Operating System Support for Active Networks

Computer and Information Sciences Technical Report MS-CIS-97-03

J. S. Shapiro, S. J. Muir, J. M. Smith and D. J. Farber[1]

Distributed Systems Laboratory

University of Pennsylvania, Philadelphia, PA 19104-6389

{shap,sjmuir,jms,farber}@dsl.cis.upenn.edu

February 8, 1997

Abstract

Active Networks provide a network infrastructure which can be programmed on a per-user or even a per-packet basis. While the flexibility of such systems is seductive, the safety and security risks are greatly increased. Active network nodes must provide programmability while respecting policy constraints of the network provider and protecting nodes and other users from faulty applications. In addition, many modern applications demand Quality of Service guarantees which will require exposing selected portions of the node resource management policy. As shared infrastructure, reliability beyond that of general purpose computing systems is required.

EROS (the Extremely Reliable Operating System) is an attractive candidate for resource management in an Active Network node. EROS is a pure capability system which provides complete accountability for persistent, consumable and multiplexed resources. We have implemented a prototype Active Network element on the EROS platform which provides the resource partitioning, quality of service, inter-client protection, and performance required of active networks. This paper presents the architecture, design and performance of this prototype, which is currently running on Intel x86 platforms.

1 Introduction

1.1 What are Active Networks?

An *Active Network*[28] consists of network elements which are programmable, on a per-node, per-flow, or per-packet basis. They are thus a further step in the merger of computing and communications, where the network infrastructure is *virtual*, and instantiated with the aid of computing devices. The IP Internet provides a great example of a virtual infrastructure, as it provides the illusion of a large addressable network via software running on elements interconnected with heterogeneous subnets.

This degree of programmability represents a major transition in thinking about networks. Network elements in a programmable network infrastructure must do more than store-and-forward; they must now, at least in some cases, store-*compute*-and-forward. This capability can be employed in many ways; examples include DNS proxying, active security measures, self-correcting networks, and use of economic priority models in self-paying information transport.

[1]This research was supported by DARPA under Contracts #N66001-96-C-852, #MDA972-95-1-0013, and #DABT63-95-C-0073. Additional support was provided by the AT&T Foundation, and the Hewlett-Packard and Intel Corporations.

Once nodes can compute on behalf of users, the complexity of node resource multiplexing increases greatly. With an IP router, the bandwidth and buffers are statistically multiplexed; the major computational effort is the determination of the routing table. With an active network element, the computing capacity of the node must be multiplexed in concert with any buffering and bandwidth resources. For advanced applications, the multiplexing discipline must be exposed to programmer control as well.

1.2 Security, Flexibility and Performance

The attraction of active networking is its flexibility. Programmable network nodes provide a high enough level of abstraction that advanced services can be constructed using the network infrastructure itself, rather than overlaying services at hosts. The fact that the nodes are shared between users unfortunately brings with it the fact that users have possibly conflicting or mutually adversarial desires. For example, most users want 100% of the bandwidth the network is capable of; when two such users collide we must multiplex the resource between them. In the case of programmable elements, each user may want to program the element as if it was a dedicated resource. Computation and storage resources must now be multiplexed as well.

These are classic problems faced by computer operating systems. Typical operating systems provide each user with the illusion that they have access to a private dedicated machine. While today's PCs are usually dedicated, their users perform multiple concurrent and independent activities. This is accomplished through the use of a *virtual machine* model; each virtual machine has an abstraction of the real machine which restricts access to certain real resources (such as address tables or I/O devices) in the interest of safely sharing these resources between users. The tradeoffs are, at a high level, between three factors: security, flexibility and performance.

1.3 Hardware, Compiler, or OS?

Software systems have traditionally been employed where complexity has precluded hardware implementation, or where the systems are sufficiently experimental that commitment to silicon would be premature. As the complexity and design choices become understood, the demand for performance causes functions appropriate for hardware implementation to migrate. For example, modern computers support virtual address translation, but all of the policy decisions which use the address translation hardware are implemented in the operating system software.

Security and control of multiplexing at the lowest level lie at the boundary, and debate, between what hardware and software elements of a system should do. For example, should the operating system's schedule be interruptable by device interrupts? How should operating-system defined protection domains be supported with virtual-address translation hardware and exception-signalling mechanisms?

