedicates one processor in a 2-way SMP server to network processing. However, it still suffers from the higher lock contention overheads, which start dominating at higher connection loads. The TCPS Early and TCPS Sched eliminate the lock contention since they do not access the global connection structure and access only the receive and send queues, which are protected by atomic operations. Therefore, these variants provide a steady throughput even in the presence of a large number of concurrent connections.

The higher throughput of the TCPServers variants is also reflected in the CPU utilization

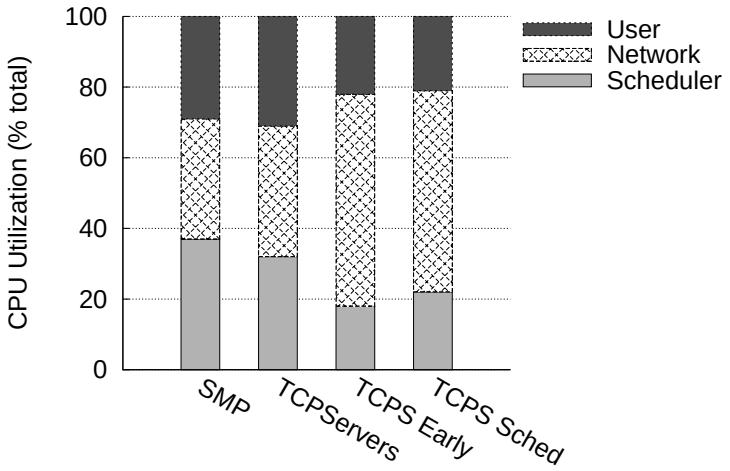

Figure 2.8: CPU utilization breakdown for network stack variants.

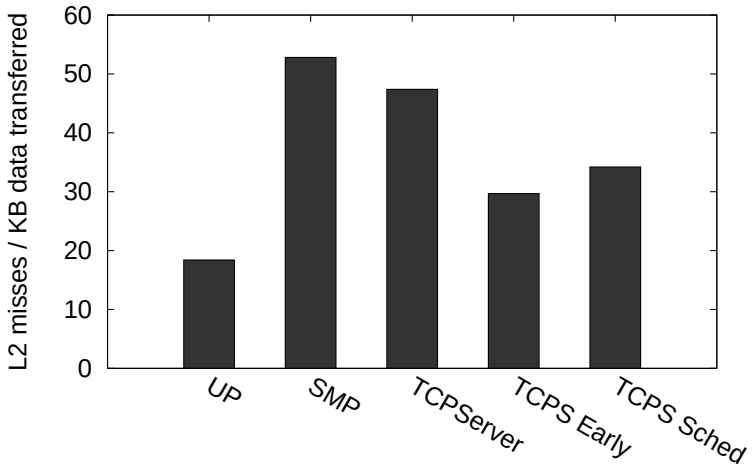

Figure 2.9: Comparison of L2 cache misses across network stack variants.

showing the fraction of CPU used by the scheduler and synchronization, the network stack, and the user-level processing. Figure 2.8 shows the CPU utilization for 256 concurrent connections. Here, the SMP system suffers from high scheduler overheads and therefore the network stack has a smaller fraction and cannot handle the high volume of traffic. In contrast, all variants of TCPServers provide the network stack with a higher fraction of CPU time by dedicating a processor. A lower synchronization overhead further explains the higher performance of the TCPS Early and TCPS Sched.

Finally, to illustrate the effects of L2 cache misses due to data migration, we show the number of L2 cache misses per KB of data transferred for 256 concurrent connections in Figure 2.9. We observe that the the L2 misses for the SMP and TCPServers variants is higher than the TCPS Early and TCPS Sched since there is limited data migration. Among TCPS Early and TCPS Sched, the scheduling data structures lead to a higher L2 cache miss rate and thus a slight performance degradation.

To illustrate the benefit of scheduling network processing at the priority of the destination socket, we first create 8 concurrent connections. We then run a separate process where clients create a number of short-lived connections. These connections are closed as soon as the user-level program receives a completed connection notification. Figure 2.10 shows the aggregate throughput as the number of short-lived connections increases. Since the new connection establishment leads to higher overhead and leads to packets belonging to other (active) connections being dropped, the throughput decreases. In contrast, the TCPS Sched assigns a lower priority to the new connections (SYN), and leads to a lower degradation in the performance.

2.6.3 Web Server Performance

To evaluate the impact the our system on a real-world application, we measure the performance of a web-server using each of the variants of our system. We use the Apache 2.0.48 [14] web server and the httperf [127] benchmark to characterize the performance. Httperf maintains a fixed request rate defined in the configuration and measures the number of completed requests within a specified timeout during a run. We generate a trace where all requests are static, for 32KB web pages. Our goal is to measure the performance improvement due to increased concurrency, therefore, we design a workload mix whose working set fits in the available memory.

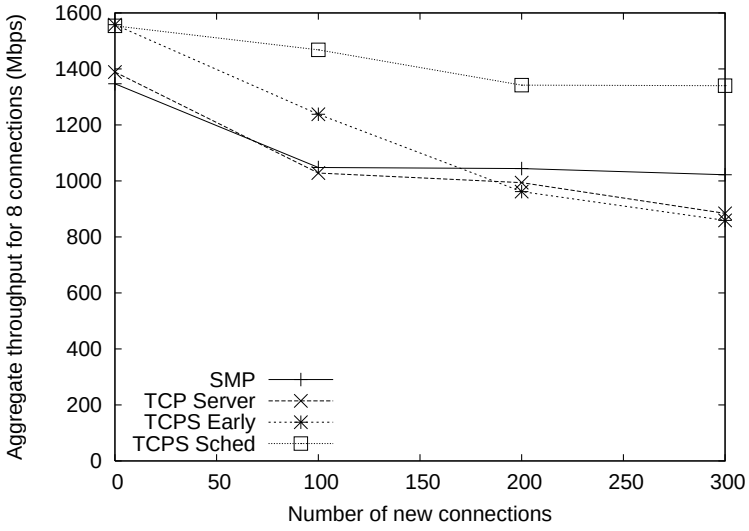

Figure 2.10: Aggregate throughput with short-lived connections

To avoid noise from the disk effects, we discard the results from the first run of our benchmark for all cases.

Figure 2.11 shows the performance of the web server measured by httperf while varying the offered load. The performance measure is the number of successful requests. We observe that for underloaded conditions, all systems demonstrate similar performance. However, as the request load increases, the TCPS Sched and TCPS Early can support a higher request load. Finally, TCPS Sched sustains the high load and shows graceful degradation even when overloaded. The gap between TCPS Early and TCPS Sched when overloaded is due to the large number of new connections being established continuously. While TCPS Early handles all packets at the same priority, TCPS Sched handles the data packets (requests from established sessions) before handling new connections. This allows more HTTP requests to be satisfied, better utilizing the CPU.

2.7 Related Work

Performance of TCP/IP network stack implementations has been a prolific research area. At different times, hardware and software solutions for improving network stack performance have been proposed. We can further classify the research into three categories based on the underlying mechanisms used to overcome network processing bottlenecks - *(i)* memory copy

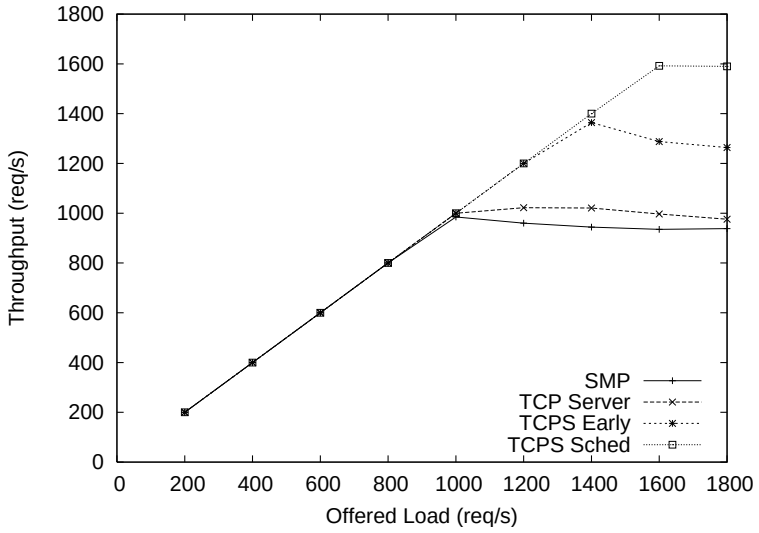

Figure 2.11: Web server throughput with varying load.

	Hardware Approaches	**Software Approaches**
Memory Copy Avoidance	UNet [23], VMMC [64], RDMA [95], IOMMU [162]	IOLite [145, 154], Trapeze [10]
Interrupt Mitigation	Interrupt batching [25]	Polling [125, 187, 118], LRP [62]
Offloading	TCP Offload Engines [10, 5, 25, 219, 76], RDMA [95]	Piglet [128, 192], MARS [161] ETA [162]

Table 2.1: Summary of techniques for improving network stack performance

avoidance, *(ii)* interrupt mitigation, and *(iii)* offloading. Table 2.1 summarizes the past work based on the above classification, and we discuss each of the categories below.

2.7.1 Memory Copy Avoidance

Memory copies have traditionally been the main performance bottleneck in network servers. Copies between user level applications and the kernel network stack requires the CPU to move data byte-by-byte. Therefore, the CPU is busy for long periods of time. Busy network servers, for example, web servers, transfer large amounts of data from stored files over the network. This requires copies from the file system to application memory and the same data is transferred over network sockets.

The inefficiency of the I/O-network interface has been addressed in I/OLite [145], where an integrated buffer management system is proposed and the applications can perform transfers directly from the buffer cache to the sockets. The I/O Lite interface has been adapted by popular Operating Systems e.g. Linux and FreeBSD in form of the `sendfile` [154] system call, where data is transferred between file descriptors (file and socket descriptors). Intelligent use of the sendfile interface along with memory mapped files allows application designers to perform zero-copy sends.

User-level networking [64, 23, 152, 95] proposes to completely bypass the Operating System and provide applications direct access to the network interface. The Operating System is involved in arbitrating the access to the NIC during connection setup, providing a safe interface to the NIC's control data structures, and generating signals for events registered with the NIC. Applications implement network protocols, buffer management, and flow control. Through a combination of virtual memory registration with the NIC and pre-posting receive buffers, user level networking implementations provide zero-copy send and receive interfaces through *virtual memory mapped communication* [64].

User level networking additionally provides asynchronous I/O support through a modified interface, e.g. VIA [67] and Infiniband [95, 59]. Asynchronous I/O allows event driven applications to perform concurrent tasks without blocking in the OS. Event driven applications run an event loop, which manages a state machine for each network connection that is updated on events signalled by the OS. Event driven servers do not suffer from the scheduling and context

switching overheads prevalent in the corresponding multithreaded implementations.

User level networking implementations require hardware support from the NIC, and require application modifications for zero-copy transfers and to use the asynchronous API. Finally, implementations of traditional networking protocols, e.g. TCP/IP and sockets, as user libraries add overheads to each application limit the benefits offered by zero-copy transfers.

2.7.2 Interrupt Mitigation

Interrupt processing imposes significant overheads on busy network servers. Interrupts affect the system performance directly as interrupt processing executes at the highest priority and prevents any other processing in the system. Indirectly, interrupts cause cache misses, TLB shootdowns, and pipeline flushes, which reduce the overall system performance. Finally, interrupt processing executes in the context of the currently executing process, which leads to unfair resource accounting, as the interrupt may be on behalf of the another non-executing process.

Mogul and Ramakrishnan [125] demonstrate that in a loaded system, interrupt processing can easily overwhelm the server CPU leading to *receive livelock*. Interrupt moderation through interrupt batching, polling, and hybrid interrupt and polling architectures have been proposed to mitigate receive livelock in busy systems.

Modern NICs have used on-board memory to store and aggregate packets before generating an interrupt for the CPU to process them. Such aggregation reduces the hardware interrupt handling cost of network processing. However, with increasing network bandwidths such interrupt moderation has limited effect due to high rate of events. Moreover, most of the network processing is performed in the software interrupt context, which still imposes significant overheads.

Polling the network interface to identify network packet arrivals eliminates receive livelock since the interrupt processing is performed inline. However, polling may lead to lost packets if the NIC is not polled frequently enough, or it may lead to CPU resource wastage if the interface is idle. A hybrid interrupt-polling architecture has been proposed where the NIC interrupts the CPU once. This interrupt leads to the system polling the NIC until all packets are handled, or for a fixed period of time when interrupts are re-enabled. Such architecture provides benefits of low latency and CPU usage while preventing receive livelock without additional hardware

support, and is implemented in the Linux networking stack as New API [168] (NAPI) and FreeBSD and MacOS-X as the Interrupt Filter API [15].

Lazy Receiver Processing(LRP) [62] uses early demultiplexing and defers packet processing to execute in the context of the process that owns the connection. LRP integrates network processing into the system's global resource management, schedules network processing at the priority of the receiving process, and discards packets of unresponsive processes early to conserve system resources and limit the effects of unfair resource allocation.

TCPServers provides separation of application and network functionality by dedicating a subset of processors to network processing. Since a dedicated processor performs all receive processing, it must handle receive events at the priority of the destination socket. Once invoked, the receive processing at the dedicated processor runs to completion, therefore the invocation of handlers must roughly follow the process priorities determined by the scheduler.

2.7.3 Network Stack Parallelization

Much of the reseach on parallelizing the network stack took place in the mid-1990s. Two forms of network procesing parallelism were examined : message-oriented and connection-oriented parallelism. In message-oriented parallel organization, any message (packet) can be processed simultaneously on a separate thread without any connection based separation. Therefore, multiple messages for the same connection may be processed concurrently in different threads, potentially improving performance. In connection-oriented parallel the granularity of parallelism is coarse grained, the messages are classified according to connections as early as possible and all messages for a connection are processed in a single thread. Threads may handle multiple connections.

Nahum et. al first examined message-oriented parallelism on an SGI Challenge shared-memory multiprocessor [135]. The study focussed on the locking granularity within the network stack, cost of mutual exclusion on protocol performance, and the opportunity for parallelization in the network stack. The authors conclude that there is limited parallelization opportunity within a single connection, but multiple connection parallelization scales with the number of CPUs. They also demonstrate that cache affinity and lock contention play a significant role in the performance of a parallel network stack. However, the network stack used in the

above was implemented in user-space and used a memory only pseudo network device ignoring the effects of hardware access. Moreover, the TCP/IP protocol implementations, costs of synchronization, OS mechanisms for scheduling and context switching have changed dramatically compared to when the study was performed necessiating a new perspective.

Yates et. al later examined the connection-oriented parallel implementation of the `x-kernel`, also utilizing a pseudo network interface and running on an SGI Challenge architecture [218]. Their study examined the effects of increasing the number of threads with the number of connections and found that a connection-oriented parallel network stack outperforms the message parallel stack. A finding that was further supported by Schmidt and Suda [173] who used modified SunOS and used a real hardware interface. They also concluded that the cost of synchronization significantly affects the efficiency of both message-oriented and connection-oriented stacks.

In modern operating systems, both message-oriented and connection-oriented parallel network stacks have been implemented. Linux and FreeBSD operating systems use a message parallel organization, whereas DragonflyBSD and Solaris implement a connection-oriented parallel stack. Willmann et. al. perform a comparative study of message and connection-oriented parallelism on modern hardware and conclude that, while the connection oriented parallelism offers benefits over message oriented parallelism, the scheduling overheads significantly reduce the efficiency [215].

In this dissertation, we focus on the message-parallel organization of the network stack in context of offloading its functionality to a set of processors in a multiprocessor system. This organization improves cache locality and reduces lock contention, which are identified as two important sources of inefficiency in the above.

2.7.4 Offloading

Network processing performs several CPU intensive tasks, e.g., data copies, checksumming, TCP segmentation, etc. These tasks are independent of any OS specific state and can be performed independent of other network processing. Moreover, offloading these tasks does not require significant changes to existing network stack implementations and do not significantly increase the processing or memory requirements of the network interfaces.

TCP/IP protocol specifications require a one's complement Cyclic Redundancy Check (CRC) checksum to be performed independently over the IP header and the payload including the TCP header [151]. This checksum can efficiently be performed at the NIC and is offloaded yielding significant savings in the packet processing overheads. Most modern NICs support checksum offload. Recently, other tasks e.g. TCP segmentation offload has been introduced, which allows the OS to DMA large chunks of data to the NIC. The network interface then segments the payload into Maximum Transmission Unit (MTU) sized chunks and sends out the packets. Such offloading support is largely stateless and requires little OS support.

TCP/IP offload engines (TOEs) propose a more aggressive offloading strategy and offload stateful tasks, e.g., TCP/IP state maintenance, IP fragment reassembly, TCP reordering, global connection variable maintenance, etc., to the NIC. Such offloading is complex and requires significant CPU and memory resources at the NIC. In an ideal scenario with a few connections and low latency, such offloading eliminates almost all network processing overheads from the host CPU. Unfortunately, unlike the host, the memory and CPU capacities at the NICs grow at a much slower rate, are more expensive, and are limited by power resources at the NIC. A hybrid approach to offloading, where the connection offload is controlled and managed by the OS, has been proposed recently [99, 76]. These systems target the poorly scaling operations like I/O bus crossings, cache misses, and interrupt processing. While such approaches demonstrate a significant performance gain for a reasonable number of connections, the system reverts to the traditional network stack with the associated overheads for larger connection rates. For busy network servers with a large number of short-lived connections, such offload does not offer significant benefit.

To overcome the resource limitations of NIC based offloading as well as to simplify development, evolution, and maintenance of networking code, dedicating a subset of processors to network processing has been proposed by us [159] as well as other researchers [128, 161, 162]. Such an architecture eliminates the limitations of memory and CPU cycles, improves cache and TLB locality, allows efficient polling of the NIC, and retains the simplicity of implementation and maintenance.

Muir et. al. propose Piglet, where the device driver functionality is offloaded to the dedicated processors. Piglet enables the system to poll the NIC, but majority of the TCP/IP functionality is still executed in the application processors. In Piglet, the dedicated processors are determined statically and the system always performs polling of the NIC. In contrast, TCPServers enables a reconfigurable dedicated processor set, and uses both interrupts and polling to handle NIC events.

Embedded Transport Acceleration (ETA) [162] dedicates one or more processors or hardware threads to perform all network processing. These dedicated Packet Processing Engines (PPEs) avoid context switches and cache conflicts with application processes, and interrupt overheads by polling the NIC. These PPEs can be extended with additional DMA engines, which support memory to memory transfers without involving the CPU. These extensions free the CPU of the memory copying task and further improve cache locality. Memory Aware Reference Stack (MARS) uses PPEs and the hardware extensions for memory copying and header separation [161, 162].

Brecht et. al. have recently proposed an extension of the ETA architecture introduces a Direct Transport Interface (DTI), which uses shared memory between the OS and the applications along with an asynchronous I/O API to enable the PPEs to poll for application generated events [36]. This system executes the entire network stack including transmit and receive processing in a dedicated processor. However, both for ETA and for the system reported by Brecht et. al. [36], the set of hardware contexts dedicated as PPEs is fixed and they use an ad-hoc scheduling strategy for NIC polling and application event handling.

TCPServers shares the design where a subset of hardware contexts are dedicated to network processing. However, TCPServers monitors the system load and the rate of NIC and application events to schedule individual event handlers and reconfigure the set of PPEs. TCPServers also separates the transmit path and the receive path of network processing. In the transmit path, TCPServers only handles packet scheduling for the NIC. This does not eliminate the memory copy and system call costs for the transmit path, however, the zero-copy send interfaces and low-overhead of system calls does not warrant the complexity of a maintaining a shared memory interface and application modification. Additionally, we expect similar benefits from the hardware extensions for memory-to-memory copies.

2.8 Summary

Network stack performance is critical and with the increasing multiprocessor availability, needs to be revisited in the context of parallelization as well as separation of application processing and network processing. In this chapter, we presented TCPServers, a system architecture that takes advantage of processing contexts available in multiprocessor systems to offload network processing to them. Such offloading enables low-overhead shared memory communication between the host and the offloaded context while retaining the isolation provided by the TOEs. We developed mechanisms for early demultiplexing and scheduling network processing at the connection priority using per-socket OS data structures: Receive Queues. We implemented TCPServers and demonstrated through our evaluation that TCPServers architecture improves the performance of our benchmark program by more than 75%, compared to a modern multiprocessor network stack.

Chapter 3

Orion: Offloading Monitoring for Improved Availability

3.1 Problem Statement

Self-healing and recoverability from events that impair the functionality of a computer system have become more and more the focus of systems research [146, 157, 158]. This trend reflects a shift from raw performance towards intelligent self-manageable computer systems. To realize the goal of self-management, the system must minimally perform two tasks, *(i)* gather monitoring data and analyze it to identify anomalies, and *(ii)* to take *healing actions* to repair or recover the system.

Hardware sensors, for example, CPU temperature sensors, fan speed monitors, disk activity sensors, etc. have been used to offload *hardware* monitoring tasks that enable a system to adapt its functionality. SMART [193], ACPI [90] and IPMI [55] have introduced the ability to offload fine-grained hardware monitoring and raising events on exceptional operating conditions. System software has incorporated support to react to these alerts for intelligent resource management and protection of the system against hardware failures, e.g., reducing the CPU frequency when the temperature is high, spinning down disks that have a faulty disk arm in a disk array, switching off idle nodes in a cluster, etc. Unfortunately, similar functionality for continuous software monitoring is limited in existing computer systems.

Traditionally, software monitoring has been performed *within* the system, by executing a special monitoring task or a daemon, which monitors *events* generated by applications and the OS. Events represent the behavior of the system and are generated internally, e.g., through logging, or externally, by interposition on the interface through which the monitored software component interacts with the rest of the system. Monitoring using internal events relies on the software components to generate detailed logs or statistics. Software monitoring using external events has incomplete knowledge and only approximates the state of the monitored component

using side-effects, e.g. statistics, system call logs, etc. Most software monitoring systems use a combination of internal and external events to identify anomalous behavior. External monitoring has broader coverage, can monitor overall system state, and does not require application modification. Moreover, internal events can be *externalized* using explicit calls to external monitoring interfaces. Therefore, in this chapter, we focus on continuous monitoring using external events.

There are several challenges in continuous OS and application monitoring. First, statistics that represent system state are updated frequently and these updates do not generate explicit events, making interposition difficult. Second, events generated in the OS are scattered across memory in various subsystems and applications. Traditional administration utilities, e.g. `vmstat, top, ps,` etc., are inefficient, as they gather information through multiple system calls. Using these utilities for continuous monitoring by periodic invocations to generate events has prohibitively high overheads. Moreover, using the system to monitor itself or "monitoring from within" cannot be used to detect and diagnose system-wide failures which render the OS unavailable, e.g., depletion of system resources, system hang failures, crashes or deadlocks. Finally, monitoring from within cannot reliably detect intrusions and malicious behavior, since the monitoring task itself may be compromised.

In this chapter, we present Orion, a system architecture based on offloading continuous monitoring and repair of OS and application state to external hardware. Offloading monitoring and repair functionality has the following requirements. First, the external hardware must be able to access the target memory to retrieve in-memory state for monitoring and modify it for repair. Second, the external hardware must be programmable, for the administrators to define and modify monitoring policies. Third, monitoring from the external hardware must not directly involve the monitoring target, and the target must not be able to disable, modify, or adversely affect monitoring functionality. Finally, monitoring must be nonintrusive, that is, it must not impose significant overheads at the target, which could change the target behavior.

Orion offloads monitoring and repair functionality to a programmable network interface. This network interface sits on the I/O bus of the target system, has its own processor and memory, can access the target memory without using host CPUs and can be accessed over the network for cooperative external monitoring. Orion enables continuous monitoring of in-memory

state without imposing any CPU overheads and without interfering with tasks executing on the system. We demonstrate mechanisms for external monitoring over Orion and show that we can continuously monitor OS and application state, quickly and reliably detect failures, and repair inconsistent or incorrect OS state without imposing significant overheads.

3.2 Operating System and Application Monitoring

In this section, we identify requirements for Operating System and application monitoring and define the usage models that motivate the design of our monitoring architecture. We identify three primary tasks that must be supported by Orion: *(i)* Failure Detection, for accurate and low-latency identification of failures, *(ii)* Failure Diagnosis, for monitoring data structure invariants to identify the *root cause* of failures, and *(iii)* Failure Prevention, for performing repair of damaged or inconsistent OS and application state.

3.2.1 Failure Detection

Network services today are hosted at large data centers, which deploy hundreds of thousands of computer systems to handle high performance and availability requirements [81, 6, 4]. At such large scales, hardware as well as software component failures are routine, and it is critical to identify a failed component, quickly, and with high accuracy. Moreover, monitoring these performance critical services must not impose high overheads and require human involvement, which is prohibitively expensive and slow to be practical.

Failure detection is a crucial part of system maintenance as it generates alerts for failed components, which must be replaced to maintain continuous service, and it allows the system to adapt and reorganize itself through load balancing and failover. Failure detection must be *(i)* low latency, to maintain uninterrupted service to clients, *(ii)* accurate, with low false positives, i.e., a component is active even when the system declares it as failed, and with low false negatives, i.e., the system fails to identify a failed component, and *(iii)* low overhead, to minimize interference with the executing applications.

Orion implements *Remote Monitoring*, which uses Memory-to-Memory Communication without involving the the CPU or OS of the target system to offload monitoring functionality to

a cooperating monitor. Remote monitoring over low-latency interconnects enables fast detection without imposing additional overheads at the target. Since Orion does not involve a local software agent on the target, the probability of false positives is low. Moreover, Orion is programmable and can be used to implement timed-perfect failure detection protocols [73], which guarantee that a node that is declared by the monitor to have failed does not cause inconsistent behavior. Finally, since the target cannot control and is unaware of the existence of an Orion monitor, monitoring functionality cannot be disabled and is always available, even when the OS is crashed, compromised, or is otherwise unavailable.

3.2.2 Failure Diagnosis

Identifying the root cause of a failure is as important as failure detection itself. Treating failures as isolated instances and ignoring the underlying cause squanders opportunities for *preventive maintenance*, cost savings, and reduced future downtimes. Several studies have shown that software failures are much more prevalent than hardware failures [201, 196, 45]. Careful analysis of software failures through offline inspection of the failed software state (core dumps) enables developers to identify and fix problems in their code, while online inspection of live state enables prevention of impending failure through repair of inconsistent state.

Failure diagnosis can be performed proactively, during development through code inspection and software verification tools, and reactively, through continuous online monitoring. Large codebases and complexity introduced due to implicit assumptions about programming invariants makes it difficult to identify and handle all failure conditions during development. Automated tools for software verification are limited in coverage and are low performance, and are hence impractical for analysis of large systems. Therefore, the only alternative to identify and diagnose failures in such systems is online monitoring and reacting to detected failures. For example, when designing a multithreaded OS, detecting a cyclic acquisition of locks is possible only by continuously monitoring the system during normal operation [19] despite efforts of hundreds of skilled developers and analysis tools [56]. Unfortunately, such monitoring functionality is intrusive and imposes significant overheads. Therefore, it is common practice to disable monitoring functionality in production environments. This leads to monitoring being limited to controlled test environments and and imprecise diagnosis of failure conditions when

deployed in a production system.