At this point, it is clear that active networks should be studied in software, and the realistic debate is between what roles should be performed by static checks (e.g., by a compiler technology), and dynamic checks (e.g., by a privileged interpreter such as an operating system). As active networks mature, and hardware support evolves, appropriate dynamic checking should migrate into hardware, and more checking will be possible at compile time.

An operating system which supports security in all of its manifestations, including identification, access control and service guarantees, is necessary as a target for any advanced compilation techniques. In the next section, we enumerate the requirements for such an O.S.

1.4 O.S. Requirements

Active network elements combine the requirements of conventional routers with the requirements of secure general-purpose computing nodes. Routers must optimize bandwidth, throughput, and latency. They must be reliable and fail-fast, and must restart rapidly. In commercial environments, routers may be called on to perform firewall functions as well as traffic management. Router application workload is largely preplanned, and resource requirements can be designed accordingly. Persistence, if present, is used for recovery rather than for active execution.

Active network elements must handle dynamically changing workloads, some of which are provided by untrusted third parties. In addition to the basic requirements of routers, active network elements must provide fault containment, general-purpose persistence, explicit management and accounting for persistent, consumable and multiplexed resources, and inter-application protection. Fault containment and recovery is a hierarchical concern; the same mechanisms that protect applications from each other should be exposed to allow intra-application fault containment and protection. If applications are to be internally composed of multiple protection domains (i.e. multiple processes), high-performance interprocess communication (IPC) and a secure resource delegation mechanism is required.[2]

Finally, we assume that active network elements operate in actively hostile environments. For such environments, the security primitives must be subjected to formal analysis. Verification of a formal model should be combined with careful analysis of the implementation to ensure that the active networking substrate is secure. These requirements, and a comparison of the degree to which several contemporary operating systems meet them, are summarized in Table 1.

1.5 Outline of this Paper

Section 2 examines some of the architectural choices for protection, scheduling and resource accounting for active network nodes. Section 3 describes the architecture of the Extremely Reliable Operating System, and shows why and how it makes these design choices. Section 4 then explains how network services such as protocols are implemented under EROS, and after

[2]While the O.S. community tends to focus on protection issues, it should be noted that protection domains are also fault-propagation domains, and as such provide essential support for fault recovery.

Active Networks Requirement	Linux	RT-Mach	Nemesis	EROS
Real-time Support	No	Yes	Yes	Yes
Reserve Delgation	No	No	No	Via capabilities
Network Performance	Medium	Low	Very High	High
Protection	User ID/Paged Virtual Mem.	User ID/Segs + Virtual Mem.	SASOS VM + IDL + User ID	Per-page Capabilities
Protection Composable	No	No	No	Yes
IPC Performance	Low	Medium	High	VERY High
Formally Provable	No	No	No	Yes

Table 1: Summary of active network requirements

explaining the basic programming model, uses an implementation of an Internet Protocol (IP) router under EROS as an example network service. Section 5 evaluates the performance of the IP implementation, and illustrates how effectively an IP router running on EROS can partition resources between domains. Section 6 compares EROS with previous and current related work, and Section 7 concludes the paper with a summary of the results and our assessment of the priorities for future work.

2 Architectural Choices for an O.S. to Support Active Nets

2.1 Access Control for Resources

Two classical mechanisms exist for implementing resource access control: access control lists (ACLs) and capabilities. ACLs bind a list of (user, actions) pairs to each object, and are widely used in current operating systems. An active network element cannot trust the endpoint's authentication service; resource and access controls must be enforced for anonymous (i.e. unidentified) clients. In addition, the resources and access rights associated with multiple anonymous connections need to be isolated and protected from each other. Simply adding an `anonymous` moniker to a user registry is an inadequate solution. In short, ACLs are not well suited to active network protection.

Perhaps more compelling, ACLs are intrinsically insecure. While the resources accessable to a program can be enumerated at any given instant, ACL-based protection systems allow one program to render an object writable by another.[3]. The space of objects accessable to a program is therefore not closed, which renders discussions of fault isolation and confinement largely impossible [13].

A capability is an unforgeable (name, actions) pair. Capability systems embody no notion of user identy; posession of a capability is a necessary *and sufficient* condition for performing the actions authorized by the capability on the object that it names. The state accessible (or mutable) by a program can be determined by performing a reflexive transitive closure on the program's initial capabilities. The resulting graph is closed, which provides the necessary underpinnings for ensuring the confinement of information and program faults (Section 3.4). For active networking, we feel that the security, fault containment, and traceability requirements argue strongly for a capability-oriented primitive protection model.

2.2 Scheduling and QoS in an Active Network Node

Active network elements must provide support for aperiodic, isochronous, and (on suitable link layers) hard periodic transmission guarantees. Given the dynamic nature of the node application set, an active network element must also provide sufficient mechanisms to control and account for CPU consumption, bandwidth consumption, persistent storage, memory working set, and the performance overheads associated with resource virtualization (e.g. paging). To deliver video frames at 60 hz (one frame per 16.6 ms), the application cannot be delayed by disk seeks of several milliseconds. While a high level of abstraction is a practical requirement for programming, the ability to manage low-level resource virtualization is essential. A clean separation of the two concerns is

[3]For example, the UNIX *chmod*(2) system call can render a file visible to a third party via the shared file namespace

essential.

Minimizing latency variance is critical to the performance of current-generation internet protocols such as TCP, and more generally to protocols with super-linear backoff strategies. As a practical matter, active networking elements will sometimes be overcommitted, which manifests as bandwidth and scheduling jitter (i.e variance). End-to-end flow control protocols, such as those used by TCP, are especially sensitive to bandwidth variance.

The work presented here ignores the problem of hard real time guarantees, but provides support for both isochronous and schedule-insensitive applications.

2.3 Reliability and Failure-Tolerance

Network node reliability has both local and global impact on the usability of the network. Network node failures have a disturbing tendency to cascade; systems are most likely to fail when under extreme load. The marginal overhead of the "recovery" strategies frequently induces failure codependence among nodes. Managing this issue requires at a minimum a low failure rate, carefully designed recovery protocols, and pre-reserved bandwidth for recovery related traffic. Depending on the failure interval and period, an unreliable node can affect traffic within the entire network fabric. This especially true in the area of adaptive algorithms; one example is the well-known "route flap" problem. Exploiting such failure modes is an area rife with opportunity for denial of service attacks.

This problem is exacerbated in active networks, where individual applications designed by non-experts may implement their own distributed recovery strategies. At best, one can safely assume that these recovery protocols will be undertested. In an *open* network fabric one must assume that a few will be simply ill-conceived.

2.4 Local Policy Restrictions

In addition to the demands of reliability and QoS, network nodes are subject to the vicissitudes of local administrative policy. The internet, for example, presently operates as a "gift economy," in which parties provide services to each other as a form of barter for mutual benefit. The near-term arrival of various "push" applications such as PointCast™ make it increasingly important for participants to be able to restrict how much of their bandwidth is taken over by third parties and which protocols should be favored or deprecated. An active networking node must provide means by which such local policy restrictions can be applied.

This issue is exmained further in our experiments.

3 EROS Architecture

EROS, the Extremely Reliable Operating System, is a new real-time system constructed at the University of Pennsylvania. Access control in EROS is based exclusively on protected capabilities. Capabilities reside in capability pages (capages) in the same way that bytes reside in data pages. Protection is assured by partitioning. Capability pages cannot be mapped into user address spaces, and therefore cannot be forged by applications. Every object down to the individual data or capability page is named by a capability, and all operations on these objects are authorized by capabilities.

All objects in EROS, including processes, are *persistent*. On a periodic basis, the system efficiently constructs a systemwide snapshot of all dirty objects and asynchronously writes them to the disk. The snapshot overhead is currently under 100ms, and should be closer to 10ms by the time of the first research release of the system.

A detailed description of the architecture [26] and implementation [27] are available elsewhere; the following description describes an application's view of the system.

3.1 Application Metadata

Address spaces and processes in EROS are constructed out of capages and pages. An address space is implemented as a tree of capages whose leaves are pages (Figure 1). Unmapped pages are indicated by a **Number Capability** holding the value zero. Address spaces are named by capabilities; applications may arrange to share memory by exchanging address space capabilities which are then mapped into both address spaces.

Processes are constructed as a suitable arrangement of capages (Figure 2). EROS processes are single threaded, and have kernel-implemented capability registers in addition to the architecture-defined register set.