To support fine-grained diagnosis and to maintain history, remote monitoring over Orion goes beyond simple heartbeat mechanisms and provides an OS abstraction, *Sensor Box*, which stores monitoring data generated by the OS and applications. Orion enables direct access to remote memory even when the OS may be hung, deadlocked, or be otherwise unavailable. Programmability of the Orion NIC enables fine-grained inspection of *live* data structures in the target memory. Finally, Orion defines a query interface, which supports defining predicates that operate on the monitoring data to detect anomalous behavior during normal operation and can be composed to implement complex policies for automated failure diagnosis.

3.2.3 Failure Prevention

To enable continuous service, anomalous system states must be identified and if possible *repaired* before the system is rendered unusable. Repairing incorrect or inconsistent state allows the system to survive hard to reproduce, unpredictable failures, which would otherwise lead to unavailability.

Software rejuvenation or restarts have previously been proposed to repair damaged state [146]. However, such mechanisms are disruptive as they always cause a restart, even when the inconsistency is localized to a small, easily repairable, portion of the state. For a network server, repair through restarts is no better than an actual failure where clients lose service for the duration of the restart. Recently, small changes to the execution environment and device driver interface have been used to avoid failures and in some cases recover from them [158, 201].

Orion supports failure prevention through remote repair, where damaged OS state is corrected through remote intervention. We regard the state of an OS as damaged when a certain OS subsystem is impaired and cannot perform its normal function. Damage to the OS state can be caused by system misconfiguration, malicious attacks, bugs triggered under heavy load, resource exhaustion, etc. Orion repairs damaged OS state through specialized handlers, called *Repair Hooks*, which replace the damaged state with correct state generated on an external monitor.

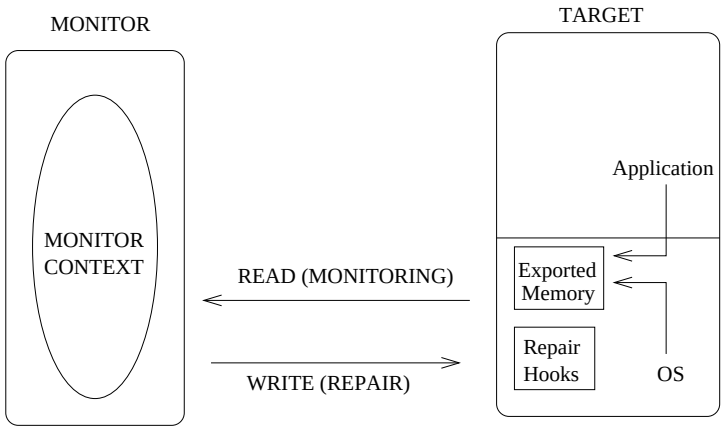

Figure 3.1: Orion Architecture

3.3 Orion Architecture

In this section, we present Orion, a system architecture that enables near real-time monitoring of OS and application events when offloaded to a programmable network interface (I-NIC) or a remote node in a collocated cluster.

Figure 3.1 shows the Orion architecture. It is defined in context of a Monitor-Target pair. The target exports regions of memory where events of interest are stored. These regions may include statistics data structures, in-memory system log, process stack buffers, etc. The target also defines specialized handlers, called Repair Hooks (RH), which allow modification of system behavior through memory modifications. Examples of RHs include dynamic configuration variables (`sysctl`), process signal masks, kernel resident processes that execute when memory locations are modified, etc.

The Monitor in Figure 3.1 is the external hardware that executes the monitoring and repair functionality in a Monitor Context. It has its own processor, memory, and communicates with the Target over a fast communication medium. It reads the exported memory to retrieve monitoring data, which is used to identify anomalous conditions and maintain a historical log. On detecting an anomaly, the monitoring context writes to the RHs exported at the target to perform repair.

In the Orion monitoring framework, the monitoring context can be instantiated on the target host as a process or a daemon, on a programmable network interface as a network interface context, and on a remote node connected over an interconnect that provides direct access to

Configuration	Communication Channel	Context	Host
Local	Shared Memory	Thread or Process	Target
NIC	DMA over PCI bus	NIC Context	Target
Remote	RDMA	Thread or Process	Monitor

Table 3.1: Orion monitoring configurations

target memory. Characteristics of these configurations are summarized in Table 3.1 and we discuss each of them below.

Local: The local configuration is the baseline configuration for Orion, where the monitor is instantiated at the target system. Monitoring is performed in a thread or process context, which uses the host CPU for execution. In this configuration, the exported memory is instantiated using shared memory established between the monitoring context and the monitored entity.

Shared memory is the fastest communication channel, therefore the access to the exported memory is almost instantaneous. Unfortunately, fast access is not enough since the monitoring thread relies on the host CPU for execution and competes with other threads on the same machine. Moreover, monitoring from within the system cannot report failures since it cannot execute on an already impaired system.

NIC: In the NIC configuration, the monitor is instantiated on a programmable Intelligent NIC (I-NIC) that sits on the PCI bus of the target host and has its own processor and memory. The monitoring context is the I-NIC specific execution context. Communication is performed over the PCI bus using I-NIC initiated DMA. A one-time registration of the exported memory with the target context generates a memory descriptor used by the monitor to fetch the SB memory through DMA.

The monitoring context in the NIC configuration can access the exported memory through DMA. This access does not involve a network transfer, is reliable, and does not require additional monitoring systems. However, the NIC is limited in its resources, is difficult to program, and monitoring logs are lost on each restart and cannot be used for future inspection as it does not have access to stable storage.

Remote: The Remote configuration enables the monitor to be instantiated on a remote node connected over a private network. The monitoring context is a thread or process that executes on the remote node.

Remote monitoring relies on a mechanism that enables direct remote access to the target memory. Serial interconnects, including USB and firewire provide such access. However, these interconnects are point-to-point cannot communicate with other hosts in the network. Therefore, Orion uses a low-latency high-bandwidth network interconnect with Remote DMA (RDMA) capabilities, e.g. Myrinet or Infiniband, to offload monitoring functionality. RDMA allows direct data transfers between the target and monitor system memory without involving the target host processors. A one-time registration of the exported memory with the remote context generates a remote memory descriptor, used by the monitor to initiate remote reads to fetch the memory contents.

Since the monitoring context executes on a remote system, it has more resources than the NIC configuration, and does not rely on the host CPU like the local configuration. Finally, development and maintenance of the monitoring system is much easier than with the I-NIC. This configuration is the most powerful as near real-time monitoring and logging to stable storage can be supported simultaneously.

3.4 Orion Design

Figure 3.2 shows an overview of the Orion Architecture for offloading OS and application monitoring functionality. The target exports memory through one or more *Sensor Boxes (SBs)*, which store the monitoring data generated by applications and the OS using the Sensor Box API. The monitor defines one or more monitoring contexts, which are associated with a Sensor Box at the target and retrieve it periodically over the fast communication channel to create a local SB View. The Orion Query Interface is used to program the monitoring contexts to define monitoring policies. This interface can also be used for on demand querying of the monitoring context state.

Orion is a monitoring framework based on functionality offloading. We use efficiency and generality as the two first order considerations in designing the Orion framework. Efficiency of

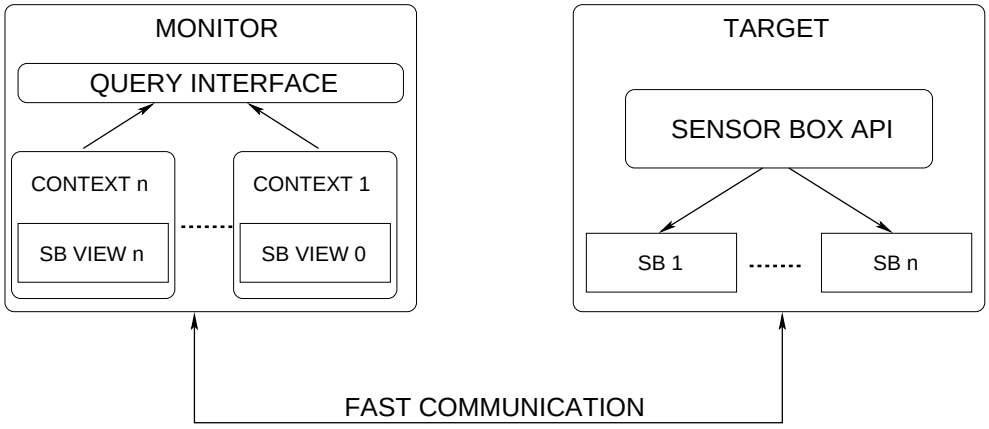

Figure 3.2: Orion Model.

the offloading mechanism is important as it determines the latency of detecting anomalies, and determines the complexity of the monitoring functionality that can be supported in different configurations. To make the monitoring framework application independent, we do not rely on specific properties of the monitoring target. Instead, the Sensor Box API is designed to allow easy adaptation and extension for diverse applications and OS subsystems.

Figure 3.3 shows details of the Orion monitoring architecture. The Monitor and Target Controllers create a binding between the monitoring entity and the target system. The monitoring entity can be instantiated on the same host, at an I-NIC, or on a remote host depending on the configuration. The control channel is used to maintain and exchange information about the two primary components of the Orion Architecture: *(i)* Sensor Box, and *(ii)* Monitoring Context, which are discussed in more detail below.

3.4.1 Sensor Box

A *Sensor Box (SB)* is a structured collection of records called *sensors*, which are allocated in the OS memory of the target system. Each update of a sensor adds a row to the sensor box. Therefore, an SB can be thought of as an infinite *event stream*, whose contents are revealed one row at a time. Sensor Boxes are created at the target system and are updated either by instrumenting code to record updates, or by using existing statistics data structures exported by the system.

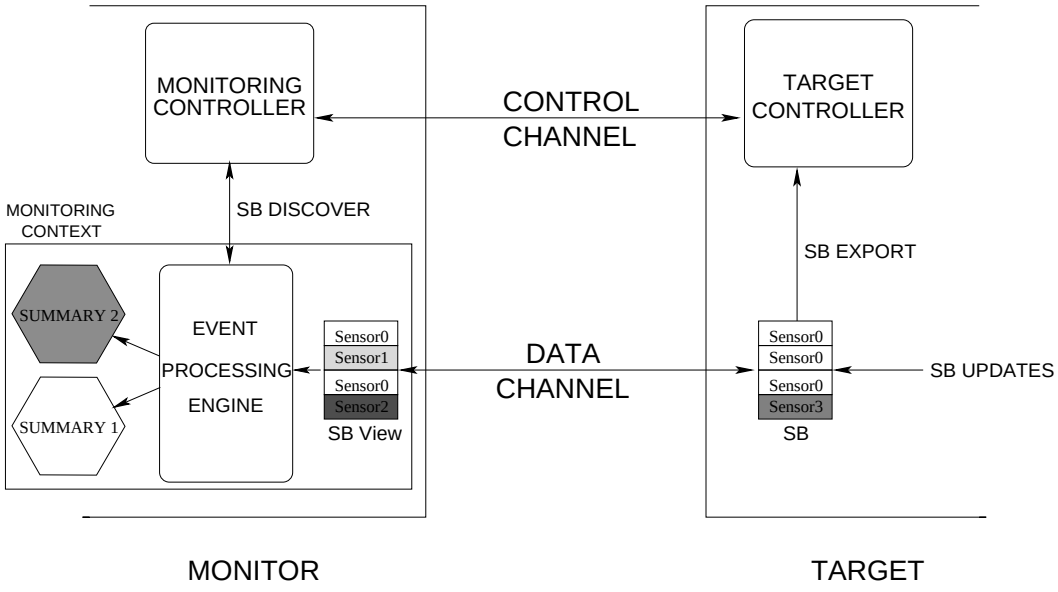

Figure 3.3: Orion Monitoring Architecture

The monitor retrieves the SB contents to create a local view that reflects the state of the monitored system. Each Sensor Box is assigned an identifier and a name, which are used by the monitoring thread to associate the appropriate Sensor Handler (SH) to handle events. Each monitoring task, for example failure detection, defines a new SB and SH pair. The SB is instantiated on the Target, while the SH is executed on the monitor.

Sensors

A sensor is represented by the tuple, $< ID, \mathcal{T}, V >$, where ID is a unique event identifier, $\mathcal{T}$ is a timestamp, and V is the sensor specific event record. A monitored entity (e.g., OS subsystem) registers the sensor with an SB that assigns it an identifier, ID. Updates to the sensor value, V, are performed through the SB interface. Each sensor defines V, which can be a scalar, a structure with multiple fields, or a reference to a memory location.

The SB records the timestamp, $\mathcal{T}$, of the update and stores the sensor tuple in the SB memory. There can be multiple tuples with the same identifier in an SB indicating multiple updates to the same sensor. Once stored by the SB, the sensor tuples cannot be modified at the target. The monitor flushes the SB contents when it has completed processing the tuples.

ID	Timestamp	Value
VM_METER	0	&(vmmeter)
$TIMER\_INTR$	1	100
$OOM\_RUNNING$	1.4	1
$FORK\_REC$	1.8	(5120, 5124)
$TIMER\_INTR$	2	100

Figure 3.4: Sensor Box example

API Routine	Input	Output	Purpose
`sbcreate`	Name, SB params.	SB, SB ID	Create a new SB
`sbdestroy`	SB ID	-	Destroy existing SB
`sbexport`	SB ID	Memory descriptor	Export SB memory
`addsensor`	SB ID	Sensor ID	Add sensor
`rmsensor`	SB ID, Sensor ID	-	Remove sensor
`updatesensor`	SB ID, Value	Status	Update sensor value

Table 3.2: Summary of Sensor Box API routines executed at the Target

Figure 3.4 shows a Sensor Box with four sensors. The $TIMER\_INTR$ sensor counts the number of timer interrupts handled by the system and is updated every second. The $FORK\_REC$ sensor records the parent and child process identifiers on each invocation of the `fork()` system call. The VM_METER sensor maintains a reference to the FreeBSD `vmmeter` data structure, which records statistics related to the Virtual Memory subsystem. Finally, the $OOM\_RUNNING$ sensor records a flag whenever the system runs out of memory and invokes a memory reclaim thread, which kills processes to free up system memory.

The monitored entity and the Sensor Handlers are tightly coupled. The Sensor Handlers must be aware of the semantics of the sensor contents recorded in the SB. The binding between the Sensors and Sensor Handlers imposes a contract between the monitor and the target. At the target system, the monitored entities must record the values, V, following the semantics expected by the associated SH. The monitor must retrieve the SB from the target and invoke the Sensor Handler to interpret the contents.

```
sbcreate(char *name, sbparams_t *sbparams);
sbdestroy(int sbid);
sbexport(int sbid);
addsensor(int sbid);
rmsensor(int sbid, int sensorid);
updatesensor(int sbid, int sensorid, void *value, size_t len);
sbdiscover(char *sbname, sbparams_t *sbparams);
sbassociate(int sbid, sh_t *sensorhandler);
sbfetch(int sbid, memdesc_t md, void *sbview, size_t len);
sbflush(memdesc_t md);
```

Figure 3.5: Sensor Box API

API Routine	**Input**	**Output**	**Purpose**
`sbdiscover`	Name	SB ID, SB params.	Find exported SB
`sbassociate`	SB ID, SH	Status	Set SH for SB
`sbfetch`	SB ID, Memory descriptor	SB Contents	Fetch SB contents
`sbflush`	Memory descriptor	-	Clear remote SB contents

Table 3.3: Summary of Sensor Box API routines executed at the Monitor

Sensor Box API

The Sensor Box API, shown in Figure 3.5, enforces the contract between the monitor and the target and provides an interface to access and update Sensor Boxes. Tables 3.2 and 3.3 summarize the Sensor Box API routines executed at the target and the monitor respectively. They describe the purpose of each routine and its input and output parameters.

At the target, the Sensor Box is created during initialization using the `sbcreate` and is exported for the monitor using the `sbexport` API calls. These calls return an SB identifier and a memory descriptor, used in all future calls. The memory descriptor is a cookie, which is used by an external monitor to gain access to the target memory. For a local configuration, it is the shared memory handle, for the I-NIC configuration, it is a DMA descriptor, and it is an RDMA descriptor in the Remote configuration. The `sbexport` call additionally registers the SB name with the target controller. The SB is destroyed using the `sbdestroy` call. To update the sensor box, the applications and the OS subsystems use the `addsensor`, `updatesensor`, and `rmsensor` to add, update, and remove sensors from an SB respectively.

The monitor identifies the Sensor Box of interest using a name through the `sbdiscover` call. This call is performed over the Control Channel and the names registered during initialization through the `sbexport` calls are provided to the monitor as its response. The SB discovery also provides the SB parameters, which include update frequency, number of sensor records, and the memory descriptors. At the end of this call, the monitor can start fetching the SB using `sbfetch` and clearing the remote SB using `sbflush` API calls. `sbassociate` is used to map handlers to sensors. When the monitor context identifies a sensor update to an associated handler, it makes an asynchronous function call to the handler.

Sensor Box Views

A Sensor Box View is a copy of the SB contents, which can be modified by the monitor. This view is used by the sensor handlers to process events recorded in the SB. For a local monitor, a Sensor Box View is a reference to the SB contents and does not require any copies. When offloading to an I-NIC, the SB View is retrieved through DMA. The SB at the target system is exported and a DMA mapping is established between the NIC and the host. This is a one-time

registration operation following which, the I-NIC initiates the transfers from host memory to the NIC memory to obtain the view.

Remote Orion configurations rely on the RDMA capability of the private network. To export an SB over RDMA, a one-time registration operation is performed at the target host through the `sbexport` API routine. The resulting memory descriptor is passed to the remote monitor, which then uses this descriptor to perform non-intrusive remote reads on the SB memory to retrieve the SB views.

Sensor Box Updates

Orion does not rely on the target OS or CPU for *analyzing* events when such processing is offloaded to the NIC or to a remote node. However, the SB contents must be updated at the target. These updates can be performed inline by instrumenting the code paths being monitored. Today, several OSes support tracing code paths through instrumentation [53]. Systems like DTrace [42] provide pervasive instrumentation that records events in system memory. If available, a reference to these buffers can be used to construct SBs for monitoring fine-grained events. Orion also supports offloading monitoring functionality *without* code instrumentation, using only existing data structures that record statistics. An SB is created using a reference to these data structures and the contents are exported to the monitor. The monitor first retrieves the SB, identifies the references to data structures and initiates further `sbfetch` calls to get the sensor values.

When SBs are defined using existing data structures without code instrumentation, the SB is not updated when the contents are modified. Therefore, updates to such sensors are performed either *explicitly*, in a thread executing on the target system, or *implicitly*, when the SB contents are retrieved by the monitor.

While explicit updates in a thread can record fine-grained events, executing a thread on the target may not always be possible, e.g. on a busy or a failed system. Moreover, even when executing locally, all updates cannot be recorded reliably, for example interrupt statistics may be updated multiple times in the interrupt context.

On the other hand, implicit updates provide an aggregated view of events. These updates do not require any involvement of the host OS in monitoring. In fact, the CPUs of the target

system are not used for monitoring. Unfortunately, implicit updates cannot be used to identify event sequences, e.g. for performance debugging or data structure monitoring. However, for monitoring applications that rely on aggregate statistics, e.g. failure detection, implicit updates are sufficient.

3.4.2 Monitoring Context

The Monitoring Context represents a unit of execution at the monitor. It is a thread context for the Local monitoring, is a task control block in the NIC configuration, and a thread context executing on the monitor for Remote monitoring. In each of the above, the Monitoring Context performs three primary tasks: *(i)* Construct the SB View at the monitor by fetching the Sensor Box from the target, *(ii)* Classify events and invoke the associated Sensor Handler, which parses the event records to generate composite events or to log the events, and *(iii)* Provide a Query Interface, which enables administrators to specify interesting events or a group of events and generate alerts when these events occur.

Event Processing Engine

The Event Processing Engine (EPE) initiates the transfer of the SB from the target node to construct an SB View. During initialization, the EPE discovers the memory descriptor of the exported Sensor Box associated with its context. It also handles the asynchronous events, for example, network disconnection events, feedback rate control for fetching the SB, etc., which are generated by the Monitoring Controller.

The EPE schedules the SB transfers periodically. The initial frequency of SB View refresh is determined by the SB parameters specified at the target. During operation, the EPE adapts this frequency based on the feedback from the Monitoring Controller and the Query Interface. This operation determines the overheads as well as the latency of detecting anomalous events. Clearly, the higher the frequency of transfers, the higher is the overhead. Therefore, the SB View refresh is delayed for events that do not require frequent updates. For example, for fast failure detection, the SB View must be refreshed as fast as possible, whereas for periodic events, e.g. to find the average CPU load, scheduling the SB transfer every second is enough.

Event Summaries

Event Summaries are the data structures stored by the Monitoring Context to represent the history and the current state of the target system. These data structures are used by the Query Interface to respond to user defined queries or to generate alerts for automated response systems.

Efficient and detailed event summaries are crucial to Orion. Limited memory and CPU overhead for updating and querying the summaries is essential to successfully offload monitoring functionality, while maintaining a reasonable response time. At the same time, it is equally important to store enough information to support queries of interest with high accuracy. These constraints make the choice of the data structure used to store summaries crucial.

We define three primary data structures used in Orion to maintain event histories, *(i)* Counters, *(ii)* Counter Aggregates, and *(iii)* Dependency Graphs.

Counters Counters are the basic data structures for maintaining event statistics. The counter values are scalars and this data structure maintains an exact count for each event identifier. Counters are useful when the number of event sources is low. The primary application of counters is in failure detection, where continuous updates to a small number of health indicators, called progress sensors, are used to determine whether the system is alive. For example, if an OS is alive, it must periodically receive the timer interrupt and update its timer variables. A counter representing the number of timer interrupts and the current timer value at the target is sufficient to determine its health. The monitoring system must maintain a list of these counters for a window of time to identify lack of activity on the target.

Unfortunately, counters are unsuitable for finer grained monitoring due to two main reasons. First, the memory required to store the counters increases linearly with the number of counters. Second, counters do not maintain causality information, and are therefore not useful to track dependencies and the order of events.

Counter Aggregates While raw counters provide a snapshot of the system state, storing a long historical record and accessing this log for each query is computationally expensive. Most often, a monitoring system uses aggregates, e.g. average, median, rate of change

of values, etc. to identify anomalous behavior. Moreover, the monitor may be interested in the evolution of aggregates rather than simple counter values, e.g. interrupt rates vs. interrupt counts. Aggregates are generated at the monitor using the raw contents of the SB views.

Randomized data structures, e.g. counting Bloom filters and randomized sketches have been previously used to generate and maintain efficient counter aggregates. These data structures require small amounts of space and provide probabilistic accuracy guarantees for query results. Orion provides implementations of Bloom filters and Count-Min sketches to support efficient storage of counter aggregates.

Dependency Graphs For sensor values which are not scalars, but represent an event identifier, the monitoring system is interested in relationships between them. For example, to protect a system from a fork bomb, a process that recursively creates new processes without doing any useful work, the original process and all its descendents must be terminated. The $FORK\_REC$ sensor, which records the parent and child process in each invocation of the `fork` system call, can be used at the monitor to generate a *shadow* data structure that mirrors the process hierarchy at the target. The ability to construct this shadow process hierarchy at the monitor allows Orion to identify and terminate the parent and all its descendants. Other examples where dependency graphs are useful are lock order verification, intrusion detection, etc.

Dependency graphs are used to represent event dependencies and enable continuous queries that verify a pre-specified order of events. Each node in the data structure is an event identified by the event identifier. An edge $ID_0 \rightarrow ID_1$ in the graph records that event ID_0 happened before ID_1. The correlations between two events can be specified a-priori, in the example above, a `fork()` system call generates a dependency between the parent and child process identifiers, or may be inferred offline by analyzing historical records of all events.

Orion constructs dependency graphs using a sequence of SB Views at the monitor. The timestamps in every sensor record impose a causal ordering on all events and are used to construct a sequence of events. This sequence of events can then be used to enforce

ordering constraints on operations at the target.

3.4.3 Orion Query Interface

The goal of any monitoring system is to support user queries and generate alerts on detecting events of interest. The Query Interface uses the event summaries generated by the Monitoring Context to evaluate queries. It may also access the raw event data through the EPE to maintain a log on stable storage or a window of events in memory.

In the Orion model, the SB is updated frequently. Multiple passes over the SB View causes a long delay between updates, which may lead to lost events at the target. Therefore, we keep the SB View in memory for exactly one update interval and the Sensor Handlers and the Query Interface can scan a sensor tuple exactly once. This model is similar to the *Data Stream Model* [18], where relations are not stored in persistent tables, but arrive as a transient stream of updates.

Orion supports only *pre-defined* queries. These queries are defined *before* any data required to satisfy the query is received. These pre-defined queries can be invoked multiple times on-demand during operation, or can be continuously evaluated by the system. The sensors can be accessed exactly once, and Orion maintains only *summaries* of the events, e.g. average value of a sensor, and not the exact event data. Therefore, a new query that refers to past events cannot be answered. Moreover, a set of queries defined at the monitor can be verified during initialization and any updates to this set may lead to inconsistencies or a compromise of the monitoring functionality. Disabling updates protects the monitor from such vulnerabilities. Finally, modifying the monitoring code during operation requires reloading the NIC firmware or restarting the monitoring thread. Therefore, in effect, the new queries are treated the same as pre-defined queries.

We classify the queries supported by Orion into the following three categories.

On-Demand Queries: These queries are evaluated when the user or administrator is interested in a snapshot of the system state, or its history. The result of an on-demand query depends on the current state of the event summaries and does not reflect any past or future values. On-demand queries are usually used in conjunction with one-shot or continuous queries

to perform fine-grained diagnosis *after* detecting an event of interest.

One-Shot Queries: Detecting events requires continuous evaluation of predicates that define the event of interest. When performing an action, administrators or recovery/repair systems are often interested in an event that happens exactly once. For example, the failure of a machine is interesting exactly once. All further alerts just confirm its occurance. In these scenarios, Orion uses one-shot queries, which are evaluated continuously until an event of interest happens.

Continuous Queries: These queries track the state of the system and are evaluated continuously, for example the heaviest contended mutexes in each second. Unlike the one-shot queries, the monitoring system generates multiple alerts, one for each time the query is evaluated.

Continuous queries for monitoring live systems typically involve identifying and tracking *aggregates* of statistics, e.g., average, median, and standard deviation. These queries are also used to track a sequence of events when maintaining data structure invariants or verifying event order.

3.5 Case Studies

In this section, we describe case studies to illustrate the use of Orion in offloading system monitoring functionality. These case studies focus on detecting failures and on repairing damaged OS state. Without Orion, improving availability of these systems would require software modifications and monitoring agents to execute on the target system. Since the OS is unavailable in the scenarios we study, the monitoring agents would not be able to execute, causing a high degree of inaccuracy in failure detection and repairing OS state impossible.