Because this metadata (the mappings and process structure) is constructed out of persistent storage, the resulting processes and address mappings are persistent. This means that processes may be treated

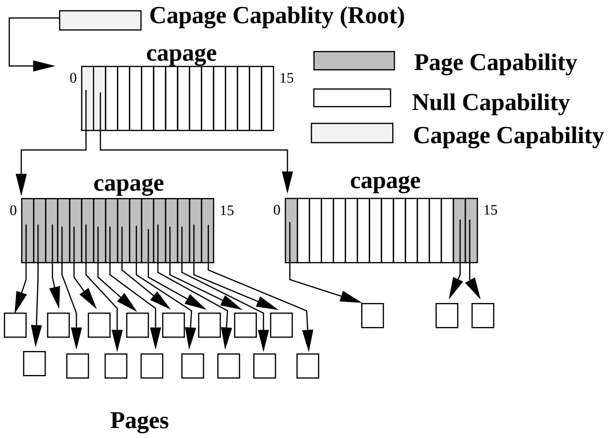

Figure 1: A 19 page address space

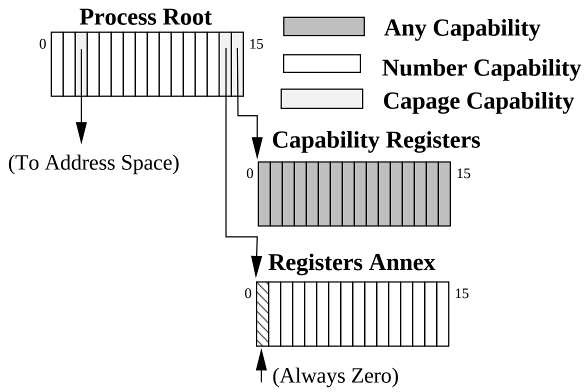

Figure 2: An EROS Process

as reliable objects; EROS has no need of kernel-implemented files or directories, and does not provide them. Similarly, the EROS equivalents to the UNIX *fork*(2) and *exec*(2) system calls are provided by user-level code. For efficiency, the kernel *caches* the state of mappings and processes in machine-dependent form [26].

3.2 Inter-Process Communication

EROS processes communicate via an efficient inter-process communication (IPC) mechanism [27]. To invoke a process, the invoker must possess and invoke a **start capability** to the invokee. Each capability invocation transfers 4 capabilities and up to 64k of contiguous data. The basic invocations are CALL, RETURN, and SEND. IPC operations are *synchronous* and *unbuffered*; the call invocation, typically used to request a service, blocks for a reply; the return operation, used by a service to return a reply to the caller, implicitly waits for the next invocation before allowing the returner to continue. The need for a start capability ensures that only authorized parties may communicate with each other. This requirement, coupled with the absence of a mutable common name space across processes, is essential to the security of the system.

Kernel services are also accessed by performing invocations on capabilities. All invocations are atomic; the invocation either proceeds to completion or does not proceed at all. The kernel is interruptable, but such nested interrupts are simply noted in a bitmask for later handling when the first-level trap exits.

3.3 Reserves

In addition to capabilities for real resources, EROS implements capabilities for consumable resources and resource virtualization in the form of processor capacity reserves and working set reserves.

3.3.1 Processor Capacity Reserves

A **processor capacity reserve** is a 6-tuple of the form: (period, duration, quanta, start time, active priority, inactive priority). The *duration* specifies the number of guaranteed nanoseconds of computation within each *period*, beginning at the given *start time* (immediately if the start time is zero). The *quanta* specifies how often a process running under this reserve will be preempted in favor of other ready processes under the same reserve. The *active priority* gives the priority of this process during its reserved period. The *inactive priority* indicates the priority (if any) with which the process runs when its period has been depleted. Reserves whose inactive priority is below that of the kernel idle thread (which is always ready to run) will not execute outside of their reserved duration.

Processor reserves are allocated by an user-level agent known as the **reserve manager**. The reserve manager accepts requests for reserves, determines if those requests can be met without violating previous reservations, and if so, returns a capability to an appropriate kernel reserve. This reserve capability can be placed in the schedule slot of a process's root capage; all processes sharing a given reserve capability will run under the named reserve.

In contrast to previous implementations [12, 18], Processor reserves are *not* inherited across IPC operations. Priority inheritance raises a host of protection domain violation issues – if reserves are inherited, a caller might call a service and then shut off its reserve, denying service to other callers. Reserve inheritance lends itself to "QoS cross-talk," [15] (which manifests as variance), and understanding the behavior of nested server calls in the face of priority inheritance is difficult. Instead, EROS enables each application to construct a single end-to-end reservation by instancing its services under whatever processor reserve is appropriate.