3.5.1 Failure Detection

Failure detection or identifying that a node is crashed, hung, or is otherwise unresponsive is the simplest and most important monitoring task. Trivially, failure detection is impossible locally or from within a target system. Therefore, all failure detection approaches are external.

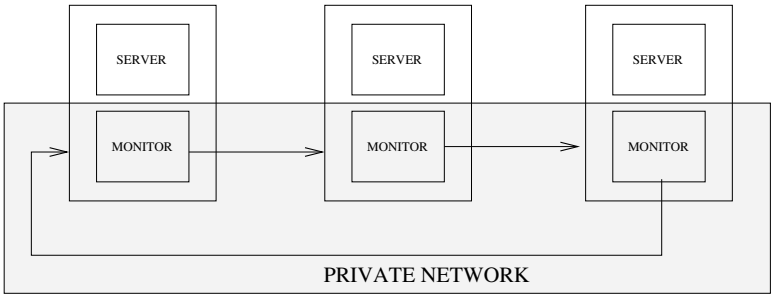

Figure 3.6: Monitoring ring configuration

Most failure detectors use a "heartbeat" mechanism, where a periodic message is sent from the monitor to the target and a valid response indicates liveness. On the other hand, if a response is not received or it is invalid, the node is declared dead and a failure is said to be detected.

The simplest monitoring configuration is *centralized*, where a single monitor sends heartbeats to all target nodes in a group. This leads to the monitor becoming a hot-spot for large groups. A *monitoring ring* configuration shown in Figure 3.6 overcomes the limitation of load on the central monitor. In this configuration, the nodes are arranged in a logical ring, each node in the ring monitors its successor in a predefined order (clockwise or anticlockwise).

The ring configuration can easily be extended to monitor not one but a fixed number $k + 1$ successor nodes for quick reorganization of the ring in presence of k failures. We use the ring configuration for monitoring in Orion and assume all failures are fail-stop and power failures are reported to the monitor through out of band link-layer mechanisms. In the following, a false positive in detecting failure means that the monitor declared a target node to be failed when it was in fact alive.

Failure Detector Accuracy

Heartbeats are periodic messages sent by a monitor to the target, that generates a response. The response can be a simple reflected message or it can include additional information, for example statistics counters. Timely responses received by the monitor indicate a healthy server. The monitor declares a system failed if it does not receive a response within a deadline.

The latency of failure detection (lag between failure event and detection) is bounded by the frequency of the heartbeat. Gupta et. al. studied the tradeoff between failure detection

accuaracy, message loss probability, and the heartbeat rate [83]. Using their notation, a monitor can detect failure within $\mathcal{T}$ with false positive probability $PM(\mathcal{T})$, over a network with the message loss probability of the underlying network, p_{ml} and the worst case Round Trip Time (RTT) by sending heartbeats at rate $R = \frac{Messages}{\mathcal{T}}$ given by

$$R \geq \frac{\log(PM\mathcal{T})}{\log(p_{ml})},\ where\ \mathcal{T} \gg RTT \tag{3.1}$$

Failure detection using simple reflected messages, e.g. ping, are limited. First, lost packets in either the forward or the reverse direction lead to false reports of a failure. Second, the successful reflected message simply indicates that the target network stack is active. A server that is otherwise unresponsive to applications or clients is not identified as failed.

Running a monitoring daemon on the target to listen and respond to the monitor's heartbeat request improves the failure detector accuracy. The monitor can identify overload at the target apart from simple liveness. Unfortunately, this daemon competes with other processes on the system and even long scheduling delays can lead to missed deadlines for target responses and lead to false positives in detecting failure.

Apart from the above, both the reflectors and monitoring daemons suffer from the monitoring overheads at the target. To maintain a reasonable failure detection accuracy and low detection latency, the monitor must probe the target frequently. High probe rates lead to high network and server load. Finally, traditional network heartbeats over ethernet suffer from no link-layer support for reliability. Lost packets are common, and the problem is further exacerbated if the monitoring traffic shares the network with applications.

Failure Detection with Orion

Of the three monitoring configurations for Orion, the local configuration cannot detect failures. Therefore, we focus on the I-NIC and the remote configurations for failure detection. In these configurations, it is important to ensure that the sensor box updates and fetch does not utilize the target processors as they might be unavailable.

For failure detection, we define the Sensor ($< ID, \mathcal{T}, V >$) using the default interrupt and VM statistics structures maintained by the OS shown as $TIMER\_INTR$ and VM_METER

sensors in Figure 3.4. These structures are updated on each interrupt, context switch, page fault, etc., which occur continuously on a live system. On the other hand, if the system is crashed, deadlocked, or otherwise unusable, these statistics are not updated.

The Sensor Box exports the memory where the interrupt and VM statistics structure (`struct vmmeter`) to the monitor. Since the SB reuses the memory as the sensor value, V, the updates to the SB are *implicit*. The monitor fetches the SB with a rate R and assigns a local timestamp to each sensor. The sensor value is an aggregate count of all events that occured between two consecutive retrievals of the SB from the target. As a liveness indicator, the monitor maintains a counter aggregate, the slope of the line connecting the two point updates in consecutive SBs. If a system is alive, this slope has a positive value, and it is 0 on a failed system, where the statistics and therefore the values in the SB View are not updated.

Failure detection queries are one-shot queries. The monitoring context evaluates the liveness condition ($slope > 0$) on each SB View refresh through an SB fetch. A crucial choice in designing a failure detection protocol is the SB View refresh rate. If the refresh interval is too small, the SB may not have been updated and would result in the monitor declaring the target as failed. If this interval is too large, the detection latency is increased. Orion uses the SB discovery to identify the refresh rate. Sensor Boxes specify their update rates when created with the `sbcreate` API call at the target. The rate is determined by the slowest sensor in the SB. For example, the SB containing the $TIMER\_INTR$ sensor defines the refresh rate as the time slice defined by the granularity of the system clock (HZ). To add hysteresis, the failure detection declares failure if the liveness condition is not satisfied over k refresh periods. In our example, we choose k to be three refresh cycles.

Once a monitor detects a failure, it stops updating its SB View and does not evaluate the liveness condition again. To ensure fail stop mode, Orion forces the remote OS to stop all execution by acquiring a Remote OS Lock, which denies access to the target system scheduler and hence any further processing on the target.

Orion takes advantage of highly reliable communication channels for fast and accurate failure detection. For the NIC configuration, the communication channel is the PCI bus, which guarantees no message loss and a bounded time for completion of SB fetch. In the Remote configuration, Orion uses Myrinet, a low latency high bandwidth interconnect that supports

RDMA. Myrinet supports a Go Back N ARQ protocol that ensures successful, in-order transmission of messages. The worst case Round Trip Time latency for Myrinet is a few microseconds. Therefore, in both cases, $p_{ml} \rightarrow 0$, which in turn implies that the sampling rate R can be arbitrarily close to $\mathcal{T}$ from Equation 3.1. In Orion, the latency of failure detection therefore depends only on the RTTs and Orion can successfully detect failures with latencies close to tens of milliseconds.

3.5.2 Repair of OS State Damaged by Resource Exhaustion

We illustrate the remote repair mechanism in scenarios where traditional techniques fail to prevent the system from becoming unavailable due to resource exhaustion, and cannot repair the system. We describe two such situations, show why the traditional mechanisms fail, and describe a remote healing solution.

Forkbomb: Process Table Repair

A forkbomb is a process that recursively spawns new processes, without doing useful work, until the resources on the system are exhausted. A forkbomb monopolizes the CPU resources of the system and does not allow other processes to execute. It also causes the process table in the OS to fill up, preventing new processes from being created on the system. The forkbomb starves all processes indirectly, as the scheduler has to traverse a large list of processes, to identify and update their priorities. No other user or system activity is possible when the scheduler is running.

In our test system (FreeBSD), the OS protects against the forkbomb, or any such runaway process by limiting *(i)* the maximum number of processes per user, and *(ii)* the maximum rate of process creation. When any of these limits is exceeded, the user is "locked out" of the system by killing all her processes, and not allowing her to create new processes. Other operating systems have similar protection mechanisms.

On a heavily loaded system, a forkbomb may never reach the system limit, so the built-in OS protection will not work. The system is not "dead", as all its hardware components and the OS are functioning correctly, but cannot do any useful processing. In this scenario, the system is inaccessible through the traditional channels (console, or network) as new processes (at least

the shell) are required to execute any repair task. The forkbomb also prevents the existing processes, e.g. daemons, from repairing the system by starving them of CPU cycles.

Orion uses two sensors to detect a resource unavailability at the target and to identify the fork bomb process and all its sub-processes, *(i)* $NPROCS$, which counts the total number of fork and exit system calls executed in the system and maintains a log of the difference between them, and *(ii)* the $FORK\_REC$ sensor shown in Figure 3.4, which records the parent-child relationships between the processes. These sensors record the total number of processes in the system and a shadow data structure representing the process hierarchy.

These sensors are recorded in the SB explicitly by registering hooks with the process create and exit subroutines. A loadable kernel module registers for these notifications and updates the SB usinng the `sbupdate` API call. The monitor maintains a dependency graph for the shadow process hierarchy and a counter aggregate for the number of processes in the system. We also use the rate of creation of new processes to predict the resource depletion before it occurs.

We use three detection policies that use the logs to identify a resource exhaustion. These policies define thresholds for the maximum number of processes, the depth, and the breadth of each level of the process hierarchy. Using the depth policy, the system can identify processes that create a large number of children. Using the breadth policy, Orion can identify sophisticated fork bomb processes, which create new process groups to limit the depth of the process hierarchy to avoid detection by the depth policy and the default OS protection mechanisms.

On detecting a fork bomb, Orion freezes all processing on the target node to prevent further damage to the OS state and to perform diagnosis and repair. To freeze the processing and to create a snapshot, Orion acquires the OS scheduler lock remotely, which prevents all further activity at the target system. Note that this suspends service to active clients for the duration of the repair but does not require termination of well behaved servers as in a restart based approach.

To repair the process table at the target OS, the fork bomb process and all its descendants must be killed. To minimize remote intrusion, we modify the process signal masks and rely on the target OS to actually terminate the process. The target defines a Remote Hook with the signal mask of all processes. This hook can be defined at startup or can be constructed on demand following detection. The external monitor constructs a signal mask with the $SIGKILL$

flag set. This signal is delivered to a process when it is scheduled for execution and cannot be masked off. Once the signal mask of all culprit processes are modified, Orion releases the remote OS lock and lets the target OS perform the cleanup.

MemoryHog: Memory System Repair

A process or a group of processes that allocate large amounts of memory may cause the system to exhaust its memory. The virtual memory abstraction allows each processes to allocate the maximum addressable memory. Under memory pressure, unused memory is moved to a backing store for anonymous memory called swap space. We define the usable memory on the system as the sum of the physical memory size, and the swap space size.

In our test system (FreeBSD), the OS limits the maximum amount of memory allocated per process. This limit cannot be too low as useful processes with a large memory footprint would be hampered. However, the maximum usable memory in the system can be exhausted with a small number of processes that allocate the maximum allowed memory without freeing it.

When the entire usable memory is exhausted, the OS has no alternative but to reclaim memory from processes. It calls an out-of-memory handler to choose the process with the largest memory footprint, and to kill it. Unfortunately such a brute force policy does not prevent a process that creates several child processes, each of which continuously allocate memory, from exhausting system memory. In one experiment, with as few as 30 such processes, we were able to prevent any useful execution on the system. Moreover, this policy can eliminate a useful process on the system if it has the largest memory footprint.

Although the system is alive, it is unusable, as no new processes can be created. Any local repair is also impossible if it requires allocation of memory as the resource has been exhausted. This prevents the administrator from carrying out any additional repair on the system. In contrast, our BD based system identified the memory hog, and repaired the system.

Orion uses three sensors, $OOM\_KILLER\_RUNNING$, the VM_METER, and the $FORK\_REC$ sensors described in Figure 3.4 to identify memory resource exhaustion at the target system. The VM_METER structure records statistics about the virtual memory subsystem including the number of pages in the buffer cache, number of pages available for low-level system processing, and thresholds for detecting low memory conditions. The

$OOM\_KILLER\_RUNNING$ is a flag, which is set when a severe low memory condition exists and the system starts terminating processes. Finally, the $FORK\_REC$ sensor is used to create a shadow process hierarchy described above to identify the process group that occupies the maximum memory in the system.

On detecting a memory shortage, Orion freezes all processing on the target system and identifies the set of processes that have the largest resident memory set. To identify these processes, Orion remotely traverses the process list and creates a candidate set for termination. The head of the process table is exported to the external monitors through the remote hooks. The remote traversal identifies the remote address of the next process structure and uses an internal mapping table to translate it to the appropriate DMA transfer descriptor, offset, and length. The process strucuture definition is statically defined in the Orion monitor.

The candidate set of processes are matched against a *whitelist* of critical processes, essential for the system to continue execution, e.g., the swapper, kernel resident threads, etc., and these processes are not terminated, even if at fault. The whitelist is specified by the administrators during initialization and may contain processes essential to maintain service to the clients. Finally, using the process table hierarchy, the process groups with the largest resident set sizes are determined and all processes belonging to this set are terminated. The remote hooks for terminating processes for memory hogs are identical to that of the fork bomb eradication policy.

3.6 Prototype Implementation

We have implemented a prototype in the FreeBSD 4.8 x86 kernel, using Myrinet Lanai-XP programmable NICs [132]. For remote monitoring and state extraction, we modified the Myrinet GM 2.0 library to provide in-kernel remote memory read/write operations between monitor and target machines.

We implemented the SB mechanism in the OS kernel. The event dispatcher is implemented as a user-space daemon, and the healing modules are user defined plug-ins for the event dispatcher. Healing modules are implemented as dynamically loadable libraries. The SB interface is implemented as a pseudo-device accessed both from the kernel (at the monitored node, for sensor updates) and from user space (at the monitor, for sampling the remote SB).

Remote OS Access

Remote access is enabled by registering the kernel memory of a target system with an I-NIC. FreeBSD allocates OS memory from a kernel virtual memory map, therefore, a kernel virtual page may map to different physical pages (if freed and reallocated). This is a problem for the NIC, which uses a translation table of virtual-to-physical memory mappings and maintains a mapping cache for fast lookups of frequently used mappings. To keep the NIC table in sync with the kernel page tables, we dynamically update virtual-to-physical mappings when needed (on kernel memory allocations).

Performing dynamic mapping updates also requires flushing stale entries from the NIC mapping cache. Flushing the cache on every mapping update incurs a high cost on a critical path (kernel memory allocation) and may create synchronization problems between the host processor and the slower Lanai. To avoid them, we chose instead to completely disable caching. The incurred penalty is negligible, given the low frequency and volume of the monitoring traffic (SB is light-weight and fits in one page, requiring just one translation lookup for access). For recovery or repair, an infrequent event, the penalty from not caching is paid only once.

Remote OS Locking

We have implemented a *remote OS locking* mechanism that blocks execution of system calls and of interrupt/exception handlers on the target machine. Remote OS locking is used: *(i)* to freeze a suspected target before starting recovery and thus completely eliminate unwanted effects of false positives in failure detection, and *(ii)* to freeze the target OS in order to have a consistent view of its state while performing diagnosis and repair operations. We have also used remote OS locking to remotely freeze a machine during emulated failure experiments.

Remote OS locking uses remote read/write operations on a "giant" shared-memory lock, in a two-phase handshake protocol. To acquire the lock, a remote requester (monitor) atomically writes a one-word lock request in target's memory. Lock acquire operations on the target OS were altered to check for posted remote lock requests after acquiring the lock, but before allowing the local acquirer to enter the critical section. If a remote lock request is pending, the local acquirer on the target relinquishes the lock to the requester by writing back to signal that

the remote OS lock is free, then blocks (spins) waiting on a flag. To later release the lock, the holder writes the lock free flag remotely and the target OS resumes normal operation.

This implementation relies on a giant lock maintained and used by the native OS for its own purposes (e.g., in some SMP versions of FreeBSD and Linux, for mutual exclusion in accessing related kernel data structures). However, in a kernel that allows fine-grained access to its data structures, adding a giant lock would kill concurrency. In this case, a better implementation is possible by noting that the *local* global locking primitive is actually not needed to block *remote* accesses. At the time remote access is granted by a local acquirer, the local global lock ensures that: *(i)* no other code will be able to enter the kernel (since accesses are guarded by lock acquires and the lock is held), and *(ii)* no code is currently executing in the kernel (since the lock can only be held by the acquirer, which has not yet entered the critical section).

The idea is to enforce these two conditions *separately*, from the remote node, with assistance from the target OS. To enforce *(i)*, the remote node will set a blocking flag that the target checks at all kernel entry points (system calls, interrupt handlers, fault handlers, etc.). To enforce *(ii)*, the target OS will maintain a counter of threads currently executing in the kernel for each class of entry points, atomically updated at entry and upon exit. After setting the blocking flag, the remote node will wait until all threads of execution have "drained" from the kernel, i.e., until all thread counters become zero. Note that in both implementations a timeout can be used to break an infinite wait, e.g., if the system hangs with the giant lock held or with a nonzero counter.

3.7 Evaluation

We present an evaluation of the Orion mechanisms to offload monitoring functionality and the effectiveness of remote monitoring and repair in the context of the two case studies. For failure detection, the goal is to evaluate the overheads imposed by Orion and the accuracy of the failure detection. For remote repair, the goal is to demonstrate the effectiveness of restoring the system to a functional state.

The experimental setup consists of DELL PowerEdge 2600 2.4 GHz, 1 GB RAM dual-processors interconnected by 1 Gb/s Ethernet. Server nodes run FreeBSD 4.8 incorporating

our BD prototype. Orion is implemented with Myrinet LanaiX NICs with a 133MHz PCI-X interface [132].

3.7.1 Failure Detection

The goal of failure detection is identifying a faulty node accurately and as soon as possible. To evaluate Orion mechanisms, we instantiate the monitor at the I-NIC and at a remote node. We introduce failures at the target node using a kernel module for a faulty network interface driver, which crashed the system upon loading it in the kernel. Once crashed, all system activity ceased and the system was rebooted for further experiments. Using a SB view refresh rate of $100ms$, and the hysteresis of 3 refresh cycles with no updates to the sensors, Orion was able to detect the failure in all cases within $400ms$ of introducing the failure.

Monitoring Overhead

The sensors used to detect failures are statistics data structures, maintained by default in the OS (`struct vmmeter` in FreeBSD), Orion does not introduce any additional overhead at the target. During initialization, a one-time registration cost of the memory descriptors is incurred, which is less than $10\mu s$.

The CPU overheads at the monitor involve retrieving the SB contents from the target and evaluating the liveness condition, and the memory overheads involve storing the log required to evaluate this condition. We use the slope of the line connecting the context switch counter and the timer interrupt counter as the liveness indicator. To maintain these slopes, we store the contents of two SB Views at the monitor. The size of the `vmmeter` structure is 196 bytes, therefore, the memory overhead at the monitor is 392 bytes.

On a monitor node, the overhead includes *(i)* monitoring cost (reading the local view of the monitored SB, comparing sensor values, etc.), and *(ii)* cost of transferring the remote SB from the target node. To determine them, we measured the CPU usage of a monitor process while varying the sampling rate of a remote SB with 100 sensors. In the worst case (sampling the SB in an infinite loop), the CPU usage was 46%. Sampling every 10 ms (the lowest granularity of a timer), the CPU usage is about 5%, while at 100 ms it drops under 1%. This shows that fast failure detection can be performed with low overhead on a monitor node.

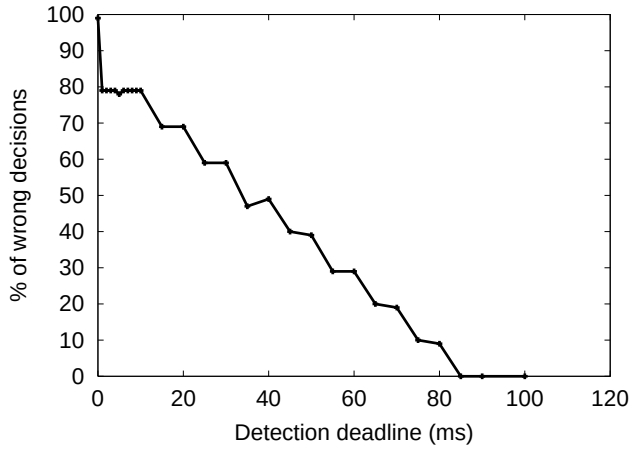

Figure 3.7: Variation of false positives in failure detection with the detection deadline.

The latency of transferring the SB contents for the I-NIC configuration is the DMA transfer time, which is less than $100\mu s$, the minimum timer granularity available at the NIC. For the RDMA connected remote monitoring configuration, we measured the transfer time to be around $300\mu s$. The low transfer times indicate that detection deadlines of less than $1ms$ are possible with Orion if the sensors are updated at the target with similar intervals.

Detection Deadlines and False Positives

This experiment shows that the detection deadline $\mathcal{T}$, must be carefully chosen to match the behavior of the sensors used for failure detection, in order to avoid false positives at a monitor. A *false positive* occurs when a healthy node is wrongly declared failed by a monitor.

In this experiment, we illustrate the effect of choosing aggressive SB View refresh rates, which may not allow the sensor values to be updated leading to false positives. We artificially induce false positives in failure detection by a monitor, using the number of context switches, recorded in the VM_METER sensor as the liveness indicator. A remote monitor samples the counter with period $\mathcal{T}$ equal to its detection deadline while a CPU-bound task runs on the target node. Since this task does not block, if there are no other runnable tasks, no context switch may take place within a time-slice. The counter may stall for the duration of a time-slice, and a monitor may declare the node faulty if the detection deadline is smaller than the time-slice. Figure 3.7 shows the fraction of false positives with increasing detection deadline. Since the normal time-slice of our test system is 100 ms, deadlines under this value run the

No. of processes	Time (ms)
100	140
500	263
1000	350
1500	426

Table 3.4: Variation of the repair cost with the number of processes in the system

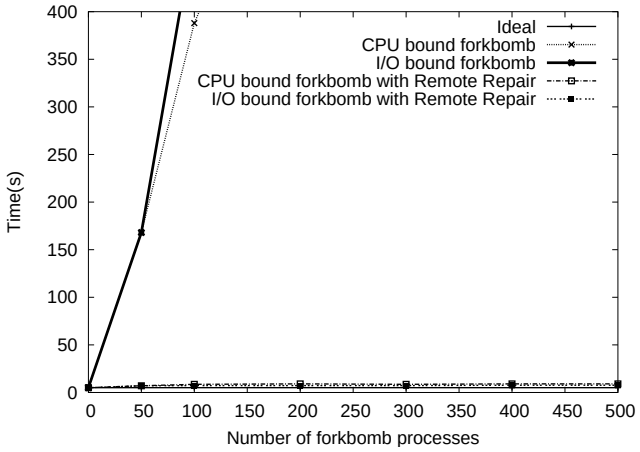

Figure 3.8: Variation of execution time of test program with number of forkbomb processes.

risk of inducing wrong decisions. As the deadline increases, so does the chance that a context switch occurs before the deadline. For deadlines larger than 85 ms, other activity in the system eliminates false detection. This shows that the system is sensitive enough to fail "as-expected" and expose programming errors caused by unrealistic detection deadlines.

3.7.2 Remote Repair

For remote repair, the goal of our evaluation is four fold. First, we show that a computer system can be brought down using the programs described in Section 3.5.2. Second, show that the system cannot be repaired locally, i.e., either the system is unresponsive, or it terminates essential processes like an application server. Third, we show that we can detect and repair such cases using Orion. Fourth, we show that the monitoring and repair in our system are efficient.

Diagnosis and Repair Cost. Diagnosis and repair involve traversing the remote process table and building a local view, identifying the culprit, and killing all its processes.

The cost of repair therefore depends on the number of processes present on the target node. Table 3.4 shows the variation of the average cost of repair with the number of processes on

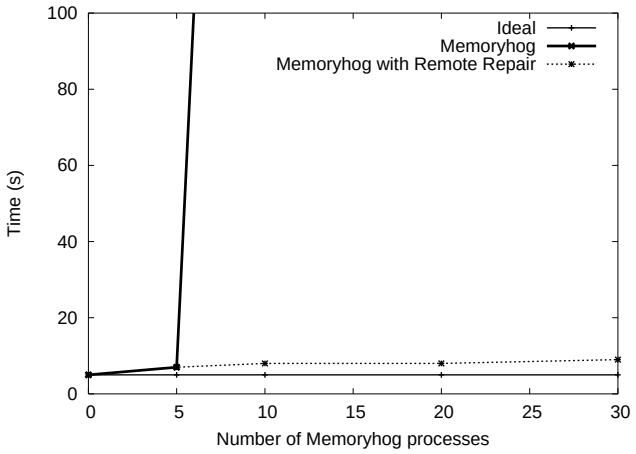

Figure 3.9: Variation of execution time of test program with number of memory hogs.

the target node. While the repair cost grows with the number of processes, in all the scenarios we studied, it takes less than half a second to execute. This shows that repair (an exceptional action) is fast, and also that it should not impose too much overhead on the monitor system.

Repair Effectiveness. To illustrate the two case studies of Section 3.5.2, and to show that remote repair works while local repair is practically impossible, we developed two test programs: a forkbomb and a memory hog.

The forkbomb creates processes that execute in a tight loop until the CPU cycles on the system are exhausted. A variation of this forkbomb continuously reads from a pseudo device (/dev/zero in FreeBSD) and writes to a NULL device (/dev/null in FreeBSD). This prevents the scheduler from lowering the priority of the forkbomb and of its children.

The memory hog exhausts the system memory and swap space. It has a controller process, and a variable number of child processes. The controller creates the children which allocate memory in a loop. If a child is killed by the system, the controller spawns a new process.

To illustrate the effect of the above two programs, we run a simple process that executes in a loop, measuring the time it takes for each iteration. The user time of this process is constant through all experiments. However, when the forkbomb or the memory hog are executing, the program takes longer to get its share, increasing the wall clock time measured on the system. The ideal time in Figures 3.8 and 3.9 is the time the program takes to execute when there is no other load on the system (5s).

Figure 3.8 shows the variation of the wall clock time for the test program with the number

of processes created by the forkbomb when *(i)* a CPU bound forkbomb executes, and *(ii)* when an I/O bound forkbomb executes. We see that the execution time grows unbounded for the forkbomb cases without repair, while it stays close to the ideal value when remote repair is performed.

Figure 3.9 shows the variation of the wall clock time for the test program with the number of processes created by the memory hog. Our system has 1GB of RAM and 2GB of swap space. Our test OS (FreeBSD) limits the maximum memory allocated by a process to 512MB, therefore up to 5 processes the system is well behaved and repair is not triggered. Once the memory is exhausted, the system becomes unavailable and execution time without remote repair explodes. With remote repair, the memory hog is identified, and all processes with the same userid are killed. The system is recovered with minimal disruption.