The EROS CPU reserve implementation differs from previous implementations by unifying reserves and priority management. Depending on the selected admission control mechanism, reserves can be used to provide isochronous scheduling guarantees or softer periodic service at high priority. We have found it occasionally useful to run infrequently executed, unreserved processes at higher priority than the reserved processes, most notably those processes associated with system recovery.

3.3.2 Working Set Reserves

The capacity to use the processor is of little benefit if the application is not in memory. To ensure that performance-critical application code and data remains resident, EROS provides **working set reserves**. A working set describes the number of total pages and capages and the number of dirty pages and capages permitted to the working set reserve. Working sets are subdivisible; an application can fabricate an empty working set and transfer reservation from some existing working set to the new one. A working set may also be marked *non-persistent*, exempting any data in it from the checkpoint mechanism.

Typical applications define a single working set for their process. Applications may also specify an address subspace that operates under its own working set. This enables protocol modules to declare working storage for in-transit packets to be non-persistent, reducing the overhead of restart after checkpoint and reducing the complexity of orderly cleanup after a system restart has occurred.

For purposes of active networking, the role played by constructors is the confinement of suspicious programs. While cryptographic signatures can be used to certify programs by previous arrangement, the provenance of an application is generally unknown, and the active node must presume that such applications are potentially hostile. The solution is to place the incoming application inside a confinement boundary, minimizing mischievious missionism.

4 Implementing Network Services under EROS

EROS implements its networking protocol support as a collection of user processes. In architecting this system, we have tried to balance the objectives of performance, security, and flexibility. Supporting untrusted applications has a definite cost: sharing a memory queue between an untrusted program and a critical service, for example, seems unwise; our architecture is designed to allow us to bypass such checks for instances of trusted programs (e.g. IP) while injecting protective barriers in front of untrusted applications.

4.1 Packet Demultiplexing

The packet dispatcher(s) and all applications share a common region of memory known as the **packet buffer**. This region is divided into equal-sized packet cells and per-client packet arrival queues. Each network interface has associated with it a packet reader/classifier whose purpose is to demultiplex the packet stream to the appropriate applications (Figure 3).

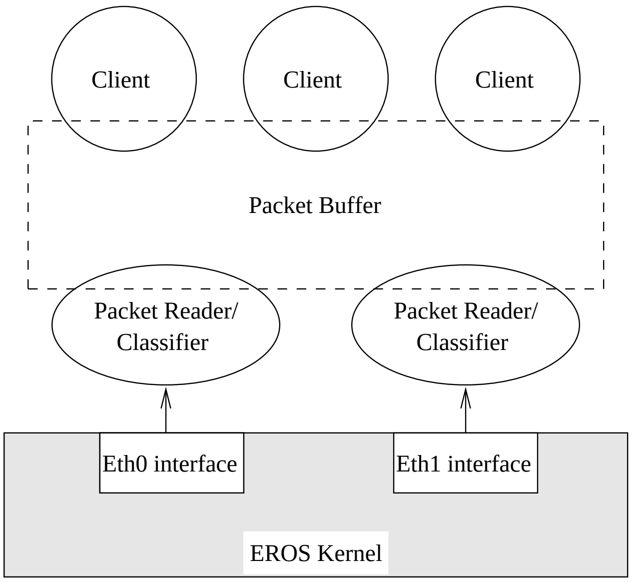

Figure 3: Packet classification

Incoming packets are read directly into the next available cell, and a packet filter is applied to determine which application(s) will receive the packet. The cell is placed directly on the appropriate input queue(s) and the application is notified of packet arrival via an

IPC **send** operation. Ill-behaved or uncertified applications are granted read-only access to the shared packet area, and may be forced to make further copies of the data.

Each client has a reserved number of cells in the packet buffer. When this watermark is exceeded, subsequent packets are discarded by the packet dispatcher. Packets may be discarded on a last-in or first-in basis, according to the requirements of the client. The packet buffer therefore acts as a first round impedance matching mechanism between the input from the network, whose content is unknown and uncontrolled, and the protocol processing applications, whose resources are controlled by the node administrator.

4.2 The Application Initiator

When a new application wishes to become resident in the active node, it contacts the **application initiator**. For each new application, subject to the limits of available storage and local policy, the application initiator provides a limited supply of initial storage, reserved CPU, reserved working set, and a capability to a directory of innocuous service constructors. The application is started in a confined virtual machine that it can abuse without compromising other clients.