With around 400 processes created by the forkbomb, or with a pool of 30 memory hog processes, the test program did not complete for more than 30 minutes. In fact, we failed to access the system for repair, and we had to reboot to regain control over the machine. With remote repair, the system correctly identified the culprit and recovered in a very short time.

3.8 Related Work

There is a large body of research and commercial products that focus on monitoring computing systems and networks.

3.8.1 Hardware Sensors and Platform Management

Intelligent Platform Management Interface (IPMI) [55] is a standard that defines hardware components and the communication protocol to gather hardware sensor information within a computer system. The term ”platform management” is used to refer to the monitoring and control functions that are built in to the platform hardware and primarily used for the purpose of monitoring the health of the system hardware. This typically includes monitoring elements such as system temperatures, voltages, fans, power supplies, bus errors, system physical security, etc. It includes automatic and manually driven recovery capabilities such as local or remote system resets and power on/off operations. It includes the logging of abnormal or ‘out-of-range’

conditions for later examination and alerting where the platform issues the alert without aid of run-time software. Lastly it includes inventory information that can help identify a failed hardware unit. IPMI is composed of a system management bus Intelligent Platform Management Bus (IPMB) which enables IPMI enabled devices and emergency system management components, e.g. network interfaces to be plugged in to the system and communicate with other components in the system. IPMI support is incorporated in several vendor offerings and is used in conjunction with OS and management software, e.g. HP Openview [179] and IBM Tivoli [110] to monitor and control hardware components. IPMI is an evolution of similar functionality in systems like Tandem [22], TARGON32 [34], and DEC Titan [137], which use monitoring and failure detection in conjunction with highly redundant hardware to provide continuous operation. Orion focuses on offloading *software* component monitoring, including application and OS subsystems. However, the sensor information generated by hardware can also be incorporated into the Sensor Box and used to define monitoring policies.

Flight Data Recorder [217] uses specialized hardware to interpose on the memory bus to monitor and log all modifications to system state, including memory, cache, and processor registers. This allows the system to replay failed executions and identify exact points of failure. Specialized hardware has also been used to monitor memory traffic to identify the page miss ratio and use this information for intelligent memory allocation. Memory traffic is not visible to Orion, therefore it is impossible to build a complete view of the memory state. However, providing access to the memory bus requires modifications to the memory bus and cache controllers and is not practical for commodity computer systems.

3.8.2 Software Monitoring

Monitoring software through internal logs and external statistics has been studied in a variety of contexts. Self-monitoring was used in [177] for adapting OS behavior with the goal of increasing performance. Our system also relies on introspection by the monitored system (through the SB mechanism), but uses external, nonintrusive observation of the SB and of other OS state to detect and diagnose exceptional events in the monitored system.

K42 [189] is an OS with built-in support for component hot-swapping. While in principle hot-swapping can fix certain cases of damaged OS state (e.g., when the cause is an OS bug, the

faulty OS module can be dynamically replaced with a correct one), it cannot address the more frequent situations when the trigger lies in user space (e.g., faulty or malicious user programs). Moreover, such systems are built from scratch to provide hot-swapping support. In contrast, our system uses a slightly modified general-purpose OS to provide generic support for monitoring, diagnosis and recovery of a computer system from damage to its OS state.

Defensive programming [157] is a technique in which compiler-assisted program annotation is used to insert introspecting and reactive code (sensors and actuators) into application/OS code, with the goal of detecting and alleviating DoS attacks. Similarly to BD, defensive programming requires cooperation from system and/or application code through the use of minimal APIs for augmenting software functionality. Defensive programming could be used (at least in principle) to detect and react to violations of constraints on system resource usage. However, in the case of OS resources, this would require extensive changes to be integrated with OS kernel code, including complex detection and repair policies to be statically built into the OS. Moreover, collocation of such code with the system it protects would incur CPU overhead to execute the monitoring and detection algorithms. In contrast, our system provides a simple monitoring abstraction along with support for flexible detection/repair mechanisms to be easily implemented, tested and deployed from a different system, without using resources of the target system.

Language support for automatic error detection and repair of data structures is explored in Acceptability Oriented Computing (AOC) [60, 166]. A BD-based architecture can be integrated with and leverage such support to define diagnosis/repair algorithms on OS data structures. Using a BD-based architecture in conjunction with AOC can also solve its vulnerability to system faults and resource exhaustion by monitoring resource constraints and performing repairs remotely from another system.

Self-Healing Systems use statistical learning techniques to characterize the behaviour of the system, identify anomalies, and take actions to *heal* the system [98]. Learning the behaviour of the system requires comprehensive training data, which must be similar to the actual observations on similar data at run-time. This approach has been shown to be successful in case of fault diagnosis in recoverable Internet services [41, 48, 47] and enterprise-wide load balancing [46, 52]. Similar techniques have been used to determine the workflow in distributed

applications and debug distributed systems. Orion builds efficient mechanisms to provide continuous access to monitoring data. Through the SB API, Orion also enables applications to participate in generating the monitoring data. Therefore, the above techniques are complementary as they operate on the available data to construct models of expected behavior, and identify anomalous conditions and adapt system behavior.

Copilot [147] is a system that provides access to system memory from an external coprocessor to detect intrusions. Similar to Orion, Copilot uses a PCI card on the I/O bus of the target to provide nonintrusive access to the target memory. However, unlike Orion, the OS and applications cannot participate in the monitoring framework, which in turn is limited to identifying intrusions and violations of data structure invariants, e.g., in file systems, through statically defined policies.

3.8.3 Interposition and Virtual Machine Monitors

Nooks [201] is a system that interposes on the communication between the OS and the device drivers by creating a thin wrapper layer between the driver modules and the OS kernel. Nooks monitors the health of the driver and on detecting a fault or hardware lockup, reloads the driver and prevents a system-wide failure. Nooks requires significant changes to the OS and is limited to monitoring software on a single node. In contrast, Orion performs external monitoring and while it cannot prevent failures from incorrect or inconsistent accesses within a system in real-time, it can be used to generate alerts on detecting failures and preserving state for future repair actions.

Application monitoring and recovery has been proposed in Rx [158], where the OS maintains snapshots of application state and on identifying a failure, performs a controlled replay while modifying environment variables. This approach recovers applications from faults that arise due to scheduling decisions and other environmental factors that affect execution external to the application logic. Repair with Orion is performed externally, from the I-NIC or from a remote monitor. However, similar to Rx, Repair Hooks in Orion use controlled modification of system memory to modify execution of application and OS subsystems for affecting repair.

Hypervisors [37] and Virtual Machine Monitors (VMMs) [21, 214, 209] interpose between the OS and the hardware and multiple OS instances to share the same hardware resources.

VMMs have been used to create primary-backup pairs for client transparent recovery [37], identify and recover from software misconfiguration [214], and even for logging all OS operations to provide a time-line of failures [66] and allow backtracking for intrusion detection [100]. Recently, VMMs have been used to allow application-specific predicates to identify and handle application monitoring and recovery. In principle, offloading monitoring to Orion is similar to offloading it to a VMM, but differs as Orion does not control access to hardware. However, Orion can be instantiated within a VMM and use a separate VM for monitoring and even for repair.

In the context of Internet Services, interposing on the client-server path has been used for monitoring client visible performance characteristics, identifying failed requests, monitoring availability of services, and implementing client-transparent failure mitigation. The Recovery Oriented Computing (ROC) [146] project studied the behavior of Internet Services from the availability perspective [141] and built systems based on software rejuvenation [41] and operator controlled rollback from an inconsistent state [38]. The Vivo project [133] studied the relationship between performance and availability in Internet Services and database systems. Interposition was used to create distinct *production* and *validation* clusters where the behavior of different versions of the service were studied using live and trace driven requests [134, 138].

Several systems use interposition to monitor network traffic. These systems capture all packets flowing on the network and extract statistics or patterns to build a view of the ongoing communication. Building such systems is challenging due to the high network bandwidths and the large number of communicating end-points, and offloading monitoring functionality to a programmable network interface is a well accepted technique in network monitoring. Network monitoring systems based on offloading have been proposed to identify traffic characteristics, heavy hitters, application communication patterns, intrusion detection, etc. While Orion has a different goal, techniques developed for maintaining historical information with a small memory footprint, e.g. randomized data structures [28, 71, 54], are relevant for improving coverage and accuracy of monitoring with Orion. Network monitoring data can also be used in conjunction with the Sensor Box data to further broaden the scope and improve efficiency of monitoring systems with Orion.

3.9 Summary

In this chapter, we presented Orion, a system architecture that offloads monitoring functionality to a programmable intelligent network interface card (I-NIC) and enables remote monitoring and repair of operating system state. The I-NIC sits on the I/O bus of the target system, has its own processor and memory, and is connected over a private network. Using the I-NIC, the target memory can be accessed nonintrusively, without involving its CPUs. Therefore, monitoring with Orion continues to function, even when the target OS is crashed, hung, or is otherwise unavailable.

We presented mechanisms to record monitoring records, *Sensors* at the monitoring target and an OS abstraction, *Sensor Box (SB)*, which collects and exports sensors to remote monitors. As part of the Orion framework, we presented the monitoring context, which retrieves the SB and processes its contents to generate an event log and provides a query interface to build monitoring policies.

We demonstrated the use of Orion through two case studies that perform failure detection and remote repair of OS state damaged due to resource exhaustion. Through the experimental evaluation of our prototype, we characterized the overheads of Orion mechanisms. Our evaluation also showed the effectiveness of Orion in accurately identifying failed nodes in a cluster, and eradicating processes that monopolize the OS process table (fork bomb) and memory resources (memory hog).

Chapter 4

FileWall: Offloading Policy Enforcement in Network File Systems

4.1 Problem Statement

Network protocol evolution is limited due to the tight binding between clients and servers. Simple extensions to protocols are difficult to deploy since they require modifications to a clients, which may have diverse hardware and software profiles, and servers, which may be proprietary and closed source. Users and administrators are willing to tackle protocol upgrades for substantial and critical upgrades, e.g., for better performance, bugfixes, and plugging security vulnerabilities. However, extensions that affect *access policies* are largely ignored. Such limitations have long been recognized in networking research, and offloading functionality to network middleboxes has become increasingly popular for protocol extensions.

Network middleboxes [210], for example, Firewalls [74], Network Address Translators (NATs) [85], Network Intrusion Detection Systems (NIDS) [213, 182], Virtual Private Network Concentrators (VPNs) [80], etc., are an integral part of the network infrastructure today. These systems interpose on the network path between critical infrastructure services and the untrusted, possibly vulnerable clients of these services. Through interposition, the middleboxes, *(i)* identify and discard unwanted, malformed, or malicious traffic, *(ii)* transform packet contents to map hosts with private addresses to the public address space, and *(iii)* generate traffic monitoring and profiling data based on administrator defined policies and the state accumulated during operation.

Network file systems, e.g., NFS, CIFS, etc., are an important part of the *critical network infrastructure* protected by network middleboxes. These file systems provide centralized storage for data shared by several users. The user interface of network file systems has been designed to be identical to the local file systems. Uniform interfaces enable applications to work seamlessly when file system data is stored locally, or over the network. Unfortunately, due to this limited

interface, the access control of network file systems is also *identical* to local file systems.

Access control in existing network file systems relies on primitive mechanisms, like Access Control Lists and permission bits, which are not enough when operating in a hostile network environment. While file servers can be protected from unauthorized network access by filtering out unauthorized network traffic at the edge of the network, e.g., at firewalls and network intrusion detection systems (NIDS), such protection is unavailable *within* the network, where all file system clients reside. Moreover, using a local firewall at the servers completely ignores the file system semantics when defining access policies.

Unfortunately, both unmodified network file systems and network middleboxes cannot independently implement policies effectively on file system accesses. On the one hand, network middleboxes are limited to using the network context information, for example IP addresses, to implement *network* access policies. On the other hand, network file systems rely on the limited local access control mechanisms, e.g., Access Control Lists and rwx permission bits, to implement access policies, and ignore the network context. As a result, simple access control policies to protect the file system, for example, preventing user accesses to the file server, simultaneously from multiple machines, are difficult, if not impossible to implement in network file systems.

We believe, a combination of file system context and network context is essential to successfully enforce context aware network file system access policies. In a network file system, the file system context includes naming, file hierarchy, and the semantics of the file system operations, while the network context includes the client identities, e.g., IP addresses and hostnames, the network characteristics, e.g., bandwidth, delay, loss rate, and the network topology.

In this chapter, we present FileWall, a middlebox that combines network and file system contexts to offload enforcement of file system access policies. It interposes between file system clients and servers and *mediates* their interaction. It captures, modifies, and generates network file system messages, provides a persistent state storage mechanism, *access context*, and defines an execution engine for rules, which implement access policies. Figure 4.1 shows the FileWall architecture. Analogous to a firewall, the file servers and FileWall are trusted and reside in the same network domain, whereas the clients are external. Administrators have exclusive access to FileWall.

FileWall provides a single point of administration for enforcing access policies on network file system communication. While a similar functionality could also be implemented at the servers, there are several benefits of interposition. First, it provides isolation by offloading the monitoring and control functionality from the file servers. This leads to a separation of concerns and allows file servers to evolve independent of the access policies. Second, by restricting access only to administrators, and allowing no user programs or daemons from executing on it, FileWall reduces the chances of subversion of the file server by rootkits. Third, offloading through interposition enables FileWall to virtualize the network endpoint visible to clients, allowing file system federation, failure recovery, and system upgrades to be handled transparent to the clients. Finally, FileWall requires no modification to existing file servers and is readily deployable.

Realizing FileWall as a network middlebox is challenging. While clients and servers have a complete knowledge of the file system state, only the state updates are visible over the network. Therefore, the the file system state must be inferred and maintained externally, using message history and protocol specifications. Network file systems are built on top of transport protocols, which implement their own semantics. Therefore, any FileWall cannot make arbitrary modifications to messages and flows, and must adhere to the semantics of the underlying transport protocols. An additional challenge for implementing FileWall in the network is the usability. For administrators to use the system, it must be easy to configure, monitor, debug, and extend. Finally, performance is a first order concern in network file systems. Latency and throughput requirements are more stringent, and they directly affect user-visible performance. Therefore offloading policy enforcement at a network middlebox should not degrade the file system protocol performance.

4.2 FileWall Model

Offloading file system access policy enforcement in the network requires a network compute element. This element must adhere to a set of constraints. First, it must be autonomous and programmable. Autonomy ensures that clients and servers are not modified when defining access policies. Programmability is necessary as the access policies may be extended, refined,

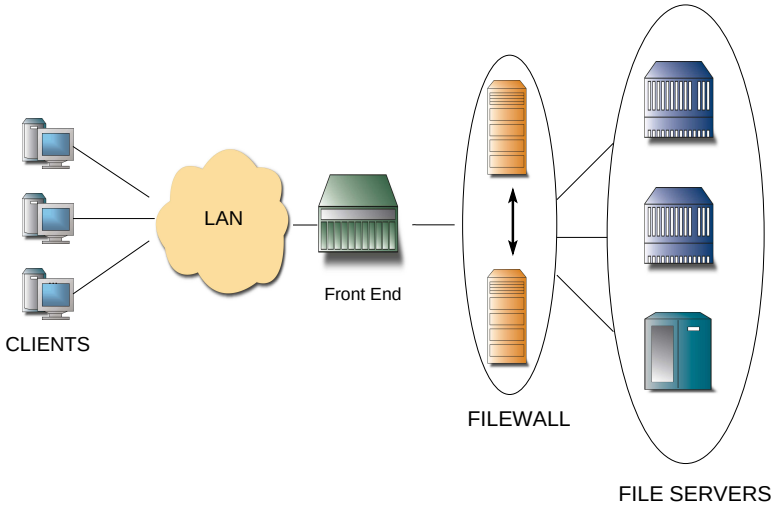

Figure 4.1: FileWall architecture.

or updated, due to organizational or legislative policy changes. Second, all network file system messages and message flows must be visible to the element. Third, modifications can be performed on messages and flows to alter the behavior of the file system protocol, but these modifications must preserve all underlying protocol semantics (e.g., RPC, TCP/UDP, etc.)

FileWall satisfies these constraints. It is a network middlebox, which intercepts all messages flowing between the client and the server, and transforms these messages to modify and extend network file systems. FileWall is transparent to clients and is collocated in the same network segment as the servers. FileWall is trusted by the file servers and shares the servers' authentication keys, which enables it to intercept, interpret, and transform encrypted or signed messages. File system policies are defined by administrators who have exclusive access to FileWall. A single policy represents a unit of FileWall processing. Similar to an object oriented program, each policy is defined by a class that determines its behavior and adheres to a common interface.

FileWall is a firewall for network file systems that offloads policy enforcement to a network middlebox. FileWall combines file system context information with network context information to evaluate policies that operate on file system messages. Table 4.1 compares the features of a typical firewall/NAT with FileWall. As shown in the table, both firewalls and FileWall

	Firewall / NAT	FileWall
Attributes	Packet	File System Message
State	Flow State	Access Context
Event	Packet Arrival, Timeout	Message Arrival, Timeout
Condition	Firewall Rules	FileWall Policies
Action	Forward, Rewrite, Discard	Forward, Rewrite, Discard, Generate
Access	Administrator Only	Administrator Only

Table 4.1: Firewall vs. FileWall

utilize attributes contained in messages. Additionally, both are event-driven, evaluating conditions (rules) on message arrival or when an timeout occurs. Both maintain state, either network flow state or access context. A firewall can either forward, rewrite, or discard a packet, while FileWall may choose to forward, rewrite, discard, or generate a new message. Finally, both are configured through an restricted interface, which is accessible only to administrators.

4.2.1 Network File System Model

In our model, all file system accesses are remote, and clients cannot log on directly to the server and modify the file system. Therefore, all modifications to the file system state are performed over the network as file system messages, and pass through FileWall.

A file system message defines a single unit of transfer between clients and servers, which is composed of one or more packets sent over the network. For a datagram protocol, e.g., UDP, each packet contains a file system message. For stream oriented protocols, e.g., TCP, client connections are made directly to FileWall, which in turn establishes connections to the servers, and messages are identified by special record markers. Each message contains attributes that determine the file system operation. A message sent by a client contains the attributes corresponding to a remote procedure call (RPC), while a server sends the response to this call.

FileWall is responsible for extracting file system attributes from messages and reconstructing messages with new or transformed attributes. Therefore, it must understand the data representation used by the protocol. In the following, we describe FileWall built for the ONC/RPC protocol that uses the XDR data representation. However, FileWall can be easily extended to work with other data representation standards.

FileWall operates on file systems that follow a transactional mode, that is, every file system

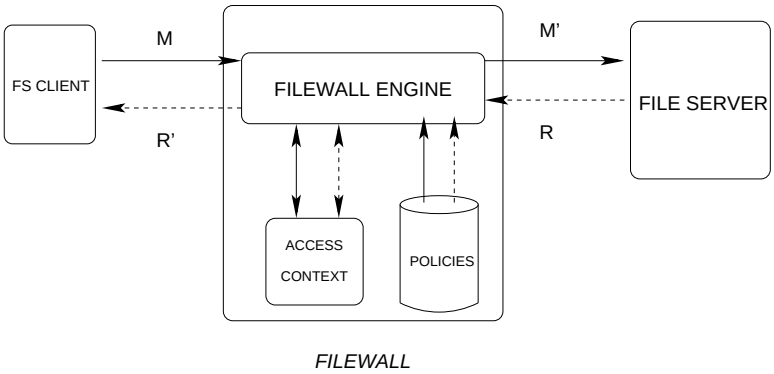

Figure 4.2: FileWall Model.

operation initiated by the client must receive a response from the server. The client state is not updated until the server reply is received. Most existing network file systems, for example, NFS and CIFS, follow the client driven transactional model.

4.2.2 State Model

FileWall policies are executed using the attributes contained in the file system message, in the context of state maintained by the policy or present in the execution environment. We call this state the *access context*. There are two components of the access context: *(i)* static context, and *(ii)* dynamic context. Each component is available to policy writers.

Static context is read-only state specified by the policy writer during initialization. This state is not updated during execution. Examples of such state include lists of user identifiers, list of file servers, and servers' authentication keys.

Dynamic context contains state generated by policies during execution. FileWall policies can store arbitrary state in the dynamic context and retrieve it at a later stage. FileWall does not interpret the state, and treats it as a black box. Policies may choose to store their dynamic context in persistent storage through a database, which guarantees the ACID properties (Atomicity, Consistency, Integrity, and Durability) of the stored state. By default, dynamic context stored in memory is purged on every restart and must be rebuilt from scratch.

4.2.3 Message Transformation

FileWall is external to clients and servers, therefore all file system policies are implemented using a combination of *attribute transformation*, *flow transformation*, and *flow coordination*. These primitives represent the minimal set of functionality supported by FileWall for message transformation.

To illustrate message transformation, Figure 4.2 shows the flow of a client request message (M) and the response message (R) through FileWall. The client sends its request to the server. FileWall intercepts this message and invokes policy handlers, which use access context to transform this message to M' and forwards M' to the server. The server responds with a message (R), which is transformed to R' and sent to the client as the file system response.

Attribute Transformation: File system messages are constructed by the sender as a set of arguments and updates in a request, or as a response to a previously received request. Upon receiving a message, the destination (client or server) executes the request, or updates its state based on the response. The attributes in the message determine the corresponding action.

Policies may modify the attributes to affect the action performed at the destination. These attributes may also be modified to request additional information. Attribute transformation includes attribute remapping, operation modification, and argument transformation.

Attribute remapping adds a layer of indirection between the client and the server. The client visible attributes are modified in each message by FileWall before being presented to the server. The server responses are modified similarly to remap the attributes to those visible to the client.

File system messages include the identifier of the operation to be performed at the server. Through operation modification, client requests are transformed to execute different functionality. The responses to the modified operations must be transformed to match the original request.

Argument transformation may modify any part of the file system message to implement policies. Arguments include the data updates (writes) and queries (reads). Therefore, FileWall can also implement data transformations through this mechanism.

Flow Transformation: A network flow represents a client-server channel where all file

system messages are exchanged. One or more messages in a sequence determine the operations performed and the state updated at the clients and servers. Flow transformation includes filtering, reordering, and injecting messages within a flow to modify the file system behavior.

Filtering removes messages sent by one end of the flow from being received at the other. Such transformation is useful to implement access control and security policies, where only messages representing permitted operations are forwarded to the server, removing all other messages from the flow.

Message injection is the dual of filtering and introduces additional messages in the flow. Recall that in our model, all file system requests must generate a response. Therefore, for every filtered request, a response must be injected in the flow to maintain protocol semantics and operation. Additionally, message injection is used to retrieve state information from the server. This state, while required by the policies, may not be available for all file system operations external to the file server and may require specialized operations. Request injection and filtering out the response are used to retrieve such state from the server.

Message reordering modifies the sequence of messages within a flow. Such transformation may be used to modify the caching and read-ahead mechanisms at the file server by changing the inputs to the existing mechanisms. Such an approach has previously been used in context of Infokernels [17], which use carefully crafted input sequences to modify existing OS mechanisms like buffer cache eviction.

Flow Coordination: For a single client-server pair, all FileWall policies may be implemented using attribute and flow transformation. However, when policies affect more than one client-server pair, additional mechanisms to transform messages across a collection of flows is necessary. Flow coordination includes multiplexing/serialization and demultiplexing/fan-out across a group of flows.

Multiplexing or fan-in operates on a collection of flows and generates a single flow of messages delivered to the client or the server. Multiplexing may be used to implement novel consistency semantics by controlling the sequence of messages across flows, atomic updates by serializing all operations performed on a file system object or a group of objects, etc.

Demultiplexing or fan-out are the duals of multiplexing and separate a single flow of messages into multiple flows. These multiple flows may correspond to a previously multiplexed

collection or may be new flows generated for replicating a message flow. These are used to implement file server replication, where multiple flows are generated, one for each server replica, from a single client-server flow. Each request must be replicated using fan-out and the responses combined using fan-in.

4.2.4 Policy Model

FileWall policies are used by administrators to modify and extend network file systems. FileWall policies are stand-alone and execute independently. Policies may be *primitive*, where the functionality is self-contained, or *composite*, where multiple primitive policies are combined by connecting them in a chain.

The scope of file system policies is broad. The policies differ in their intended functionality, state requirements, and transformations required. File system policies have previously been proposed for performance, including aggregating file servers and functional decomposition, for access control and security, for file system management, including monitoring, replication and failover, and for extending semantics or implementing new policies for consistency, atomic updates, versioned updates, etc.

We define four classes of primitive policies and describe a small set of representative policies from these classes. This set is by no means exhaustive. However, it illustrates the key features, which are used to implement policies, and the ease of implementing such policies using FileWall. These policies define templates, which can be utilized by administrators, to implement the most commonly used functionality. Moreover, these templates can also be used as a starting point for writing customized policies.

The first class of FileWall policies performs only attribute extraction and state updates. To illustrate this, we describe a message flow statistics gathering policy. The purpose of this policy is to collect per-client/per-server statistics about the messages flowing between the clients and servers.

The second class of policies performs attribute transformation. To illustrate this, we describe a file handle security policy, which generates per-client virtual file handles and stores a mapping between the virtual and real (server generated) file handles. The purpose of this policy is to add an additional layer of security. Server generated file handles typically reveal

information about the server, such as OS version and date of file system creation, as described in [206].

The third class of policies performs flow transformation. To illustrate this, we describe a temporal access control policy implemented with FileWall. This policy prevents all accesses to the file system during a specified time window. This example also demonstrates how an extended ACL attribute, time, can be added to the default `rwx` access control. This attribute is maintained only by FileWall and is not visible either to the client or the server.

The final class of policies performs flow coordination. To illustrate this, we describe a replication policy, which generates copies of client-server messages for each replica server. Virtual file handles are used to maintain a one-to-many mapping across replicas. We assume all servers start with identical state and the failure mode is fail-stop. That is, once a server fails, it must be brought back to a consistent state external to the normal file system operation. Server-client messages are forwarded to the clients only after a response is received from each of the replica servers.

4.3 FileWall Design

In this section, we present the design of the FileWall system. Our goal while designing FileWall was to keep the core of the system small and provide a set of primitive policies, which can then be composed. The core functionality of FileWall is message construction, attribute extraction, scheduling, and message forwarding. Message transformation and state maintenance, which extend file system functionality, are implemented by policies in the context of message handlers. The minimal functionality of the FileWall core leads to a small and easy-to-maintain code base and encourages modular policies and code reuse.

In the following, we first describe policy execution, composition, and scheduling. We then discuss how FileWall associates requests and responses, and finally describe how policies are specified in the system.

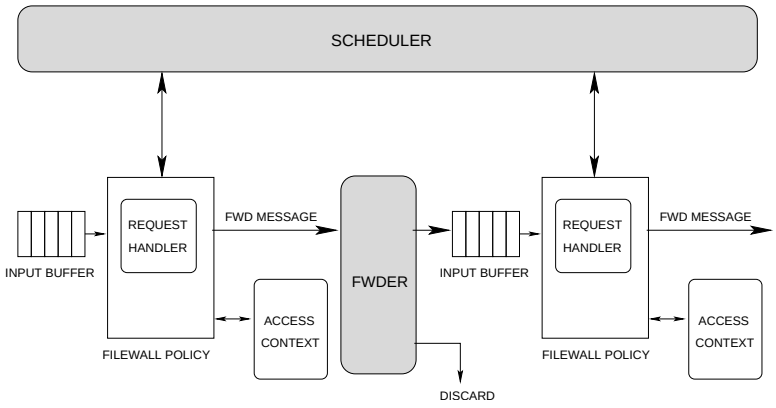

Figure 4.3: FileWall policy chains with the scheduler and forwarder.

4.3.1 Policy Execution and Scheduling

A policy is a unit of FileWall processing. Its behavior is defined by handlers, which are invoked on receiving messages. FileWall policies may be linked as shown in Figure 4.3. FileWall manages fixed-sized input buffers for each policy. A policy interacts with FileWall through either reading/writing the access context, or through explicitly forwarding a message. FileWall places the forwarded messages in the input buffer of the next policy in the chain.

Two FileWall components, *Scheduler* and *Forwarder* (shaded boxes in Figure 4.3), enable policy chains. The forwarder is invoked by policies with a message, it classifies the message as a request or response, finds the next policy in the chain, and places a reference to the message in the next policy's buffer. If this buffer has no empty slots, the message is dropped. Messages dropped by FileWall are no different than messages dropped by the network and do not affect the correctness of the file system protocol.

Policy chains are organized as two ordered queues: the input and the output queue, which are mirror images of each other. That is, the policy at the head of the input queue is at the tail of the output queue. Intuitively, if a policy, E0, sees a request R0 before E1, then the response for R0 will be processed first by E1 and then by E0. The start (RECV) and the end (SEND) of the policy chains are fixed, and receive and send messages over the network, respectively. Policy chains are defined by the administrator using a plain-text configuration file that is parsed by FileWall. This file lists the policies in the order they must see the client request messages.

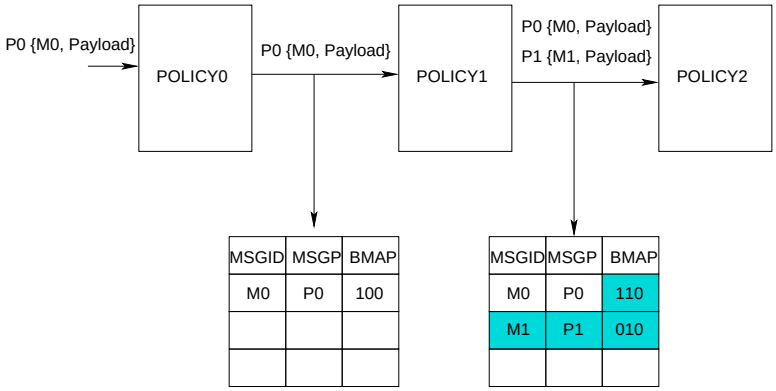

Figure 4.4: FileWall pending request map.

FileWall gains control of execution either asynchronously, on receiving a new message, or synchronously, on completion of a policy handler. On receiving a new message, FileWall simply inserts it in the input buffer of the *RECV* policy. Note that the messages are well formed file system messages. Reassembly and classification of messages are performed external to FileWall. No scheduling decisions are made during the execution of a policy.

FileWall schedules the policies synchronously by invoking the corresponding handler. Policies are scheduled round-robin, from RECV-to-SEND, in the chain specified during configuration and run to completion. More complicated scheduling algorithms, for example proportional fair share or lottery scheduling, can be used as replacements for our default implementation.

4.3.2 Matching Requests and Responses

FileWall maintains a global pending request map, which is initialized when client to server request messages are generated, either by receiving them from the network, or by policy handlers. The pending request map maintains the messages indexed by the unique transaction identifier, and a bitmap of all policies that have handled the message.

Figure 4.4 shows the flow of two messages through a policy chain. There are three policies in the system with identifiers 0, 1, and 2. The message P0 has the message identifier M0 and is seen by all three policies. As each policy forwards the message, the bit in the pending request map corresponding to this policy is set. Policy 1 generates a new message, which has not been seen by Policy 0. On creation, FileWall allocates a new entry in the pending request map and

sets the bit for policy 1. The modifications to the request map are shown as shaded boxes.

On receiving a server to client response message, the corresponding client to server request message is identified, is retrieved from the pending request map, and is presented to the policy handler along with the response message. Before handling the response, FileWall checks if its bit is set in the bitmap. If not, the message is passed to the next policy in the chain. If none of the successor policies in the response queue handle the message, the message is discarded.

FileWall ensure exactly once execution semantics through the bitmap. Response messages are handled only by policies that have previously handled the corresponding request message. Therefore, if an policy generates a new request message, none of its predecessors in the policy chain see the response. Similarly, if a response message is generated by a policy, the successors in the policy chain see neither the request nor the response message.

If the message is dropped, either by a policy or by the network, the pending requests occupy FileWall memory. FileWall defines a timeout that purges the request messages and clears the bitmap. All responses for this request are subsequently discarded.

4.3.3 Policy Specification

FileWall provides support for policy specification in two ways. First, a convenient macro-like language (FWL) for specifying new policies is provided. FWL policies are first translated into C or C++, then compiled into object code using the native compiler. However, developers (or power users) who wish low-level access can bypass the macro language completely and work directly with the native language policy implementations (C or C++). A base library of generic policies, which can be used to specify more complex policies (chains of policies), is provided.

A FileWall policy consists of *sections* that correspond to the respective processing in the policy. Administrators define new policies using the following format (in FWL):

```
@CONFIG::PolicyName {
    /* Declarations, Configuration variables*/ }
@CLItoSRV {
    /* Client-side message handling code */ }
@SRVtoCLI {
    /*Server-side message handling code */ }
```

A policy is specified, using FWL, in three sections: configuration (CONFIG), client-side

(CLItoSRV), and server-side (SRVtoCLI) that handle the corresponding FileWall processing. The CONFIG section contains all of the configuration variables and references to externally-defined functions or libraries. The CLItoSRV section is placed in the body of the client-side message handler and contains the code that is executed for any message received by FileWall from a client. The SRVtoCLI section is placed in the body of the server-side message handler and contains the code that is executed for any message received by FileWall from a server.

FWL provides two additional conveniences: attribute extraction and associative arrays. Attributes are encoded in packets in a host-independent data representation format (e.g., XDR). FileWall provides an extraction operator "@", which decodes the packets and presents the policy writer a reference to the decoded attribute.

Associative arrays are declared in the configuration section and are usually stored, as an in-memory hash table, in the access context of a policy. However, if a persistent array is required, it can be specified using a special operator, #, in the array declaration. To support this persistence, FileWall uses a transactional database on stable storage. Reading an element in a persistent associative array translates to a database lookup, and an assignment to a database update.

4.4 FileWall Policy Templates

In this section, we describe the FWL implementations of four network file system policies introduced in Section 4.2.4. Each policy represents a specific case of message transformation. In the accompanying descriptions, we present the sections of FWL code that illustrate the FileWall features we wish to highlight.

4.4.1 State Extraction and Update

The first example represents the class of FileWall policies that does not perform any message transformation. We illustrate attribute extraction and access context usage, without message transformation, by presenting the relevant FileWall implementation details from the message flow statistics gathering policy.

The access context generated by the statistics policy is counters associated with each client-server pair, and is stored in two non-persistent associative arrays: `reqstats` and `rspstats`. A record in `reqstats` or `rspstats` represents the count of an operation in the client-server request and server-client response streams respectively. The array index types are defined in FileWall and the NFS protocol specification. The following FWL code initializes these arrays.

```
@CONFIG::stats {
  int reqstats[IPFlowID][nfs3_ops];
  int rspstats[IPFlowID][nfs3_ops];
}
```

For each message, this policy extracts the flow identifier and operation number from the message, and updates the operation count stored in the `reqstats` or `rspstats` arrays accordingly. Also note that, while the operation number is not available in the server-client message, FileWall attribute extraction uses the matching client-server message to find the operation identifier. The following code performs these actions for client-server messages. The server-client code is identical, but substitutes the `rspstats` array for `reqstats`.

```
@CLItoSRV {
  op = op@$MSG;
  flowid = flowid@$MSG;
  reqstats[flowid][op]++;
}
```

While this policy is simplistic, it is the basic building block for any policy that builds up state by observing message flows between clients and servers. Similar policies can track access patterns, monitor server response times, etc., which can in turn be used for more sophisticated policies.

4.4.2 Attribute Transformation

This next example represents the class of policies that performs attribute transformation. We illustrate this by presenting the relevant FileWall implementation details from the file handle security policy.

This policy maintains two persistent maps as access context (forward and reverse file handle

to virtual file handle maps). The policy also uses an external function, `randfh()`, which is used to generate the random file handles in the SRVtoCLI code section (not shown).

For each client-server message, this policy extracts the flow identifier (`flowid`) and the virtual file handle (`vfh`) from the message. The policy performs a lookup in the access context (`fwdmap`), using `flowid` and `vfh`, to determine the real server file handle (`fh`). Once found, the client-server message is transformed to replace the `vfh` with the `fh`.

```
@CLItoSRV {
  flowid = flowid@$MSG;
  vfh = fhreq@$MSG;
  fh = fwdmap[flowid][vfh];
  fhreq@$MSG = fh;
  forward($MSG);
}
```

For each server-client message (code not shown), this policy extracts the flow identifier and the server file handle (`fh`), if it exists, from the message. If the `fh` is not contained in the message (e.g., ACCESS responses), the policy exits, forwarding the message. Otherwise, the policy performs a lookup in the access context (`revmap`), using `flowid` and `fh`, to determine the virtual file handle (`vfh`). If the `vfh` does not already exist, one is generated by the `randfh()` externally defined function and the new mapping is inserted into `fwdmap` and `revmap`. Once a `vfh` is obtained, the response is transformed to replace the `fh` with the `vfh`. For server-client READDIRPLUS response messages, any file handles returned by the server in the directory entry attribute listing must also be transformed.

The policy described above uses flow transformation, shown in Section 4.4.3, to generate a deny response to any request that does not contain a previously encountered virtual file handle. This prevents malformed or malicious requests from reaching the server leading to a potential crash or information disclosure.

4.4.3 Flow Transformation

The next example represents the class of policies that performs flow transformation. We illustrate this by presenting the relevant FileWall implementation details for the temporal access control policy.

For each client-server message, this policy extracts the current time from the environment. Between specified (configuration parameters) start (start_time) and end (end_time) times, the policy generates a deny message in response to client-server messages using the `discard()` and `denyrsp()` functions. In the code shown below, `denyrsp()` creates a new server-client message based on the attributes contained in the client-server message argument and `discard()` drops the message. The server-client message handler (not shown) simply forwards messages it receives.

```
@CLItoSRV {
  if(($curtime > start_time) &&
      ($curtime < stop_time)) {
    rsp = denyrsp($MSG);
    forward(rsp);
    discard($MSG);
  }
  else
    forward($MSG);
}
```

4.4.4 Flow Coordination

The final example represents the class of policies that performs flow coordination. We illustrate this by describing the relevant FileWall implementation details for the replication policy.

This policy uses a list of replicated file servers specified in the configuration section (not shown). Each server is assigned an identifier and a liveness bitmap is maintained in the environment. The failover policy generates virtual file handles for each client-server pair similar to the file handle security using forward and reverse maps.

Upon receiving a client-server message, a new message is generated for each live server in the server list. The new message is a copy of the incoming client-server message. The file handle is replaced similar to the file handle security policy. However, in this policy, each server has its own `vfh` to `fh` map and the message is forwarded to all live servers. This is equivalent to the creation of a new flow, for each client-replica server pair.

```
@CLItoSRV {
  flowid = flowid@$MSG;
  vfh = fhreq@$MSG;
```

```
  XID = XID@$MSG;
  bmap = 0;
  foreach saddr in replicas {
    server = servermaps[saddr];
    fh = server.fwdmap[flowid][vfh];
    newmsg = copymsg($MSG);
    fhreq@newmsg = fh;
    dstaddr@newmsg = s;
    forward(newmsg);
    bmap |= (1 << server.id);
  }
  pendingreqs[XID] = bmap;
  discard($MSG);
  ...
}
```

Upon receiving a server-client message (code not shown), the policy performs two main functions. First, it suppresses multiple server-client responses for the same client-server message. FileWall forwards one server-client message to the client after all server-client responses are received. However, this may easily be modified to only wait for the first server-client response or until a majority of server-client messages have been received. Second, it populates the per-server `vfh` to `fh` maps on receiving the file handle information, e.g., in LOOKUP and READDIRPLUS messages. This is equivalent to the aggregation of all client-replica server pair flows, into one flow per client.

4.5 Case Study: Role Based Access Control using FileWall

In this section, we describe the use of FileWall in implementing a role-based access control policy for network file systems. Through this case study, we illustrate the use of FileWall to offload *(i)* enforcement of an administrator defined access policy, and *(ii)* virtualization of the file system namespace to incorporate this policy without modifying the application interface at clients.

In recent years, role-based access control (RBAC) has emerged as a model for enforcing dynamic access control policies across a wide range of enterprise resources [96, 72]. There are three main benefits of using RBAC. First, RBAC models have been shown to be “policy

neutral", that is by using role hierarchies and constraints, a wide range of security policies can be expressed. Second, security administration is simplified by using roles to organize access privileges. For example, if a user moves to a new function in an organization, the user's role can simply be reassigned. In contrast, without RBAC, permissions on individual files would have to be updated. Third, by using constraints on the activation of user assigned roles, the principle of least privilege [170] can be enforced. In fact, today, RBAC models have matured to the point where they are prescribed as a generalized approach to access control.

While RBAC is attractive, there are several reasons for the reluctance of file system administrators to adopt RBAC. First, network file systems are performance critical and user applications expect their performance to be similar to local file systems. Second, due to the reliance on standardized file-system interfaces, users and applications expect and tolerate no modifications to existing behavior. Finally, deployability of any new mechanism requires that it does not modify either the clients or the servers. The servers may be proprietary and closed source precluding any modifications to them, and clients may refuse or be unable to execute any agents to support the modified access control protocols.

4.5.1 Example Access Control Policy

In the following, we define an RBAC security policy ($\mathcal{F}$) applied to a network file system. We use this policy to illustrate how FileWall applies RBAC concepts and use it as a running example. This example illustrates four basic principles of RBAC. First, the principle of least privilege is enforced. Specifically, a user accesses files at the lowest privilege level she is assigned to that is required for accessing an object. Second, dynamic escalation of roles is defined across user sessions. Third, delegation of roles is illustrated through a dynamic time-based policy, and fourth, per-file access control policies, defined by the user, are enforced.

Principal Definition: We assume a system with four users, $USERS$, who are assigned to three roles, $ROLES$, which have a partial order relationship defined by $HIER$. Permissions ($PERMS$) are derived from the NFSv4 ACL model and apply to each file system object (files, directories, links, etc) in the context of NFS operations defined for the object. Formally, the principal sets are defined as:

Role	**Users**
user	alice, bob*, charles*, root*
developer	bob, charles, root*
admin	root
threat	NULL

Table 4.2: User assignment for Role Based Access Control policy with FileWall.

	GETATTR	READ	WRITE	REMOVE
ALLOW	$user$	$user$	$developer$	$OWNER$
LOG	$admin$	$admin$	$admin$	$admin$
ALARM	$threat$	$threat$	$threat$	$threat$

Table 4.3: Subset of the FileWall access control matrix

$$USERS \leftarrow \{alice, bob, charles, root\}$$

$$ROLES \leftarrow \{user, developer, threat, admin\}$$

$$HIER \leftarrow \{user \leq developer \leq admin, threat\}$$

$$PERMS \leftarrow \{ALLOW, DENY, LOG, ALARM\}$$

User Assignment: User assignment defines the mapping between users and active roles. In a hierarchical RBAC system, the users are assigned to all roles below the initial assignment through inheritance. The default role assignments for $\mathcal{F}$ are shown in Table 4.2. The users marked by a * in the table show the inherited role memberships.

While the user assignment provides a list of all available roles, the active roles are determined for each session subject to dynamic separation of duty constraints. $\mathcal{F}$ includes a dynamic separation of duty constraint, which states that a user cannot acquire the $admin$ role in a session with any other role ($user$, $developer$, or $threat$) active.

Permission Assignment: Permission assignment maps the roles to the set of permissions for operations on an object. The permission assignment is statically defined by the policy, and may be dynamically updated by the owner of the object.

Table 4.3 shows a subset of the permission assignment for a file. Here, the $user$ role

is allowed `GETATTR` and `READ` operations, while `WRITE` operation is permitted only for the $developer$ role. The special tag $OWNER$ is derived from the file attributes and only the owner (or the admin) is allowed to remove the file. If the operation is allowed, FileWall additionally evaluates the `LOG` permission and logs the operation if required. Also, if the operation is denied, the `ALARM` permission is evaluated to possibly generate an alarm for the system administrator.

The members of the role $user$ can only access the files owned by them. Members of the role $developer$ derive all the user properties and can, additionaly, access all files owned by users in the developer role. Administrators belong to the $admin$ role and have unrestricted access to the file system. However, all administrator actions that modify the file system are logged. Finally, a user can be made a part of the $threat$ role, based on a dynamic rule. As an illustration, all users with role $user$ who try to escalate their privileges to the role $admin$ are threats and all updates made by members of this role are logged.

Session Management: A *session* determines the set of active roles assigned to a user. All role assignments, as well as, dynamic separation of duty constraints are evaluated within a session. While the user may acquire multiple roles, the set of active roles within a session is fixed. If the user wishes to modify this set of roles, the old session must be terminated and a new session created with the new roles.

In our example policy, on session initiation, all users are part of the $user$ role. If a user is a member of the $developer$ role, she additionally retains the developer privileges. Modification of the active roles in a session can be *explicit* or *implicit*. For explicit modification of active roles, the user must initiate modifications to the role set through FileWall. In $\mathcal{F}$, the administrator delegates his role to members of the developer group during the non-work hours. That is, the members of the developer group can acquire administrator privileges between 5PM and 9AM. This delegation is often desired but is seldom implemented without explicit RBAC support. The escalation of $developer$ to $admin$ is performed explicitly, and a new session is created.

$\mathcal{F}$ also supports an implicit session update using the environment. When the access control system determines a user is a threat, based on previously specified access patterns or attempted accesses to sensitive files, FileWall terminates the active user session and initiates a new session with the $threat$ role. This update does not involve the user and prohibits access to the file

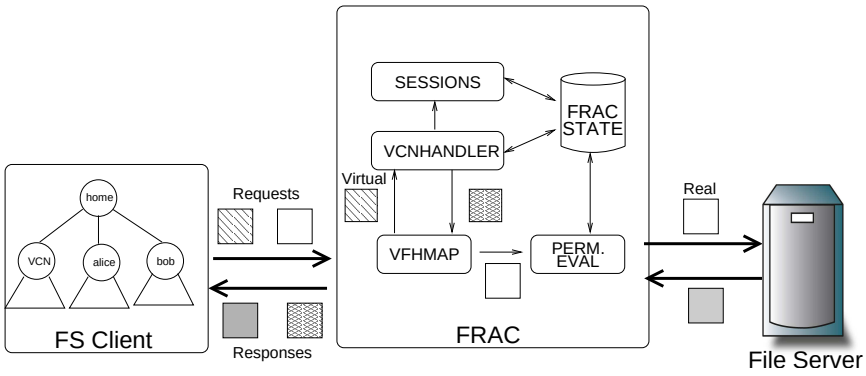

Figure 4.5: Overview of Role Based Access Control implementation with FileWall.

system including any role updates. Administrator intervention is required to re-enable the user access.

Figure 4.5 shows the flow of requests and responses through FileWall. The client generates file system requests for both real (solid boxes) and virtual (patterned boxes) objects. The requests for virtual objects are handled by a special VCN handler, which generates the file system responses. Requests for real objects are forwarded to the file server, which generates the responses sent to the client. All requests are subject to permission evaluation and deny responses are generated for all requests that are denied.

Policy Specification and Update: Access control policies must be defined before any accesses to the system are performed by clients. Therefore, to bootstrap the system, FileWall requires administrators to provide an initial policy description during startup. This initial description contains the principal definitions and access control constraints. Several policy specification languages have previously been proposed for RBAC. We use XACML [8], a standardized RBAC specification language that provides not only the core RBAC specification but also allows us to define temporal and dynamic access control policies.

During execution, FileWall provides interfaces for administrators to update access control policies, and for users to manage their sessions and permissions for the files they own. These interfaces are realized using a virtual namespace, called the Virtual Control Namespace *(VCN)*, contained within the existing file system namespace (see Section 4.5.2).

State Maintenance: FileWall maintains persistent state, which must be available through the

lifetime of the system and is essential for correct enforcement of RBAC. We identify two persistent state components for FileWall: *(i)* Policy definition, which includes user and role identifiers, role hierarchies, user assignment, and any separation of duty constraints. *(ii)* Permission assignment, which includes per-file access control matrix and owner information.

FileWall also maintains soft state, which is throw-away, and can be reconstructed from the persistent state and by observing the file system requests and responses. This state has two main components: *(i)* Session state, which consists of active user sessions and their currently active roles. FileWall assumes each user has at most one session on each machine. This creates a unique identifier for a user session, which is then used to identify the active roles associated with the session, and *(ii)* a Virtual File Handle mapping, which represents an association between FileWall generated client-visible file identifiers, distinct from server-defined file identifiers.

Algorithm 2 FileWall algorithm for implementing RBAC.

1: $sid \leftarrow SESSIONS[MSG.IP][MSG.UID]$;
2: $roles_{sid} \leftarrow ROLES[sid]$;
3: $vfh \leftarrow MSG.FH$;
4: $vfhentry \leftarrow VFHMAP[vfh]$;
5: $fh \leftarrow vfhentry.fh$;
6: **if** $fh = NULL$ **then**
7: $vcnhandler(MSG, vfhentry.shadow)$;
8: **return**
9: **end if**
10: $shadow \leftarrow SHADOWFILES[vfhentry.shadow]$;
11: $ownerid \leftarrow shadow.ownerid$;
12: $perms \leftarrow shadow.perms$;
13: $op \leftarrow MSG.OP$;
14: $minrole \leftarrow perms[op]$;
15: **for all** $r \in roles_{sid}$ **do**
16: $found \leftarrow DFS(HIER, r, minrole)$
17: **if** $found =$ TRUE **then**
18: $MSG' \leftarrow MSG$
19: $MSG'.UID \leftarrow ownerid$;
20: $MSG'.FH \leftarrow fh$;
21: $forward(MSG')$;
22: **return**
23: **end if**
24: **end for**
25: $deny(MSG)$;
26: **return**

Permission Evaluation: FileWall maintains the state required for making access control decisions and generates a boolean $ALLOW$ or $DENY$ verdict for each file system request. The requests that are allowed are forwarded to the file server. For requests that are denied, FileWall generates an appropriate deny response *without* contacting the server.

Algorithm 2 shows the algorithm used by FileWall for permission evaluation. The algorithm uses the incoming request message as input and uses attributes contained in the message to identify the set of active roles (Lines 1-2). These roles are maintained in order of increasing privilege level. It then extracts the virtual file handle (vfh) from the request message and uses this VFH to obtain the structure containing the real file handle (fh). If this mapping structure does not contain a FH, then the VFH refers to a VCN object, which is handled by the $vcnhandler$ (Line 6-9).

For the case of real file system objects, the algorithm obtains the corresponding access control information (shadow file) (Line 10). The least privileged role required to perform *op* is identified from the permission map. Finally, a depth first search (DFS) is performed on the role hierarchy defined by the policy starting at the current role (Line 16). If the target role (defined by the permission) is found during the DFS, the message is allowed. Otherwise, if all active roles are exhausted, the request is denied.

While forwarding the message, FileWall replaces the UID and VFH in the request with the $ownerid$ and real fh respectively, and the message is forwarded to the server (Line 21). This ensures that the operation is performed as the owner of the file and the base file system permissions are still enforced. All other messages are denied (Line 25).

FileWall uses four tables to maintain the state required for permission evaluation. The $SESSIONS$ table maintains a mapping of the user identifier and the IP address to a local session identifier. This identifier indexes into the $ROLES$ table, which maintains the set of active roles for each active session. The $VFHMAP$ stores a mapping from the VFH to the 2-tuple $< fh, shadowfh >$. Finally, the $SHADOWFILES$ map maintains the permission assignment and the file owner information for each file indexed by the file identifier (file handle).

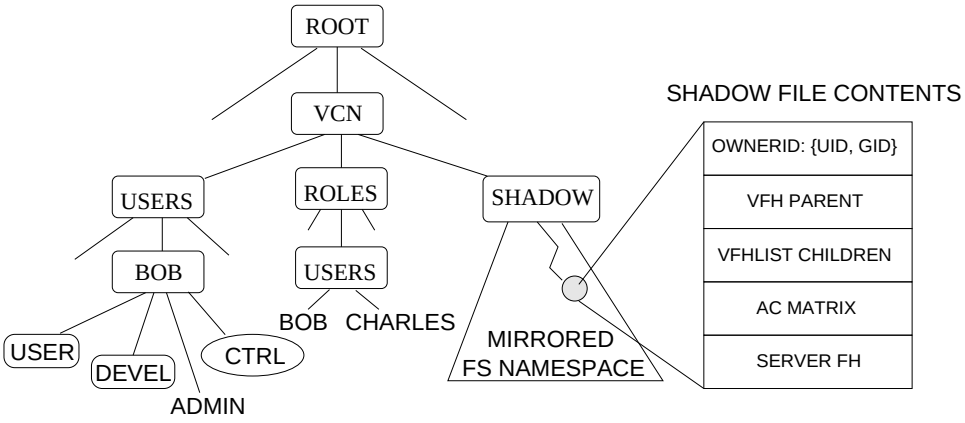

Figure 4.6: Subset of the FileWall Virtual Control Namespace.

4.5.2 Virtual Control Namespace

FileWall defines a virtual namespace, called the Virtual Control Namespace (VCN), for session management and dynamic permission assignment. The VCN is initialized when a file system is mounted over NFS. All operations on objects in the VCN are sent using the standard file system protocol and are interpreted as commands at FileWall. Through the VCN, clients can interact with FileWall over the well known file system interface using a familiar set of tools. Therefore, the VCN eliminates the need to run separate software at the clients or servers. It is important to note that user authentication is performed externally and is not the focus of FileWall. However, authentication mechanisms can be easily incorporated using the VCN.

The functionality of the VCN is similar to the well known `/proc` file system in Linux. FileWall handlers are invoked on receiving a file system request for virtual objects. For read-only requests, for example, READ, READDIR, GETATTR, etc., the handlers query the FileWall state and generate the file contents dynamically. The query operation presents a snapshot of the FileWall state at the clients. The WRITE operations update the FileWall state and are used to modify active roles, permission assignment, and other session state defined by the system-wide policy. No metadata update operations, e.g. SETATTR, are supported on virtual files to restrict the update interface.