Once running, the new application can invoke a trusted service (the packet dispatcher front-end) to request that a new packet filter be inserted into the packet classification map. After checking the validity of the application's requested filter the front-end inserts the new filter into a region of memory shared between itself and the packet dispatcher(s), allowing all dispatchers to use the same set of packet filters.

This interface is adequate for most applications, but further sophistication is required to allow more privileged or more trusted users to obtain greater access to node resources. One of the common services provided in our active netorking element is an authentication service. The application can instantiate an authenticator, present itself to the authenticator instance saying "I am certified by the local administrator, kindly check my digital signature to verify this and give me the rights, powers, and privileges that were designated for me." The authenticator is able both to verify that the application code is certified, and that this code is in fact in control of the application process.[4] This mechanism can be used for remote upgrades, supporting long-term resident applications, applications with unusual degrees of trust (such as our dynamically adaptive bandwidth suballocator), application-specific resource allocation, etc.

[4] The second check is necessary to prevent an attacker from simply carting along a copy of a trusted program to present to the verifier.

5 Evaluation

We have implemented a prototype of the active network router described in Section 4. The prototype implements four clients, each of which routes selected packets (Table 2). Router 3 enforces a bandwidth limiting policy on its packets, allowing us to evaluate the effectiveness of reservation in partitioning available network resources.

Client	Routes	Queue sz	Reserve
Router 1	TCP, port 5100	32 pkts	*varies*
Router 2	TCP, other ports	16 pkts	normal
Router 3	ICMP	8 pkts	normal
Router 4	UDP	16 pkts	normal

Table 2: Experimental clients

To stress this configuration, we set up an isolated network configured as shown in Figure 4. All machines used were single-processor 133MHz Pentiums with 32MB of memory, running RedHat Linux release 4.0 (kernel version 2.0.18) with 3Com 3c509 Ethernet cards. The router ran the latest version of the EROS kernel with native EROS drivers for the EtherLink cards.

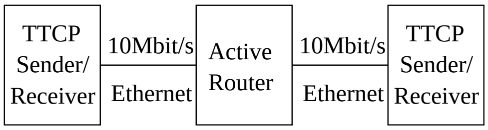

Figure 4: Experimental setup

Latency measurements were collected using pingflood. Throughput and partitioning measurements were conducted using **ttcp**.

5.1 Round Trip Latency

Cross-router latency is a challenge for the proposed architecture, particularly at small packet sizes. The

EROS active networking element must perform two context switches for each packet to implement protection domain boundaries. Conventional routers do not need to provide such boundaries, and therefore do not carry the associated protection domain overhead.

The results, shown in Figure 5, show that our router performs comparably to a Linux system running on the same hardware, optimized for use as a router. The Linux-based router is entirely an in-kernel implementation. It does not provide the protection, untrusted program admission, bandwidth control, or other "active" features of an active networking element.

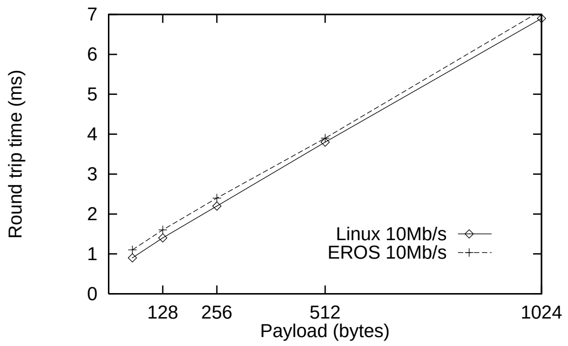

Figure 5: Comparison of flood-ping latencies

The EROS implementation imposes between 0.15 and 0.2 milliseconds of overhead at all transfer sizes. Of this, 0.1 ms is due to an unoptimized IPC implementation, which currently takes 50 μS for each of two invocations. This can be reduced to 2.4 μS per invocation by careful tuning [27]. The remaining difference is the true cost of protection.

5.2 IP Throughput

The second baseline measure of performance is end-to-end throughput. Once again, the active networking element compares favorably to Linux (Figure 6). TCP aggregates data for transfer to reduce processing overhead. At large packet sizes, the majority of the cost lies in data movement. For smaller packet sizes, the protection domain crossings become a substantial fraction of the overall costs. To examine the overhead on smaller packet sizes, we enabled the "push" option of ttcp.

Smaller packet overheads can be amortized by taking advantage of the fact that the underlying network is a multiplexed resource. Our current ethernet driver

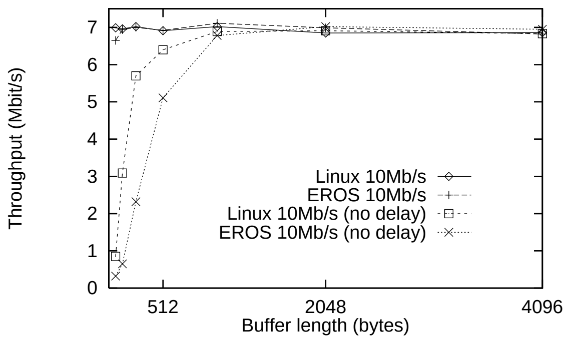

Figure 6: Comparison of ttcp throughput

interface delivers a single packet to the dispatcher on each invocation. An interface that delivers multiple packets with each invocation should improve processing times in the single-stream case shown and allow us to take advantage of the shared nature of the underlying resource.

5.3 Bandwidth Constraint and Partitioning

To measure the effectiveness of reservation-based bandwidth partitioning, we constructed a simple feedback process. At periodic intervals this process estimates the bandwidth used by the constrained client within the last sampling period. If the client has exceeded its allotted bandwidth, its reserve duration is reduced. If the client has not exceeded its allotted bandwidth in the past **N** sample periods then its duration is slightly incremented. It is not necessary to precisely specify how much CPU time is required to restrict an application to a given proportion of the bandwidth; the feedback mechanism converges (surprisingly quickly) automatically on the correct values.

The results (Figure 7) show that the constraint mechanism works well at all packet sizes and constraints. The effective ttcp throughput is approximately 10% below that of the nominal constraint due to retransmissions. This is due to dropped packets, which the previously described multiread interface should resolve.

Having established that we could constrain a stream, the next question was whether the bandwidth could be provided in the face of competition. Given a second (or third or fourth) stream competing for the bandwidth, could we effectively guarantee that a reserved stream would not be crowded out. This may

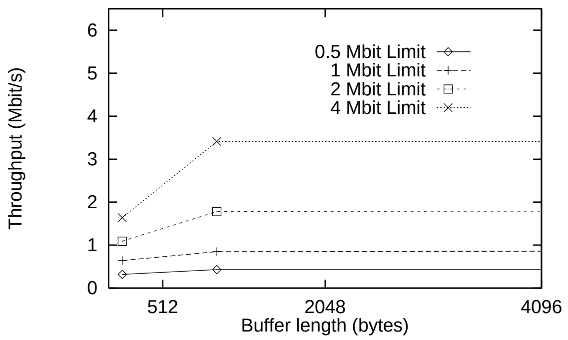

Figure 7: Constrained ttcp throughput

be thought of as using the reserves on the guaranteed stream to enforce flow control on the non-reserved streams by means of artificially induced delay.

For this experiment, we modified **ttcp** to stripe data across multiple TCP ports with throughput measurable either on a per-port basis or aggregated across all ports.[5] The modified ttcp was used to transfer data between the two end machines using the "dedicated port" active routing application (client 1 in Table 2) in parallel with an unreserved active routing application (client 2). All packets from the sender to the receiver are routed by the dedicated client, but return traffic, including acknowledgements, are routed by the unreserved path. The reserved process sees no more than 10% deviation from its allocated reserve at all reservation levels, and in most cases deviation of less than 5% is sustained.

6 Related Work

There has been extensive work done on capability-oriented systems[16], real-time systems[17, 4, 8], etc. over the last forty years. In this paper, we have restricted ourselves to the system requirements of an active network node, as we believe those requirements are both unique and particularly well-met by EROS. However, a number of other approaches to node construction and system construction exist.

6.1 MIT IPv6/Linux

Tennenhouse's team at MIT[28] has designed and implemented an active network node infrastructure based on modifying the IPv4 implementation in a Linux workstation to support the ACTIVE_IP option, which enables active packets ("capsules") to be passed to a computational engine. For the initial prototype, the capsule programming language was TCL, but research is underway to provide a Java computational infrastructure accessed through an IPv6 stack. The experimental system will be implemented using Linux workstations.