Figure 4.6 shows a subset of the VCN for the example in Section 4.5.1. The VCN has two main components: Session namespace, and Shadow namespace. The directories `users` and `roles` make up the session namespace and provide an interface to control the user sessions

and user assignments, respectively. In the session namespace, a control file, `ctrl`, provides the update interface for each directory in the VCN. Users write the identifiers, of the new roles they want activated, to this file. If successful, the new active roles are instantiated and any modifications to the available roles are reflected in the contents of the directory. If unsuccessful, a permission denied message is returned to the client and the session remains unmodified.

The `shadow` namespace mirrors the file system. The names for objects in the `shadow` directory are derived directly from the files they represent in the primary file system. The shadow files provide an interface to query and update the permission assignments for each file. Effective permissions are determined using the owner permissions, specified as file attributes (rwx) of real files, and using the policy defined role-based permissions.

The initial permission assignment identifies the owner of each file and assigns her least privileged role the rights defined by the file permission bits. For example, if a file owned by alice has permissions $rw-$, FileWall identifies the least privileged role for alice ($user$), and provides that role with read and write privileges to the file. To update permissions for a file, users write the updated permission assignment to the corresponding file in the shadow namespace. If successful, the modifications are reflected in the contents of the shadow file. Owners can also update the permissions on the files through the `chmod` interface. For such updates, the effective permissions are re-evaluated and the contents of the corresponding shadow files are also updated.

Figure 4.6 shows the contents of a shadow file. For a shadow directory, the file handle information and shadow file identifiers of each of its children are also stored. This enables the VCN handlers for directory listing to construct a virtual namespace hierarchy that mirrors the real file system namespace.

The shadow file hierarchy is built lazily, using the file system requests to populate itself. FileWall generates shadow files by extracting records from the responses to directory listing and file lookup client requests (READDIR, LOOKUP). The root directory is special since it is identified during the mount protocol and the VCN root directory appears as an immediate child of the root directory. FileWall adds an additional record for the VCN for any root directory listing request. The state required to generate the shadow files includes owner information, server specified file identifiers, and the access control matrix. While the owner information and

the file handles can be regenerated, the access control matrix must be available across FileWall restarts as it represents the permission assignments that may have been updated by the owners.

All objects in the VCN are virtual and there is no file (or directory) at the server that stores the content of these objects. Virtual File Handles (VFH) are file identifiers generated by FileWall to transparently implement the VCN. Each virtual object in the VCN has a unique VFH. To prevent collisions between VFHs and file server specified FHs, FileWall generates a VFH even for real file system objects. FileWall maintains the VFH to FH mapping in its local state as a file handle mapping table (VFHMap). For real objects, FileWall replaces the VFH with the corresponding FH when forwarding a request message, and the FH with the VFH for a response message. For virtual objects, the appropriate FileWall handler is invoked. VFHs are also used to implement the write-once semantics for virtual objects by generating a new VFH every time the virtual object is updated.

4.6 Implementation

We implemented FileWall in the Click modular router framework [103] as an external user-level package. Click element classes define input and output ports, which represent connections to other elements in the router. Elements are classified as *push* and *pull*. *Push* elements are invoked in context of a network packet and run to completion. Optionally, these elements output a packet to one or more ports through a synchronous call. *Pull* elements, on the other hand, wait for packets to be generated by upstream elements and execute when packets are available. Explicit queues are used to buffer packets between push and pull elements.

Figure 4.7 shows the Click configuration that realizes FileWall with two policies: *(i)* a primitive policy (*FWExt0*), and *(ii)* a composite policy (*FWExt1/FWExt2*). Each policy is implemented as an autonomous Click push element, which runs to completion. The FileWall scheduler is a pull element and chooses the policy to execute. As shown in the figure, *FileWallSched* interposes between all FileWall extensions to schedule and maintain global state across extensions.

We implemented the access context using an open-source database – BerkeleyDB [184]. This database shares a process's address space allowing direct function calls to be made to

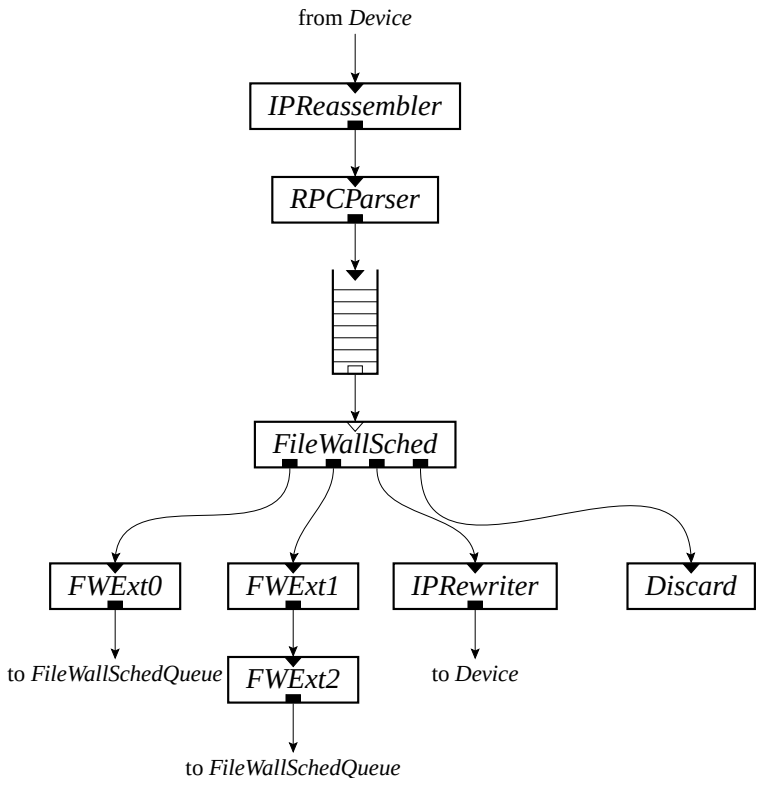

Figure 4.7: FileWall implementation using the Click modular router.

the DB. We use the default DB configuration where the data is stored as B-Trees. Separate databases are created for each extension in the system to maintain private access context.

The FileWall templates define the skeleton of the classes and the initialization code required to construct a Click element class. The FWL compiler is implemented using treecc [212], which converts input files in the treecc syntax into source code that permits creating and walking abstract syntax trees. We use treecc along with standard compiler generation tools lex and yacc to generate Click element code from the FWL code.

4.7 Evaluation

In this section, we present an evaluation of our system. Our goal in this evaluation is to measure FileWall *overheads* and to characterize FileWall *behavior*, under varying network conditions and workloads.

4.7.1 Setup

In our experimental setup, all systems are Dell Poweredge 2600 SMP, systems with two 2.4GHz Intel Xeon II CPUs, with 2GB of RAM, and 36GB 15K RPM SCSI drives. All systems run the Fedora Core 3 distribution with a Linux-2.6.16 kernel and are connected using a Gigabit Ethernet switch. The average round-trip time between any two hosts on the switch is $30\mu s$. FileWall is configured to interpose on all NFS requests and responses. Unless otherwise specified, all experiments with FileWall use the filehandle security extension described in Sections 4.2.4 and 4.4.2.

Microbenchmark: To study the behavior of the file system, with and without FileWall, we developed our own microbenchmark. This benchmark is an RPC client and issues NFS requests *without* relying on the client file system interface. Using this benchmark eliminates the noise due to the client buffer cache and other file system optimizations, and allows fine-grained measurements to be collected. The benchmark measures the CPU cycles between a request and the corresponding response.

4.7.2 Latency

In this section, we study the effect of adding FileWall to the network file system message path on the client observed latency. To isolate the FileWall overheads, we study three systems: *(i)* Tunnel, which is a pass-through network tunnel implemented at the user level and does not include FileWall, *(ii)* Kernel Tunnel, which is identical to Tunnel implemented in the kernel, and *(iii)* FileWall, which is built on top of the Tunnel system with the attribute mapping extension described in Section 4.4.2. As a baseline, we present the latency of the default NFS protocol.

Interposition Overheads: The latency of an operation includes the network delays and the server processing overheads. Interposition overheads manifest themselves as increased latency for each network file system message. Increased latency has three components: *(i)* the packet capture and injection latency, *(ii)* the FileWall processing latency, and *(iii)* the extension latency. Formally, the base NFS request-response latency and the latency with interposition is given by

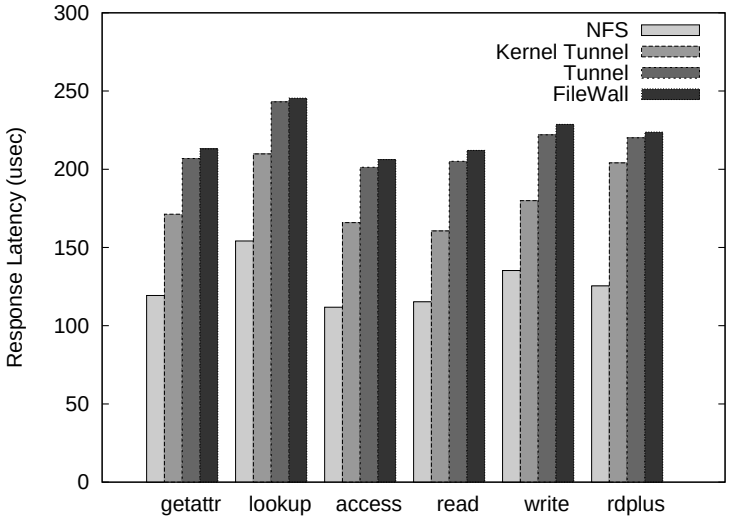

Figure 4.8: FileWall interposition overheads.

$$L_{NFS} = 2 \times D_{CS} + T_S \tag{4.1}$$

$$L_{Tun} = 2 \times (D_{CF} + T_P + D_{FS}) + T_S \tag{4.2}$$

$$L_{FileWall} = 2 \times (D_{CF} + T_P + T_F + D_{FS}) + T_S \tag{4.3}$$

where L represents the client observed latency, D represents the one-way network delay, and T is the processing overhead of each component. The subscripts C, S, and F represent the client, server, and FileWall respectively. T_P is the per-message processing overhead due to the packet capture and injection.

Our first goal is to isolate the overheads imposed by FileWall (T_F) and per-message processing (T_P), eliminating the impact of network delays. From Eqs. 4.2 and 4.3, we observe that the difference between $L_{FileWall}$ and L_{Tun} represents the FileWall and extension overheads.

$$T_F = \frac{L_{FileWall} - L_{Tun}}{2} \tag{4.4}$$

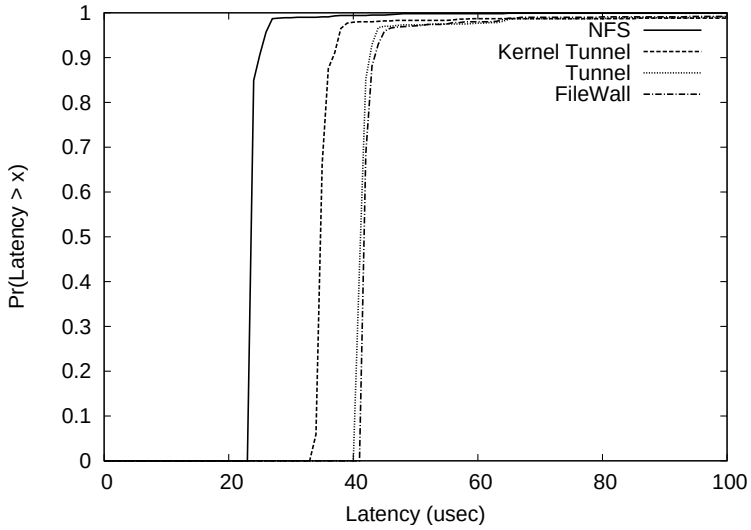

Figure 4.9: Distribution of response latency for NFS with interposition.

Figure 4.8 shows the client observed latency for different NFS operations. For clarity, we show only the most common operations, as reported by various file system workload studies [68] in the figure. We observed similar results for other extensions as well as for the NFS operations not shown. In the figure, each group of bars has 4 members, base NFS, tunnel, kernel tunnel, and FileWall. The height of each bar shows the average response latency for 1000 instances of the call.

We observe that FileWall imposes minimal overhead compared to Tunnel overhead, as shown in Eq. 4.4. As expected, the kernel-tunnel is more efficient than the user level implementation, but the difference between them is less than $60\mu s$. Figure 4.9 shows the cumulative distribution function (CDF) for all READDIRPLUS calls to illustrate that these overheads do not vary significantly across measurements and are largely fixed.

To isolate per-message processing overheads (T_P), we observe from Eqs. 4.1 and 4.2 that

$$T_P = \frac{L_{Tun} - L_{NFS}}{2} - (D_{CF} + D_{FS} - D_{CS}) \tag{4.5}$$

Therefore, if $(D_{CF} + D_{FS} - D_{CS}) \rightarrow 0$, we can identify the interposition overheads using

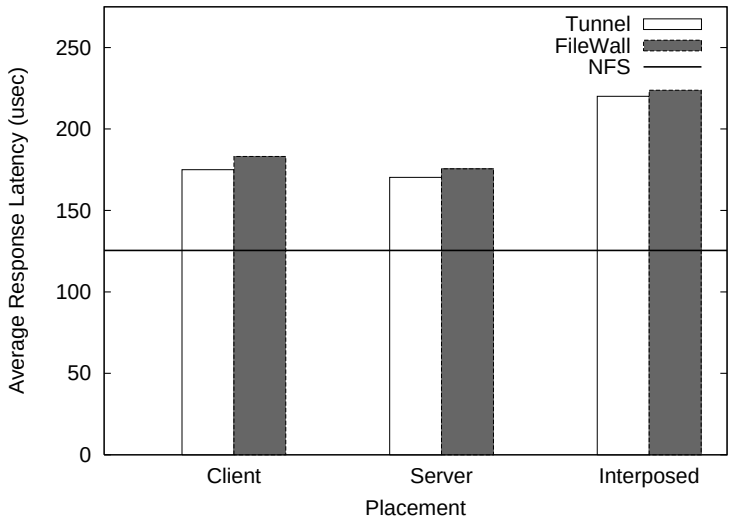

Figure 4.10: Overheads of placing FileWall at client, server, and interposed

operation latencies over the two systems. To accomplish this, we instantiate the network tunnel at the client, where $(D_{CF} \ll D_{FS})$ and $D_{FS} \approx D_{CS}$, and at the server $(D_{CF} \gg D_{FS})$ and $D_{CF} \approx D_{CS}$.

Figure 4.10 shows the results for our microbenchmark for the READDIRPLUS call. The figure shows the average time between the call and its response at the client when the tunnel and FileWall are instantiated at the client, server, and on a separate machine on the network. As a baseline, we include the latency for the default NFS.

We observe that FileWall instantiation on a separate node (our model) performs well. The interposition overheads (T_P) are within $100\mu sec$ and can be improved further using an in-kernel implementation.

Network Delays: To study the effect of network delays on client-observed latency with a metadata intensive workload mix, we vary the network delays and study the effect of this variation using Postmark.

Postmark [97] is a synthetic benchmark that measures file system performance with a workload composed of many short-lived, relatively small files. Postmark workloads are characterized by a mix of metadata intensive operations. The benchmark begins by creating a pool of files, performs a sequence of transactions, and concludes by deleting all of the files created.

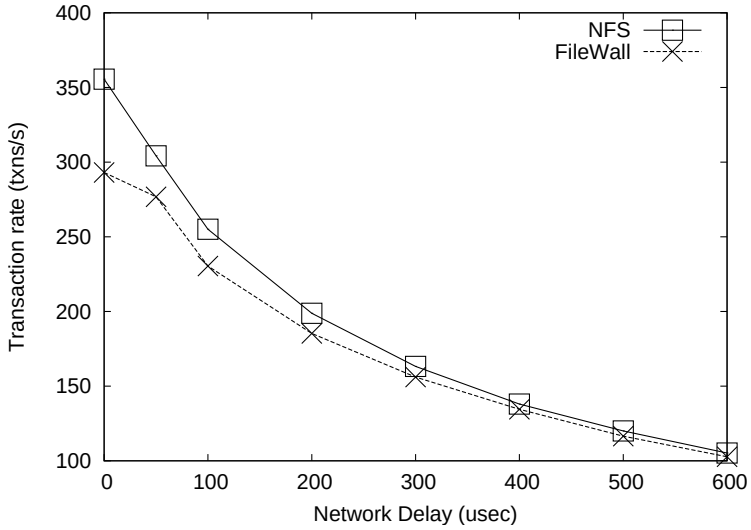

Figure 4.11: NFS metadata performance with varying network delay

Each transaction consists of two operations: a randomly chosen CREATE or DELETE, paired with a randomly chosen READ or WRITE.

In our experiments, we use 8KB block sizes for read and write operations for Postmark. The initial file set consists of 5,000 files with sizes distributed randomly between 1KB and 16KB. For each run of the benchmark, we perform 20,000 transactions and report the transaction rate.

Figure 4.11 shows the Postmark results for varying the one-way network delay between the client and the server (D_{CS}) for base NFS and the client-FileWall delay (D_{CF}) for FileWall. The FileWall-server link is unmodified.

We observe that for low delays ($< 100\mu s$), FileWall and the base NFS differ significantly in the supported transaction rate. However, for networks with larger delays, the difference between FileWall and NFS performance is minimal. FileWall introduces delays of the order of 10s of μs, therefore, for comparable network delays, the performance of latency sensitive operations is greatly affected. However, these delays are hidden when the network delays dominate the overall delay between the client and the server and the operations are pipelined.

For typical deployments, file servers are physically separated from clients and delays of up to $300\mu s$ are common. Therefore, in realistic scenarios, we believe that FileWall will not affect performance and will be transparent to the clients.

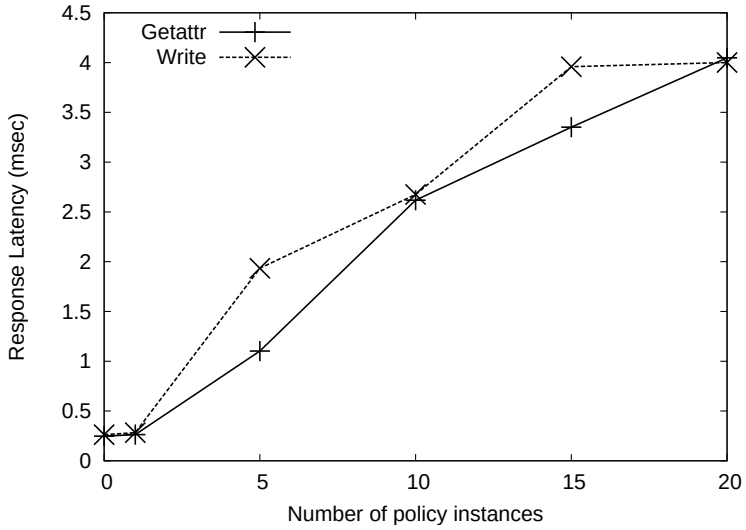

Figure 4.12: FileWall scalability with increasing number of extensions.

Scalability: In this experiment, our goal is to study the effect of the processing overheads due to FileWall extensions (T_F in Eq. 4.3). As we increase the number of extensions, we expect this overhead to increase. However, this increase should not be exponential and ideally should be linear.

The number of FileWall extensions vary across deployments. Therefore, it is difficult to understand the client perceived performance as the number of extensions increases. We define a synthetic extension that captures the tasks common across a wide range of extensions and vary the number of instances of this extension to study FileWall behavior.

Each extension performs the following tasks: On a request message, it stores a fixed number (20) keys, each of size 50 bytes in the access context (DB insert). On a response, it looks up all the keys inserted by the corresponding request and deletes them (DB lookup and delete). Since DB access is the most CPU intensive task performed by a FileWall extension, our results capture the expected behavior. However, these results may vary with different extensions.

Figure 4.12 shows the average response latency for our benchmark as the number of extensions is varied. The two curves are for GETATTR and WRITE requests, which illustrate the FileWall performance for metadata and data operations respectively. We observe that FileWall overheads increase linearly in the worst case and, even with 20 extensions, the response time is

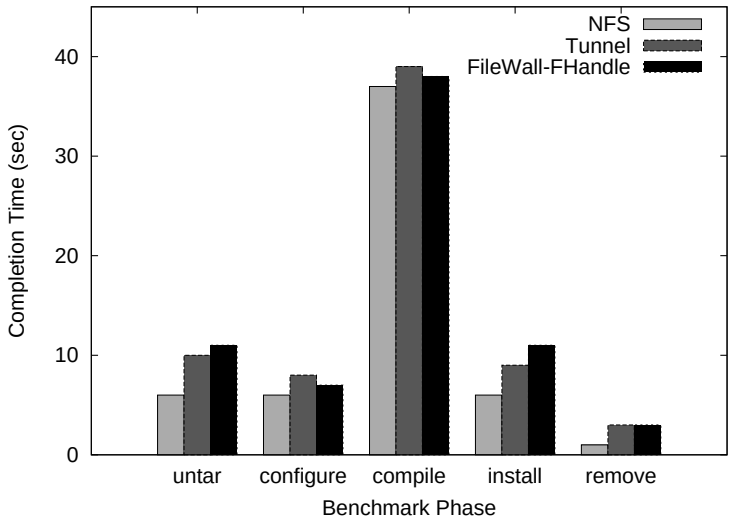

Figure 4.13: FileWall performance for emacs compilation.

within 5 ms.

4.7.3 Real Workloads

Emacs Compilation: In the following, we compare the performance of FileWall with the file handle security extension, default NFS, and a pass-through tunnel, for a multi-stage software build similar to a modified Andrew Benchmark [91]. We measure the time taken to *untar*, *configure*, *compile*, *install*, and *remove* an Emacs 20.7 distribution.

The Emacs distribution size is 76MB, which reduces to 20MB when tarred and gzipped. It contains 43 directories and 76,400 files. The compilation performs a large number of read, write, lookup, create, and remove operations. At the end of the compilation, the total number of files in the directory is 101,644 and the size is 95MB. We performed one run of the benchmark to load the compiler binaries and associated libraries that are external to the file system under test. We discarded this result and performed ten further runs. Between each run of the benchmark, we unmounted the file system and mounted it to start with a cold cache on the file system under test at the client.

Figure 4.13 shows the time taken for each phase of the Emacs compilation benchmark from left to right. The bars in each group are NFS, Pass-through tunnel, and FileWall with the file

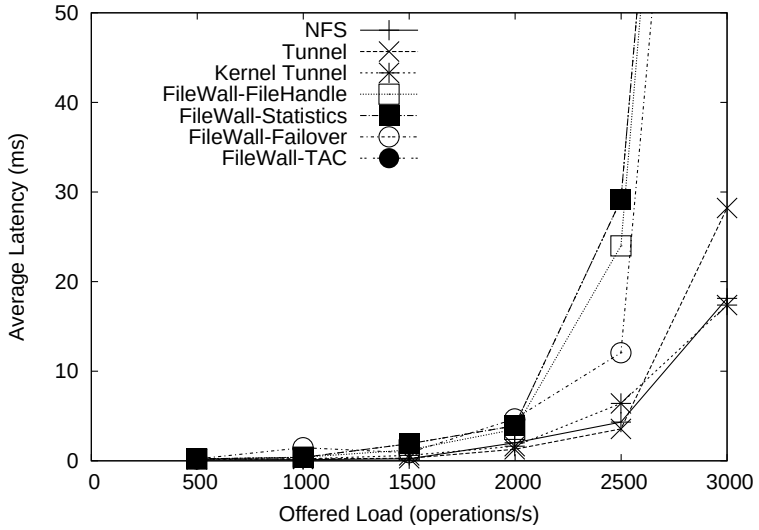

Figure 4.14: FileWall throughput vs. latency

handle security extension. Experiments with other extensions described in the paper yielded similar results and are omitted for clarity.

We observe that FileWall imposes a modest overhead of around 12% for the sum of all phases of the benchmark. The most expensive data intensive phases have small $(< 10\%)$ overheads. However, the metadata intensive untar, install, and remove phases show significant degradation. These results are pessimistic for FileWall due to the extremely low network delay without interposition $(30\mu s)$. Recall from our microbenchmark that interposition imposes tens of μs of additional latency, which is comparable to the network delays in our environment. These overheads reflect in the extra time required for the metadata intensive phases of the benchmark. For networks with larger delays, the performance of all three – NFS, tunnel, and FileWall is identical.

Fstress: Fstress [9] is a synthetic, self-scaling benchmark that measures server scalability. We use the canned SpecSFS97-like workload distributed with Fstress, which performs random read-write accesses with file sizes varying from 1KB–1MB. The workload is characterized by a large number of directories with thousands of files in each directory. The size of the file set accessed by Fstress is adjusted to reflect the offered load on the system. The operation distribution is identical to that defined by SPECsfs97, which in turn is based on a survey of file

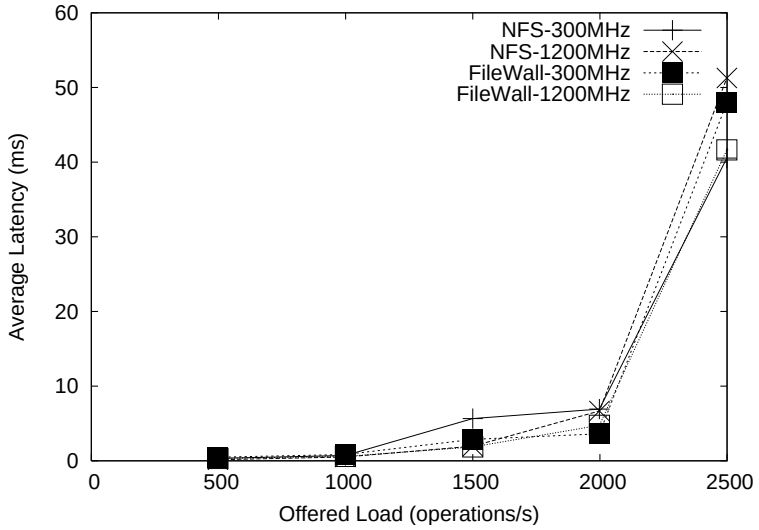

Figure 4.15: FileWall overheads under varying server CPU speeds.

set distributions and workloads on one thousand NFS servers.

Fstress increases the offered load by issuing NFS requests according to the workload mix. The metric used is the latency of these operations. For an overloaded server, the client observed latencies grow rapidly, whereas for an underloaded server, these latencies are small. For our experiments, we use three load generators (clients) that have the configurations described in Section 4.7.1.

Figure 4.14 shows the results from the Fstress benchmark. We observe that FileWall latencies are comparable to both the default NFS and the network tunnels (user and kernel) up to an offered load of 2,500 requests/s. The systems diverge rapidly beyond this point. However, in all cases, the NFS server is overloaded at around 3,000 requests/s. The sharp increase in observed latency with FileWall is due to the CPU at FileWall being overloaded. At overload, the small processing overheads imposed by FileWall become significant and the performance drops. However, the overheads are small (around 15%) and the enhanced functionality provided in exchange by FileWall make them acceptable.

FileWall does not necessarily saturate before the NFS server. The impact of FileWall under heavy load is due to the relative performance between the FileWall machine and the NFS server machine. This is illustrated in Figure 4.15. For this experiment we run Fstress, while varying

the CPU speed of the NFS server. We use the Intel Speedstep voltage scaling to reduce the CPU speed to 300MHz and 1200MHz, while maintaining all other system parameters constant. The curves in the Figure represent the base NFS and FileWall performance for the different NFS server CPU speeds. In all cases, we observe that the performance is similar with and without FileWall, and the NFS server saturates beyond 2000 requests/s. We conclude that given sufficient resources relative to the NFS server, FileWall does not impose significant overheads even under heavy workloads.

4.8 Discussion

FileWall is meant for evolutionary extensions of file systems. Large scale modifications, for example protocol upgrades, must be performed at the end-hosts. However, we believe such drastic upgrades are rare. The reluctance of administrators to upgrade existing infrastructure, especially file systems, is evident in the slow adoption of the new NFSv4 standard, which has been available for several years now.

In this chapter, we have described a proof of concept implementation for the NFSv3 protocol. The choice of this protocol is due to its popularity, and due to its limited support for callbacks and per-file open and close messages. This allows us to study the limitations of implementing access policies with the least support from the protocol. However, this is not a fundamental limitation of our system. Over the years, most systems which extend NFS functionality have introduced callbacks [92, 101] and support for OPEN and CLOSE [77, 131]. With such support, we can easily implement extensions to provide novel consistency semantics, atomic updates, application aware caching, etc., with FileWall.

FileWall provides an interface for administrators to define file system access policies. However, this interface is protocol dependent. That is, the administrator must have knowledge of and FileWall must understand the underlying file system protocols. FileWall is a first step towards the ultimate goal of a policy specification and enforcement platform, where administrators can describe system-wide policies, and the low-level implementation of the policy is generated automatically. In the future, we plan to design a protocol-independent high-level language, which

eases extension specification and enables verification and debugging of primitive and composite extensions. Such a language could also be used to enforce and extract invariants, which can be used by administrators for writing more robust and effective FileWall policies.

A limitation of our system is the limited support for fault-tolerance and reliability. Since we introduce an additional host in the network file system message path, its failure would lead to loss of service even when both clients and servers are active. However, we believe we can introduce a primary-backup failover scheme for FileWall. Since FileWall state is built up during execution by observing messages, a backup system that snoops all traffic entering and leaving FileWall can build up its own copy of the state. For stateful protocols, e.g., TCP, we can restore the state associated with live connections terminated or initiated at FileWall using connection migration protocols that have been proposed before in [197, 188].

4.9 Related Work

FileWall is related to a variety of work in the areas of *(i)* Distributed and Extensible File Systems and *(ii)* Composable Network Processing. The following section is a survey of the work related to FileWall in these areas.

Composable Network Processing: FileWall is inspired by packet filters [35], network address translators, and firewalls, which implement network access policies by interposing on network traffic and mapping private addresses to public Internet addresses.

FileWall interposes on file system requests to implement *file* access policies. However, unlike the packet processing systems, FileWall maintains a chain of policies, each with a private access context, which is maintained in persistent storage. It supports attribute extraction from messages and provides a framework for network file system policy specification.

The x-kernel [93] is a framework for implementing and composing network protocols. An x-kernel configuration is a graph of processing nodes, and packets are passed between nodes through function calls. Unlike FileWall, where policies are composed as chains and file system attributes are transformed during processing, the x-kernel nodes represent network protocols in a protocol stack. the x-kernel nodes are arranged in an acyclic graph and the inter-node communication is more complex than a simple function call.

The Scout Operating System [126] is an OS architecture where *paths* defined by a sequence of processing nodes traversed by a network packet are made explicit. Packets are classified into the correct path as early as possible, so that packets are treated differently as soon as they arrive on the host. For example, ethernet packets containing video data are treated differently. Each path in Scout is executed as a separate thread. In addition, Scout paths support different kinds of inter-node communication beyond a simple packet flow. In contrast, FileWall handles only file system messages and all messages flow through the same policy chain.

FileWall is implemented within the Click [103] modular router framework. In contrast to the above, which process network packets, FileWall processing is in context of file system messages, which may be composed of multiple packets. FileWall also supports persistent state using a database, which is not required for a router.

Distributed and Extensible File Systems: Several proposals for extending local file systems by interposing on the request path have been proposed [220, 216]. These systems interpose between the file systems and transform the vnode operations to enhance functionality. Such systems have been used to trace file system calls [16], build versioning file systems [129], virtualized namespaces [149, 160], virus protection [121], etc. Unlike the above, FileWall extends network file systems and these extensions are implemented external to both clients and servers through message transformation.

Network storage systems have been extended for functional decomposition [11], enhanced security [119], saving bandwidth [130, 13], repair and rollback [224], extended access control [87, 120], and using interposing proxies. These systems interpose on the client-server path to implement a solution for a specific problem. In contrast, FileWall provides a platform for extending file systems and builds mechanisms, which go beyond a specific problem instance. In fact, each of the above extensions can be implemented using a FileWall policy.

FiST [220] is a stackable template language that shares a similar structure with FWL. FiST templates are used in conjunction with a base file system to extend the operating system vnode interface. Unlike FiST that operates on local file system calls and a stackable vnode interface, FWL operates on file system messages to implement extensions.

Extensible Policies: SPIN [183] and VINO [176] are extensible operating systems that allow

applications to make policy decisions and modify operating system interfaces by safely downloading extensions into the kernel. SPIN ensures safety using a type-safe language Modula-3, while VINO implements mechanisms for fault isolation. Unlike the extensible OS, FileWall does not modify OS interfaces, targets network file system policies, and implements the policies external to the client and server systems.

Exokernel [70] exports fine-grained hardware services, for example, TLB management, directly to applications. Exokernel provides no abstractions beyond those minimally provided by the hardware and allows libraries to implement OS policies. FileWall policies are similar in spirit to the Exokernel stable storage system XN. To implement file systems, XN allows users to define on disk data structures and methods to implement them with libraries (libFSes). FileWall uses the minimal abstractions provided by the message streams to implement network file system policies and allows administrators to define policies.

Namespaces and Name Resolution: FileWall incorporates file access policies within existing file system protocols by constructing virtual namespaces, which are a subset of the file system namespace. The use of client namespaces is motivated by seminal work on namespaces [109, 148, 142], each of which recognizes the importance of names as a unifying feature in all distributed systems.

The Andrew File System [92] introduced the concept of global namespaces in file systems. Several systems have explored providing a global namespace for clustered file systems [174, 78, 205], distributed file systems [12, 143, 27], and more recently to federated file systems [3, 204, 114]. While the focus of all of the above is organizing collections of objects or file servers into a global namespace, FileWall focuses on network file system *policies*. Unlike the above, we target file systems where file servers and clients are in the same local area network, and the entire file system is located at the file server.

Plan9 [149] uses a hierarchical file system to represent every resource in the system. A user or a process constructs a private namespace view by connecting these resources. FileWall uses the namespace for defining and applying policies on network file systems.Unlike the above systems, the file names represent more than a mapping to a physical resource (files, network sockets etc.). Along with the access context and policies, names are used as the vehicle to implement file access policies: from on-the-fly modification of access control, to supporting

quality of service.

Semantic File Systems [180], Jade [160], and Prospero [136] use programmable namespaces to organize and extend physical namespaces. FileWall shares the idea of programmability using the control namespace. However, there are several important differences. First, unlike the above where the filters execute on the client, FileWall policy enforcement is offloaded to a network middlebox and does not require modifications to the server. Second, the policies are dynamic and can be modified during execution. Finally, while FileWall can be used to organize data sources, its primary goal is protection by controlling access to portions of the namespace.

In addition to the classical literature on naming in distributed systems [175, 109, 123], there has been some recent research in wide area naming and resolution with a focus on programmability, e.g., *ActiveNames* [208] and Intentional Naming System (INS) [2]. ActiveNames allow applications to define arbitrary computation that executes on names at name resolvers. The goal of ActiveNames is to locate and transport wide area services through a programmable naming abstraction. Intentional Naming System utilizes late binding and declarative style data structures for maintaining attribute-value pairs used to bind a user-specified name to an appropriate instance of the target resource.

FileWall differs from both ActiveNames and INS in its goal as well as its realization. First, in FileWall, the goal is to implement file access policies and not to locate resources. Second, unlike name resolution, where names are mapped to resources, virtual namespaces in FileWall are the result of applying policies using the access context and are a subset of the file system namespace. Finally, FileWall interposes on the client-server path and is not invoked by the client system as the primary target of name resolution.

Role-Based Access Control (RBAC): The role-based access control model has evolved from the use of groups in UNIX and other operating systems, privilege groupings in database management systems, and separation of duty concepts. The modern concept of RBAC embodies the above in a single access control model ([96, 72] and references therein). Gustaffson et. al. [84] demonstrate the use of NFS to implement RBAC for distributed systems. They use modified NFS servers and clients to implement user-role mappings. However, their system does not support user session modification, role hierarchies, or separation of duty constraints.

Law Governed Interaction (LGI) [120] proposes a distributed policy definition and enforcement framework for heterogeneous distributed systems [178]. He et. al. use the LGI framework to implement RBAC for network filesystems [87]. However, their system requires the server to execute an access control agent to hook into the security framework. It also requires specialized user agents at each client to communicate with the access control system. Unfortunately, unlike FileWall, the server modifications and client agents limit the deployability of their system.

4.10 Summary

In this chapter, we presented FileWall, a system architecture that offloads enforcement of network file system access policies to a network middlebox. Using FileWall, we demonstrate that policies that cannot otherwise be implemented without significant modification of clients and servers can easily be realized through offloading. We defined a state maintenance and message transformation model, which determines the capabilities, requirements, and restrictions for a system to offload file system policy enforcement in the network.

We used FileWall to define four primitive classes of policies, for statistics monitoring, file handle security, time based access control, and client transparent failover in network file systems, respectively. Through our experimental evaluation of a prototype system, we demonstrated that these policies combine network and file system contexts, are easy to specify, and do not impose significant overheads.

As a case study, we implemented a Role Based Access Control (RBAC) policy for NFS, which does not require client-server modifications or specialized user agents to participate in the access control framework. We demonstrated the use of our system through an example access control policy that supports role hierarchies, user sessions, and static and dynamic separation of duty constraints.

Chapter 5

Conclusions

In this dissertation, we investigated how to architect highly available, extensible, high performance server systems using functionality offloading as the basic building block. We presented three system architectures based on offloading: *TCPServers*, for improving network protocol performance, *Orion*, for continuous OS and application monitoring, and *FileWall*, for implementing access policies for network file systems. The main conclusion of this research is that offloading provides a powerful mechanism to partition and isolate parts of the functionality of system architectures, which would otherwise require significant software modifications or customized hardware. With the availability of idle CPU resources in multiprocessor servers, programmable peripheral devices, and network middleboxes, offloading is practical, and with appropriate software support can be used to improve performance and introduce new functionality without significant modifications to existing systems and protocols.

At different times in the past, functionality offloading has been regarded as the solution for improving system performance. However, continuously increasing CPU speeds and the relatively slow increases in peripheral speeds have relegated it to the background. Detractors of offloading have often cited limitations of the offloading hardware, difficulty in programming such hardware, and extending or fixing protocol implementations as drawbacks that far outweigh the performance benefits.

Today, the growth in the CPU speeds is much slower and the peripherals, especially networking is catching up with them. With network speeds approaching 10Gbps, and the availability of multiple execution contexts in multiple processors or cores, system designers are again looking at offloading to handle network processing. Unlike in the past, when increases in CPU speeds could overcome the performance bottlenecks, system architectures must adapt to exploit increased parallelism to harness the available computing resources. As a consequence, rather

than data copying overheads, limited parallelism and memory access overheads due to reduced locality of reference are limiting the protocol performance today.

TCPServers is a system architecture that focuses on offloading network processing *within* a multiprocessor system. TCPServers partitions the processors of a multiprocessor system into dedicated packet processing engines, which handle all network processing, and application processors, which execute the applications. In TCPServers, the packet processing engines are not resource constrained as they are identical to the host processors and have access to the entire system memory. Moreover, programming for the packet processing enginges (PPEs) is identical to that for the host OS and development environments are familiar. This enables easy maintenance and easy upgrades of the offloaded functionality.

The goal of offloading in TCPServers is to reduce the loss of memory access locality due to cohabitation overheads, which occur when applications and the network stack share the same processors. By offloading, TCPServers reduces the impact of interrupts, context switches, TLB flushes, and migration of tasks across CPUs, on both application and network stack processing. Our experimental results showed that, while such offloading improves network performance, the gains are small due to the limited concurrency of the OS network stack.

To improve the network stack concurrency, we designed mechanisms that build on the basic TCPServers offloading architecture to perform early demultiplexing. We presented Receive Queues, which are per-socket OS data structures that store incoming packets destined for the connection. Using Receive Queues and a scheduling algorithm for processing packets at connection priorities, we demonstrated significant increase in the concurrency, lower synchronization and scheduler overhead, and graceful degradation of performance at high connection loads.

In computer systems today, monitoring and management functionality are arguably more important than raw performance. With the low cost of hardware and increasing parallelism in server workloads, it has become possible to use a large collection of servers to deliver expected performance to the users. However, providing continuous service still remains a challenge. Due to reliance on a large number external software components, unknown or unpredictable hardware failures, OS deadlocks or crashes, and presence of malicious or inept users, it is difficult to design software that handles all possible failures. Using redundancy to maintain continuous

service relies on online monitoring, diagnosis using live state, and if possible, repair of the incorrect state. Unfortunately, continuous monitoring of software systems imposes significant overheads on the primary system functionality and is not effective.

We presented Orion, a system architecture that offloads monitoring and repair functionality to external hardware. Orion offloads monitoring functionality to a programmable network interface that sits on the system I/O bus, has its own processor, and has direct access to target memory. This network interface can be connected over a private network to cooperatively monitor nodes in a collocated cluster. We use offloading as the basic building block of a framework where monitoring is performed without involving the target CPU. This framework is extensible and defines a programming interface that allows developers and system administrators to build monitoring and repair support for current and future software components. We have implemented Orion and through our case studies demonstrated its use in failure detection and remote repair of OS state damaged due to resource exhaustion. These case studies illustrate the benefits of offloading in improving availability. Continuous monitoring with Orion imposes little overhead during failure-free execution, and Orion successfully performs failure detection and repair of OS state damaged due to resource exhaustion.

Administration is an important aspect of system management. It is increasingly important for administrators to extend network protocols to enforce access policies on critical system resources. Enforcing policies is especially important in network file systems, which store critical data in any enterprise network. Today, incorporating such functionality requires modifications to both clients and servers to extend the protocol implementations. Modifying file system protocols is impractical due to the size and diversity of the client population, and the servers being proprietary and closed source.

We studied the role of offloading administrative functionality through FileWall, a system architecture that offloads policy enforcement in network file systems. FileWall is a network middlebox that interposes on the client-server network path and transforms messages to implement policies. FileWall allows administrators to define access policies using the network and file system contexts and store persistent state at FileWall. We used FileWall to implement several primitive policies to illustrate attribute transformation, flow transformation, and flow coordination, which are different aspects of message transformation. As a case-study, we used

FileWall to implement a Role-Based Access Control policy for the NFSv3 protocol. With FileWall, implementing RBAC for NFS does not require any modification to clients or servers, and due to a Virtual Namespace, eliminates the need for specialized software agents for client participation in the access control protocol. Through our experimental evaluation, we demonstrated that offloading policy enforcement to FileWall does not impose significant overheads compared to the base file system protocol.

5.1 Role of Offloading in Future System Architectures

Functionality offloading has been used since the early years of computer systems to alleviate the overheads of handling CPU intensive functionality. Over the years, offloading high performance graphics, floating point arithmetic, cryptographic operations, etc. have been accepted as integral parts of system architectures and generate little controversy. Since this functionality is easily isolated and is largely stateless, specialized hardware has always outperformed software implementations.

In contrast, network protocol offloading is one of the most contentious topics in systems. Proponents of offloading have always proclaimed that the next upgrade in network speeds would bring the system down to its knees, and the only solution is to use specialized hardware to offload functionality. The opponents have always pointed to the ever increasing CPU speeds and contended that apart from additional complexity and a stop-gap solution, offloading has little benefit [124, 181]. In this debate, the opponents have almost always prevailed and rightly so, since there has never been the situation where offloading is the only solution to alleviate the performance bottlenecks.

Today, we are again at the crossroads. On one hand, the network speeds are increasing both at the end host, with 10Gbps interfaces, and end-to-end, with broadband connectivity. Applications such as internet video, online games, virtual worlds, etc., are saturating the bandwidth as well as the CPUs at the end hosts. On the other hand, the growth in CPU speeds, which, in the past, allowed the OS to handle high network speeds, has slowed considerably. CPU vendors are now offering simultaneous multi-threading (SMT) or hyperthreading and chip multi-processing

(CMP) or multi-core processors, and peripheral devices are commonly equipped with fast processors and large memories. For the first time, the crucial system design challenge has become how to utilize the parallelism available in the hardware.

Systems research has always "reacted" to hardware innovation and workload pressures to reinvent itself. The design considerations change, the assumptions made in the previous generations are revisited, and novel system architectures are proposed to handle the new reality. There are several examples of systems that were proposed in the early years of computing being revived in a modern context. Virtual machines, which were first proposed for IBM S370 [82] systems in early 1970s, are the most active areas of system research and commercial adoption today. Isolation of peripheral handling and dedicated I/O processors, which were a part of early mainframe designs, are being revisited with intelligent devices [30, 75]. Parallel programming runtimes [94, 29] and programming models have made a comeback in the context of multi and many core processors with transactional memory [89] and Recognition, Synthesis, and Mining initiatives [63]. Thin clients have been revived with the focus on wide area networks and mobile desktops [108].

We believe that the trends above are a reflection of workload characteristics that were earlier limited to *extreme* computing environments becoming commonplace. That is, each of the original systems were designed for applications and environments that required computing resources at the limit or even beyond those available. Today, the same requirements are manifest in applications for desktop systems and even mobile phones. Therefore, system designers must continue to design systems for functionality that current hardware is unable to support effectively.

In this context, an interesting question is to revisit the OS structure itself. Microkernels [1, 112, 183, 186] partitioned the OS functionality into multiple independent servers. This enabled modularity, easy extension, and workload specific adaptation. Similarly, Exokernels [70] proposed to eliminate all OS abstractions and allow applications to build customized libraries to perform OS functionality. The primary drawbacks of these approaches was the *sharing* of scarce CPU resources and the overheads imposed to enable such sharing. With the availability of idle CPU cycles, we must revisit the question of whether the benefits of a monolithic kernel are diminished when compared to the microkernel architecture. In the same

context, offloading can be viewed as a mechanism to introduce new functionality on general purpose hardware through *software* mechanisms. However, we must better understand the interaction of the offloaded functionality with the OS resource management mechanisms, and the interfaces that enable strong isolation of offloaded functionality while retaining the performance benefits of parallelization.

With the growing use of computer systems, the ability to monitor and effectively manage the computing infrastructure is a problem faced not only by highly trained administrators, but also for naive home users. Providing continuous service, identifying anomalies, and the ability to define and enforce high level policies are increasingly becoming the primary evaluation criteria for computer systems. Unfortunately, existing architectures, runtimes, and programming languages provide limited support to incorporate these principles in practice. An important direction for investigation is new languages and runtimes that make it possible to offload application specific monitoring to external hardware or software entities, to ease the programming burden. These runtimes can be viewed as a safety net for programmers who can use and even extend the monitoring functionality *external* to the primary application code. Recently, such runtimes have been investigated using virtual machines [66, 214] and OS support [201, 158] and are an important first step towards the ultimate goal of writing highly reliable and available software.

An important change we envision in the computing landscape is the growing importance of the handheld computers. CPU intensive applications, which also require significant memory and energy resources, are being introduced in cellular phones today. The use of offloading in this context has been limited to migrating all processing to more powerful servers [20], even when idle resources are available locally [113]. We must better understand the application workloads, available hardware resources, and system requirements for performance and reliability to define offloading architectures tailored for such environments.

Going forward, offloading must be recognized as a first order design principle for system architectures. However, it must not be viewed in context of a narrow definition of moving functionality closer to peripherals. As demonstrated in part by this dissertation, offloading is a way of architecting systems to utilize hardware resources, which are otherwise idle.

References

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference* (Atlanta, Georgia, 1986), pp. 93–112.

[2] ADJIE-WINOTO, W., SCHWARTZ, E., BALAKRISHNAN, H., AND LILLEY, J. The design and implementation of an intentional naming system. In *Proc. of SOSP* (1999).

[3] ADYA, A., BOLOSKY, W., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI* (Boston, MA, December 2002).

[4] Akamai. http://www.akamai.com.

[5] Alacritech Storage and Network Acceleration. http://www.alacritech.com.

[6] AMAZON, I. http://www.amazon.com.

[7] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)* (2001), pp. 131–145.

[8] ANDERSON, A. XACML Profile for Role Based Access Control (RBAC). *OASIS Access Control TC Committee Draft 1* (2004), 13.

[9] ANDERSON, D., AND CHASE, J. Fstress: A flexible network file service benchmark, 2001.

[10] ANDERSON, D. C., CHASE, J. S., GADDE, S., GALLATIN, A. J., YOCUM, K. G., AND FEELEY, M. J. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proc. of the 1998 USENIX Technical Conference* (June 1998), pp. 143–154.

[11] ANDERSON, D. C., CHASE, J. S., AND VAHDAT, A. Interposed request routing for scalable network storage. *ACM Trans. Comput. Syst. 20*, 1 (2002), 25–48.

[12] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *Proc. of SOSP* (New York, NY, USA, 1995).

[13] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling File Servers via Cooperative Caching. In *Proc. of 2nd Usenix Symposium on Network Systems Design and Implementation NSDI'05* (Boston, MA, May 2005).

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

[15] APPLE, I. *Porting Drivers to MacOS X*. Apple, Inc., October 2006.

[16] ARANYA, A., WRIGHT, C. P., AND ZADOK, E. Tracefs: A File System to Trace Them All. In *Proc. of FAST* (San Francisco, CA, March/April 2004), USENIX Association.

[17] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BURNETT, N. C., DENEHY, T. E., ENGLE, T. J., GUNAWI, H. S., NUGENT, J., AND POPOVICI, F. I. Transforming Policies into Mechanisms with Infokernel. In *Proc. of SOSP* (October 2003).

[18] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. Models and issues in data stream systems. In *Proceedings of the ACM Symposium on Principles of Database Systems* (June 2002), pp. 1–16.

[19] BALDWIN, J. H. Locking in the multithreaded freebsd kernel. In *Proceedings of BSD-Con 2002* (2002), pp. 27–35.

[20] BARATTO, R. A., POTTER, S., SU, G., AND NIEH, J. Mobidesk: mobile virtual desktop computing. In *Proceedings of the Annual International Conference on Mobile Computing and Networking* (2004), pp. 1–15.

[21] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. of SOSP* (oct 2003).

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

[23] BASU, A., BUCH, V., VOGELS, W., AND VON EICKEN, T. 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).

[24] BAVIER, A. C., BOWMAN, M., CHUN, B. N., CULLER, D. E., KARLIN, S., MUIR, S., PETERSON, L. L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating systems support for planetary-scale network services. In *1st Symposium on Networked Systems Design and Implementation (NSDI)* (2004), pp. 253–266.

[25] Broadcom Corporation: BCM5706: 10/100/1000BASE-T TCP Offload Engine, RDMA, ISCSI/ISER and Ethernet Controller. http://www.broadcom.com/products/product.php?product_id=BCM5706&category_id=39.

[26] BIANCHINI, R., AND RAJAMONY, R. Power and energy management for server systems. *IEEE Computer 37*, 11 (2004), 68–74.

[27] BIRRELL, A., HISGEN, A., JERIAN, C., MANN, T., AND SWART, G. The echo distributed file system. Tech. Rep. 111, DEC SRC, Palo Alto, CA, 10 1993.

[28] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[29] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1995), pp. 207–216.

[30] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. Myrinet: A gigabit-per-second local area network. *IEEE Micro 15*, 1 (Feb. 1995), 29–36.

[31] BOHRA, A., BALIGA, A., AND IFTODE, L. Orion: Looking for constellations in physical memory. Tech. Rep. DCS-TR-569, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854, 2005.

[32] BOHRA, A., NEAMTIU, I., GALLARD, P., SULTAN, F., AND IFTODE, L. Remote Repair of OS State Using Backdoors. In *Proc. ICAC* (May 2004).

[33] BOHRA, A., SMALDONE, S., AND IFTODE, L. Frac: Implementing role-based access control for network file systems. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA)* (2007), pp. 95–104.

[34] BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems (TOCS) 7*, 1 (1989), 1–24.

[35] BOS, H., DE BRUIJN, W., CRISTEA, M.-L., NGUYEN, T., AND PORTOKALIDIS, G. Ffpf: Fairly fast packet filters. In *Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 347–363.

[36] BRECHT, T., JANAKIRAMAN, G. J., LYNN, B., SALETORE, V., AND TURNER, Y. Evaluating network processing efficiency with processor partitioning and asynchronous i/o. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), ACM, pp. 265–278.

[37] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based Fault Tolerance. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1995).

[38] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proc. USENIX Annual Technical Conference* (June 2003).

[39] BYERS, J., CONSIDINE, J., MITZENMACHER, M., AND ROST, S. Informed content delivery across adaptive overlay networks. In *Proceedings of ACM SIGCOMM 2002* (Pittsburgh, PA, August 2002).

[40] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot - a technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 31–44.

[41] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot - a technique for cheap recovery. In *Proceedings of Symposium on Operating Systems Design and Implementation(OSDI'04)* (2004), pp. 31–44.

[42] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the Usenix Annual Technical Conference, General track* (2004), pp. 15–28.

[43] CAPRITA, B., CHAN, W. C., NIEH, J., STEIN, C., AND ZHENG, H. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX* (2005), pp. 337–352.

[44] CASTRO, M., ET AL. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003), pp. 298–313.

[45] CHANDRA, S., AND CHEN, P. M. Whither generic recovery from application faults? a fault study using open-source software. In *000 International Conference on Dependable Systems and Networks (DSN, formerly FTCS-30 and DCCA-8)* (2000), pp. 97–106.

[46] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A., AND DOYLE, R. P. Managing energy and server resources in hosting centres. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 103–116.

[47] CHEN, M., KICIMAN, E., BREWER, E., AND FOX, A. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of International Conference on Dependable Systems and Networks (DSN 2002)* (June 2002).

[48] CHEN, M. Y., ACCARDI, A., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. Path-based failure and evolution management. In *Proceedings of Symposium on Networked Systems Design and Implementation(NSDI'04)* (2004), pp. 309–322.

[49] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio File Cache: Surviving Operating System Crashes. In *Proc. ASPLOS'96* (1996).

[50] CLARK, D. D., JACOBSON, V., ROMKEY, J., AND SALWEN, H. An analysis of TCP processing overhead. *IEEE Communications Magazine 27*, 6 (1989), 23–29.

[51] COHEN, B. Incentives Build Robustness in BitTorrent. http://bittorrent.com/bittorrentecon.pdf, May 2003.

[52] COHEN, I., CHASE, J. S., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of Symposium on Operating Systems Design and Implementation* (2004), pp. 231–244.

[53] COHEN, W. H. Gaining insight into the linux kernel with kprobes. http://www.redhat.com/magazine/005mar05/features/kprobes/, May 2005.

[54] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[55] CORP., I. Ipmi: Intelligent platform management interface specification second generation. http://www.intel.com/design/servers/ipmi/spec.htm, Feb 2004.

[56] COVERITY, I. http://www.coverity.com.

[57] CRANOR, C., GAO, Y., JOHNSON, T., SHKAPENYUK, V., AND SPATSCHECK, O. Gigascope: high performance network monitoring with an SQL interface. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (2002), 623–623.

[58] DABEK, F., ET AL. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Chateau Lake Louise, Banff, Canada, October 2001).

[59] DAT COLLABORATIVE. Direct access transport. http://www.datcollaborative.org/.

[60] DEMSKY, B., AND RINARD, M. C. Automatic detection and repair of errors in data structures. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA* (2003), pp. 78–95.

[61] DEVELOPERS, V. *Valgrind User Manual*, 3.2.0 ed. Valgrind, June 2006. http://valgrind.org/docs/manual/manual.html.

[62] DRUSCHEL, P., AND BANGA, G. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation* (1996), pp. 261–275.

[63] DUBEY, P. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Intel Technology Journal* (2005), 1.

[64] DUBNICKI, C., ET AL. Software Support for Virtual Memory-Mapped Communication. In *Proc. of the 10th International Parallel Processing Symposium* (1996), pp. 372–281.

[65] DUNAGAN, J., HARVEY, N. J. A., JONES, M. B., KOSTIC, D., THEIMER, M., AND WOLMAN, A. Fuse: Lightweight guaranteed distributed failure notification. In *6th Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 151–166.

[66] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI* (2002).

[67] DUNNING, D., ET AL. The Virtual Interface Architecture. *IEEE Micro* (1998).

[68] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. I. Passive NFS Tracing of Email and Research Workloads. In *Proc. FAST* (San Francisco, CA, 2003).

[69] ELLARD, D., AND SELTZER, M. I. Nfs tricks and benchmarking traps. In *USENIX Annual Technical Conference, FREENIX Track* (2003), pp. 101–114.

[70] ENGLER, D., KAASHOEK, M., AND JR., J. O. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995).

[71] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst. 21*, 3 (2003), 270–313.

[72] FERRAIOLO, D., ET AL. Proposed NIST standard for role-based access control. *ACM Trans. on Information and System Security (TISSEC) 4*, 3 (2001), 224–274.

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

[74] Firewall. http://en.wikipedia.org/wiki/Firewall_%28networking%29.

[75] FIUCZYNSKI, M., MARTIN, R., OWA, T., AND BERSHAD, B. On using intelligent network interface cards to support multimedia applications. In *Proceedings of the Seventh Workshop on Network and Operating System Support for Digital Audio and Video* (July 1998).

[76] FREIMUTH, D., HU, E. C., LAVOIE, J. D., MRAZ, R., NAHUM, E. M., PRADHAN, P., AND TRACEY, J. M. Server network scalability and tcp offload. In *Proceedings of the Usenix Annual Technical Conference, General track* (2005), pp. 209–222.

[77] FU, K., KAASHOEK, M. F., AND MAZIERES, D. Fast and Secure Distributed Read-Only File System. *ACM Trans. Comput. Syst. 20*, 1 (2002), 1–24.

[78] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proc. of SOSP* (2003).

[79] GIBSON, G. A., ET AL. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct 1998), pp. 92–103.

[80] GLEESON, B., LIN, A., HEINANEN, J., ARMITAGE, G., AND MALIS, A. A Framework for IP Based Virtual Private Networks.

[81] Google. http://www.google.com/.

[82] GUM, P. H. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development 27*, 6 (1983), 530–544.

[83] GUPTA, I., CHANDRA, T., AND GOLDSZMIDT, G. On Scalable and Efficient Distributed Failure Detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing* (Apr. 2001).

[84] GUSTAFSSON, M., DELIGNY, B., AND SHAHMEHRI, N. Using nfs to implement role-based access control. In *Proceedings of WET-ICE '97* (Washington, DC, USA, 1997).

[85] HAIN, T. Architectural implications of NAT. RFC 2993, Internet Engineering Task Force, Nov. 2000.

[86] HARTMEIER, D. Design and performance of the openbsd stateful packet filter (pf). In *FREENIX* (2002), pp. 171–180.

[87] HE, Z., PHAN, T., AND NGUYEN., T. D. Enforcing enterprise-wide policies over standard client-server interactions. In *Proc. of SRDS* (oct 2005).

[88] HEATH, T., CENTENO, A. P., GEORGE, P., RAMOS, L., AND JALURIA, Y. Mercury and freon: temperature emulation and management for server systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS* (2006), pp. 106–116.

[89] HERLIHY, M., ELIOT, J., AND MOSS, B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the ACM SIGARCH International Symposium on Computer Architecture* (1993), pp. 289–300.

[90] HEWLETT-PACKARD CORPORATION, INTEL CORPORATION, MICROSOFT CORPORATION, PHOENIX TECHNOLOGIES LTD., AND TOSHIBA CORPORATION. *Advanced Configuration and Power Interface Specification*, 3.0b ed., October 2006.

[91] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 51–81.

[92] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 51–81.

[93] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng. 17*, 1 (1991), 64–76.

[94] IFTODE, L., SINGH, J. P., AND LI, K. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture* (May 1996).

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

[96] INTERNATIONAL COMMITTEE FOR INFORMATION TECHNOLOGY. Role-based access control. ANSI/INCITS 359-2004, Feb. 2004.

[97] KATCHER, J. Postmark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance Inc., October 1997.

[98] KEPHART, J. O., AND CHESS, D. M. The vision of autonomic computing. *IEEE Computer 36*, 1 (2003), 41–50.

[99] KIM, H., AND RIXNER, S. Tcp offload through connection handoff. In *Proceedings of Eurosys 2006* (Leuven, Belgium, April 2006).

[100] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *ACM Trans. Comput. Syst. 23*, 1 (2005), 51–76.

[101] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst. 10*, 1 (1992).

[102] KOCH, R. R., SHORTIKAR, S., MOSER, L. E., AND MELLIAR-SMITH, P. M. Transparent TCP Connection Failover. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)* (2003).

[103] KOHLER, E., ET AL. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (August 2000).

[104] KOKKU, R., BOHRA, A., GANGULY, S., AND VENKATARAMANI, A. A multipath background network architecture. In *INFOCOM* (2007), IEEE, pp. 1352–1360.

[105] KOSTIC, D., ET AL. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003), pp. 282–297.

[106] KROHN, M., FREEDMAN, M., AND MAZIERES, D. On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution. In *Proceedings of IEEE Symposium on Security and Privacy* (Oakland, CA, May 2004).

[107] KUBIATOWICZ, J., ET AL. OceanStore: an Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), pp. 190–201.

[108] LAI, A. M., AND NIEH, J. On the performance of wide-area thin-client computing. *ACM Trans. Comput. Syst. 24*, 2 (2006), 175–209.

[109] LAMPSON, B. W. Designing a global name service. In *Proc. of PODC* (1986).

[110] LANFRANCHI, G., DELLA PERUTA, P., PERRONE, A., AND CALVANESE, D. Toward a new landscape of systems management in an autonomic computing environment. *IBM SYSTEMS JOURNAL 42*, 1 (2003), 120.

[111] LIAO, C., JIANG, D., IFTODE, L., MARTONOSI, M., AND CLARK, D. W. Monitoring shared virtual memory performance on a myrinet-based pc cluster. In *International Conference on Supercomputing* (1998), pp. 251–258.

[112] LIEDTKE, J. On micro-kernel construction. In *Proceedings of ACM Symposium on Operating Systems PrinciplesSOSP(95)* (1995), pp. 237–250.

[113] LIMITED, A. The arm cortex-a9 processors. http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf, September 2007.

[114] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., AND SHRIRA, L. Replication in the harp file system. In *Proc. of SOSP* (New York, NY, USA, 1991).

[115] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In *4th Symposium on Operating System Design and Implementation (OSDI)* (2000), pp. 289–304.

[116] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS XII)* (2006), pp. 37–48.

[117] MANOCHA, D. General-Purpose Computations Using Graphics Processors. *Computer 38*, 8 (2005), 85–88.

[118] MAQUELIN, O., GAO, G. R., HUM, H. H. J., THEOBALD, K., AND TIAN, X. Polling watchdog : Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual Intl. Symposium on Computer Architecture* (New York, May 22–24 1996), ACM Press, pp. 179–190.

[119] MAZIERES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proc. of SOSP* (dec 1999).

[120] MINSKY, N. H., AND UNGUREANU, V. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol. 9*, 3 (2000).

[121] MIRETSKIY, Y., DAS, A., WRIGHT, C. P., AND ZADOK, E. Avfs: An On-Access Anti-Virus File System. In *Proc. of Security 2004* (San Diego, CA, August 2004), USENIX Association.

[122] MISHRA, S., MARWAH, M., AND FETZER, C. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)* (2003).

[123] MOCKAPETRIS, P., AND DUNLAP, K. J. Development of the domain name system. In *Proc. of SIGCOMM* (1988).

[124] MOGUL, J. C. Tcp offload is a dumb idea whose time has come. In *HotOS* (2003), pp. 25–30.

[125] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proc. of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA* (Jan. 1996), pp. 99–111.

[126] MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., PROEBSTING, T. A., AND HARTMAN, J. H. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation* (1994), p. 200.

[127] MOSBERGER, D., AND JIN, T. httperf – A Tool for Measuring Web Server Performance, 1998.

[128] MUIR, S., AND SMITH, J. AsyMOS - An Asymetric Multiprocessor Operating System. In *Proceedings of Open Architectures and Network Programming* (San Francisco, CA, April 1998).

[129] MUNISWAMY-REDDY, K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A Versatile and User-Oriented Versioning File System. In *Proc. of FAST* (San Francisco, CA, March/April 2004).

[130] MUTHITACHAROEN, A., CHEN, B., AND MAZIèRES, D. A low-bandwidth network file system. In *Proc. of SOSP* (New York, NY, USA, 2001).

[131] MUTHITACHAROEN, A., ET AL. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)* (Boston, Massachusetts, December 2002).

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

[133] NAGARAJA, K. *A Systematic Approach to Quantifying and Improving the Availability of Internet Services*. PhD thesis, Rutgers University, 2005.

[134] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and dealing with operator mistakes in internet services. In *6th Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 61–76.

[135] NAHUM, E. M., YATES, D. J., KUROSE, J. F., AND TOWSLEY, D. Performance isses in parallelized network protocols. In *Proceedings of Symposium on Operating Systems Design and Implementation(OSDI'94)* (Nov 1994).

[136] NEUMAN, B. C. The prospero file system: A global file system based on the virtual system model. *Computing Systems 5*, 4 (1992), 407–432.

[137] NIELSEN, M. J. K. Titan System Manual. Tech. Rep. WRL-86-1, HP Labs, Sept. 1986.

[138] OLIVEIRA, F., NAGARAJA, K., BACHWANI, R., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and validating database system administration. In *Proceedings of the Usenix Annual Technical Conference, General track(Usenix'06)* (2006), pp. 213–228.

[139] OLSHEFSKI, D., AND NIEH, J. Understanding the management of client perceived response time. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems* (2006), pp. 240–251.

[140] OLSHEFSKI, D. P., NIEH, J., AND NAHUM, E. M. ksniffer: Determining the remote client perceived response time from live packet streams. In *Proceedings of Symposium on Operating Systems Design and Implementation(OSDI'04)* (2004), pp. 333–346.

[141] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why Do Internet Services Fail, and What Can Be Done About It? In *Proc. 4th USENIX Symp. on Internet Technologies and Systems (USITS)* (Mar. 2003).

[142] O'TOOLE, J. W., AND GIFFORD, D. K. Names should mean what, not where. In *Proc. of ACM SIGOPS-EW* (1992).

[143] OUSTERHOUT, J., CHERENSON, A., DOUGLIS, F., NELSON, M., AND WELCH, B. The sprite network operating system. *IEEE Computer 21*, 2 (1988).

[144] PAI, V., ET AL. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[145] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems 18*, 1 (2000), 37–66.

[146] PATTERSON, D., ET AL. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[147] PETRONI JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 179–194.

[148] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from bell labs. *Computing Systems 8*, 3 (Summer 1995), 221–254.

[149] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The use of name spaces in Plan 9. *Operating Systems Review 27*, 2 (1993), 72–76.

[150] PINHEIRO, E., BIANCHINI, R., AND DUBNICKI, C. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance* (2006), pp. 15–26.

[151] POSTEL, J. RFC 793: Transmission Control Protocol, Sept. 1981.

[152] PRATT, I. A., AND FRASER, K. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (2001), pp. 67–76.

[153] PROJECT, F. *ngether – Ethernet netgraph node type*. FreeBSD Man Pages.

[154] PROJECT, F. *sendfile – send a file to a socket*. FreeBSD System Calls Manual.

[155] PROJECT, N. Netfilter: Firewalling, nat, and packet filtering for linux. http://www.netfilter.org/.

[156] PROVOS, N. libevent - an event notification library. http://monkey.org/ provos/libevent/.

[157] QIE, X., PANG, R., AND PETERSON, L. Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software. In *Proceedings of Symposium on Operating Systems Design and Implementation(OSDI '02)* (December 2002).

[158] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: treating bugs as allergies - a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (2005), pp. 235–248.

[159] RANGARAJAN, M., ET AL. TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance. Tech. Rep. 481, Rutgers University, Mar 2002.

[160] RAO, H. C., AND PETERSON, L. L. Accessing files in an internet: The jade file system. *IEEE Trans on Software Eng. 19*, 6 (1993).

[161] REGNIER, G. J., MAKINENI, S., ILLIKKAL, R., IYER, R. R., MINTURN, D. B., HUGGAHALLI, R., NEWELL, D., CLINE, L. S., AND FOONG, A. Tcp onloading for data center servers. *IEEE Computer 37*, 11 (2004), 48–58.

[162] REGNIER, G. J., MINTURN, D. B., MCALPINE, G. L., SALETORE, V. A., AND FOONG, A. Eta: Experience with an intel xeon processor as a packet processing engine. *IEEE Micro 24*, 1 (2004), 24–31.

[163] REYNOLDS, P., WIENER, J. L., MOGUL, J. C., AGUILERA, M. K., AND VAHDAT, A. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web (WWW)* (2006), pp. 347–356.

[164] RHEA, S. C., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. Opendht: a public dht service and its uses. In *Proceedings of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2005), pp. 73–84.

[165] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K., AND HOPPER, A. Virtual network computing. *Internet Computing, IEEE 2*, 1 (1998), 33–38.

[166] RINARD, M. C., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND BEEBEE, W. S. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of Symposium on Operating Systems Design and Implementation(OSDI'04)* (2004), pp. 303–316.

[167] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001), pp. 188–201.

[168] SALIM, J. H., OLSSON, R., AND KUZNETSOV, A. Beyond softnet. In *Proc. 5th Annual Linux Showcase and Conference* (Oakland, CA, Nov 2001), pp. 165–172.

[169] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (1984), 277–288.

[170] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *IEEE, Proceedings 63* (1975), 1278–1308.

[171] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the Migration of Virtual Computers. In *Proc. of OSDI'02* (Dec. 2002).

[172] SCHEIFLER, R., AND GETTYS, J. The X window system. *ACM Transactions on Graphics (TOG) 5*, 2 (1986), 79–109.

[173] SCHMIDT, D. C., AND SUDA, T. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (1995), pp. 624–633.

[174] SCHMUCK, F. B., AND HASKIN, R. L. Gpfs: A shared-disk file system for large computing clusters. In *Proc. of FAST* (Berkeley, CA, USA, 2002).

[175] SCHROEDER, M. D., BIRRELL, A. D., AND NEEDHAM, R. M. Experience with grapevine (summary): the growth of a distributed system. In *Proc. of SOSP* (1983).

[176] SELTZER, M., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation (OSDI)* (Oct. 1996).

[177] SELTZER, M., AND SMALL, C. Self-Monitoring and Self-Adapting Operating Systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems* (May 1997).

[178] SERBAN, C., AND MINSKY, N. H. Generalized access control of synchronous communication. In *Proceedings of ACM/IFIP/USENIX 7th International Middleware Conference, Middleware (2006)* (2006), pp. 281–300.

[179] SHEERS, K. HP OpenView Event Correlation Services. *Hewlett-Packard Journal 47*, 5 (1996), 31–42.

[180] SHELDON, M., GIFFORD, D., JOUVELOT, P., AND JR., J. O. Semantic file systems. In *Proc. of SOSP* (Pacific Grove, CA, 1991).

[181] SHIVAM, P., AND CHASE, J. S. On the elusive benefits of protocol offload. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence* (New York, NY, USA, 2003), ACM Press, pp. 179–184.

[182] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. The EarlyBird System for Real-time Detection of Unknown Worms. *ACM Workshop on Hot Topics in Networks* (2003).

[183] SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles* (Dec. 1995).

[184] SLEEPYCAT SOFTWARE INC. Berkeley DB. http://dev.sleepycat.com/.

[185] SMALDONE, S., BOHRA, A., AND IFTODE, L. Filewall: A firewall for network file systems. In *Proceedings of 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'07)* (Columbia, MD, Sep 2007).

[186] SMALL, C., AND SELTZER, M. A Comparison of OS Extension Technologies, . In *Proceedings of the 1996 USENIX Conference*.

[187] SMITH, J. M., AND TRAW, C. B. S. Giving Applications Access to Gb/s Networking. *IEEE Network 7*, 4 (July 1993), 44–52.

[188] SNOEREN, A. C., AND BALAKRISHNAN, H. An End-to-End Approach to Host Mobility. In *Proc. 6th ACM MOBICOM* (Aug. 2000).

[189] SOULES, C., ET AL. System support for online reconfiguration. In *In Proc. USENIX Annual Technical Conference* (June 2003).

[190] SPRING, N. T., MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. E. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw. 12*, 1 (2004), 2–16.

[191] STEINER, J. G., NEUMAN, B. C., AND SCHILLER, J. I. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX winter conference* (1988), pp. 191–202.

[192] STEVE MUIR AND JONATHAN SMITH. Functional divisions in the Piglet multiprocessor operating system. In *Eighth ACM SIGOPS European Workshop* (September 1998).

[193] STEVENS, C. E. *Information Technology - AT Attachment 8 - ATA/ATAPI Commad Set (ATA8-ACS)*. American National Standard Institute (ANSI), December 2006.

[194] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking 11* (2003).

[195] STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. Self-securing storage: Protecting data in compromised systems. In *Proc. of OSDI* (October 2000).

[196] SULLIVAN, M., AND CHILLAREGE, R. Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems. In *Proc. 21st Int'l. Symp. on Fault-Tolerant Computing (FTCS-21)* (1991).

[197] SULTAN, F., BOHRA, A., AND IFTODE, L. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proc. SRDS* (Oct. 2003).

[198] SULTAN, F., BOHRA, A., NEAMTIU, I., AND IFTODE, L. Nonintrusive Remote Healing Using Backdoors. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems, FCRC* (San Diego, CA, June 2003).

[199] SULTAN, F., BOHRA, A., SMALDONE, S., PAN, Y., GALLARD, P., NEAMTIU, I., AND IFTODE, L. Recovering internet service sessions from operating system failures. *IEEE Internet Computing 9*, 2 (2005), 17–27.

[200] SULTAN, F., SRINIVASAN, K., AND IFTODE, L. Transport Layer Support for Highly-Available Network Services. In *Proc. HotOS-VIII* (May 2001). Extended version: Technical Report DCS-TR-429, Rutgers University.

[201] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Proc. 19th Symp. on Operating Systems Principles (SOSP)* (Oct. 2003).

[202] SYMANTEC, I. http://www.symantec.com.

[203] TENNENHOUSE, D., SMITH, J., SINCOSKIE, W., WETHERALL, D., AND MINDEN, G. A survey of active network research. *Communications Magazine, IEEE 35*, 1 (1997), 80–86.

[204] TEWARI, R., HASWELL, J. M., NAIK, M. P., AND PARKES, S. M. Glamour: A wide-area file system middleware using nfsv4. Tech. Rep. RJ10368(A0507-011), IBM Research Division, Almaden Research Center, San Jose, CA, July 2005.

[205] THEKKATH, C., MANN, T., AND LEE, E. Frangipani: A scalable distributed file system. In *Proc. of SOSP* (Saint-Malo, France, 1997).

[206] TRAEGER, A., RAI, A., WRIGHT, C. P., AND ZADOK, E. NFS File Handle Security. Tech. Rep. FSL-04-03, Computer Science Department, Stony Brook University, May 2004.

[207] TRIPWIRE, I. Tripwire configuration audit and control. http://www.tripwire.com/.

[208] VAHDAT, A., DAHLIN, M., ANDERSON, T., AND AGGARWAL, A. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proc. of USITS* (1999).

[209] WALDSPURGER, C. A. Memory resource management in vmware esx server. In *5th Symposium on Operating System Design and Implementation (OSDI)* (2002).

[210] WALFISH, M., STRIBLING, J., KROHN, M. N., BALAKRISHNAN, H., MORRIS, R., AND SHENKER, S. Middleboxes no longer considered harmful. In *Proceedings of 6th Usenix Conference on Operating System Design and Implementation (OSDI)* (2004), pp. 215–230.

[211] WANG, L., PAI, V., AND PETERSON, L. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of 5th Usenix Conference on Operating System Design and Implementation (OSDI)* (Boston, MA, December 2002).

[212] WEATHERLEY, R. An aspect oriented approach to writing compilers. http://www.southern-storm.com.au/treecc.html.

[213] WEAVER, N., STANIFORD, S., AND PAXSON, V. Very fast containment of scanning worms. In *USENIX Security Symposium* (2004), pp. 29–44.

[214] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating System Design and Implementation (OSDI)* (2002).

[215] WILLMANN, P., RIXNER, S., AND COX, A. L. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the Usenix Annual Technical Conference, General track* (Boston, MA, June 2006).

[216] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. Versatility and unix semantics in namespace unification. *ACM TOS 2*, 1 (March 2006). To appear.

[217] XU, M., BODÍK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th International Symposium on Computer Architecture (ISCA)* (2003), pp. 122–133.

[218] YATES, D. J., NAHUM, E. M., KUROSE, J. F., AND TOWSLEY, D. Networking support for large scale multiprocessor servers. In *Proceedings of ACM SIGMETRICS 1996* (May 1996), pp. 116–125.

[219] YOUB KIM, H., AND RIXNER, S. Connection handoff policies for tcp offload network interfaces. In *7th Symposium on Operating System Design and Implementation (OSDI)* (2006), pp. 293–306.

[220] ZADOK, E., AND NIEH, J. FiST: A Language for Stackable File Systems. In *Proc. of USENIX* (San Diego, CA, June 2000), USENIX Association.

[221] ZHANG, M., LAI, J., KRISHNAMURTHY, A., PETERSON, L. L., AND WANG, R. Y. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *USENIX Annual Technical Conference, General Track* (2004), pp. 99–112.

[222] ZHANG, M., ZHANG, C., PAI, V. S., PETERSON, L. L., AND WANG, R. Y. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *OSDI* (2004), pp. 167–182.

[223] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS XI)* (2004), pp. 177–188.

[224] ZHU, N., ELLARD, D., AND CHIEUH, T.-C. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proc. of FAST* (San Francisco, CA, December 2005).

[225] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (2005), pp. 177–190.

[226] ZHUANG, X., ZHANG, T., AND PANDE, S. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS XI)* (2004), pp. 72–84.

Vita

Aniruddha Bohra

Education

- Ph.D. Computer Science, Rutgers University, New Jersey (2008)
- M.S. Computer Science, Rutgers University, New Jersey (2002)
- B.E. Computer Engineering, Netaji Subhas Inst. of Tech., New Delhi, India (1999)

Occupations and Positions Held

- Research Staff Member, NEC Laboratories America, Princeton, New Jersey (2005-)

Publications

- A. Bohra, S. Smaldone, and L. Iftode "FileWall: A Firewall for Network File Systems" *Proceedings of IEEE Dependable Autonomic and Secure Computing, DASC 2007*
- A. Bohra and L. Iftode "Improving Network Stack Concurrency using TCPServers" *Proceedings of IEEE Network Computing and Applications, NCA 2007*
- A. Bohra, S. Smaldone, and L. Iftode "FRAC: Implementing Role-Based Access Control for Network File Systems" *Proceedings of IEEE Network Computing and Applications, NCA 2007*
- R. Kokku, A. Bohra, S. Ganguly, and V. Arun "A Multipath Background Network Architecture" *Proceedings of IEEE Infocom, 2007*
- V. Navda, A. Bohra, S. Ganguly, and D. Rubenstein "Using Channel Hopping to Increase 802.11 Resilience to Jamming Attacks" *Proceedings of IEEE Infocom Minisymposium, 2007*
- J. Liang, A. Bohra, H. Zhang, S. Ganguly, and R. Izmailov "Minimizing Metadata Access Latency in Wide Area File Systems" *Proceedings of IEEE High Performance Computing HiPC'06, 2006*
- F. Sultan, A. Bohra, P. Gallard, I. Neamtiu, S. Smaldone, Y. Pan. Neamtiu, and L. Iftode. "Recovering Internet Service Sessions from Operating System Failures." *in IEEE Internet Computing,ICSI-0116-0804 Special Issue - Recovery-Oriented Approaches to Dependability*.

- A. Bohra, I. Neamtiu, P. Gallard , F. Sultan , and L. Iftode. "Remote Repair of Operating System State Using Backdoors." *Proceedings of First IEEE International Conference on Autonomic Computing (ICAC'04). 2004*.

- F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode. "Nonintrusive Remote Healing Using Backdoors." *Proceedings of First Workshop on Algorithms and Architectures for Self-Managing Systems(Self Manage'03), June 2003.*

- F. Sultan, A. Bohra, and L. Iftode. "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions". *Proceedings of IEEE Symposium on Reliable Distributed Systems, SRDS 03.*

- A. Bohra and E. Gabber. "Are Mallocs Free of Fragmentation?" *Proceedings of Usenix Annual Technical Conference, Freenix Track, 2001.*