This approach will provide a system with considerable flexibility, and may in fact provide acceptable performance. The Java security model, however, is fundamentally flawed, and requires a more secure operating system substrate than Linux to be executed safely. The Java programming model does not reify resource consumption or scheduling concerns, making resource constraint at the programming language level impossible.[6] Their system, however, would be an attractive option to support under EROS.

6.2 Mach/RT-Mach

The Mach operating system is a widely-used operating system research vehicle. Like EROS, it employs a capability-oriented scheme[25], using abstractions such as ports. Attempts at maximizing performance [5] have shown that MACH's networking performance has been inhibited by some internal design decisions; EROS has avoided these performance problems by careful choice of abstraction, and this is reflected in microbenchmarks which demonstrate high interprocess communication performance, as well as macrobenchmarks such as those reported in Section 5.

More important from a resource multiplexing perspective are the specializations of Mach to support bounded resource consumption, such as RT-Mach[29]. RT-Mach was an attempt to both specify and enforce scheduled behavior by applications. The major mechanism to come out of this research was the idea of "Processor Capacity Reserves" developed by Mercer[18, 19]. The CPU reserve mechanism in EROS is builds on this earlier work by fully integrating reservations and priority-based scheduling. As a result, EROS is able to realize "soft" reserves as well as CPU guarantees.

[5] This modification is necessary to coordinate the start of the transmissions, or the delays associated with *fork* and *exec* dominate the measurement.

[6] These issues exist in most programming language approaches to active networks, and that a hybrid approaching leveraging both operating system and programming language techniques is required.

6.3 Nemesis

While targeted more at distributed multimedia support [10] than at the construction of active networks, the Nemesis operating system developed at the University of Cambridge has many aspects in common with EROS [15]. In particular, the choice of an extremely low level at which to control processor multiplexing [24] allows applications much more control of their resource multiplexing [2] than is possible in a system with simple tags such as priorities (e.g., as in one of the many variants on UNIX scheduling). Treating service specifications as security-related (e.g., in the face of denial-of-service attacks on security), their system provides considerable protection against starvation, receive livelock, etc.

We believe that the security and protection domain enforcement of EROS, relying on capabilities, is much more appropriate for the construction of an active network node. The Nemesis protection scheme is based on Roscoe's [24], which employs a single address space to enhance sharing of data and code. Systems are constructed from objects communicating through interfaces specified in MIDDL, an interface definition language used to provide typed interfaces; interfaces are implemented as closures. While there is considerable language support for constructing new domains with interface closures, dynamic linking, etc., there appears to be a basic presumption that the application code is both trusted and reliable. EROS makes no such presumption; our IP routing module operates as an untrusted application.

7 Conclusion

7.1 Summary of results

We have demonstrated a working basis for a, secure active network element, and shown that objectives of security and quality of service need not be mutually exclusive. The system presented implements all of the necessary trust boundaries to run untrusted protocol modules (e.g. our IP routing modules). It delivers performance that meets or exceeds that of conventional network elements while providing a fundamental enhancement in the power of the networking substrate. Given a substrate such as the one described here, it becomes possible to tightly marry the application to the network, enabling a new generation of network-aware applications.

Taking the existing EROS application tool set as a basis, the entire active networking application suite is implemented in less than 5,000 lines of portable C code.

7.2 Future work

The application initiator described here is incomplete. The remaining effort is a matter of integrating well-understood components such as certificate-based authentication with EROS's existing confinement technology. Once this has been fully implemented, it will be possible to extend this work to demonstrate remote, secure administration and to download application-specific network intelligence (e.g. MBONE) as untrusted components.

While the security and fault containment provided by the EROS substrate are powerful tools for the construction of reliable and recoverable applications, they do not fully solve the problem of application reliability. Further work in programming languages and formal methods necessary to advance the state of the art in these areas. In addition, the proper service architecture of an active network element remains a fertile area for further work.

Dawson Engler and others have put extensive work into the area of efficient packet filtering [7]. It has been our intention from the start to incorporate this work into our implementation.

7.3 Status and Availability

The system described in this paper is currently running, and has effectively replaced one of the routers in our laboratory. Since it is more reliable than the surrounding machines (indeed it has yet to crash), it does not appear to present an impediment to network usability.

Early public access to the EROS system is currently scheduled for 2Q97. To join the announcement list, or to obtain further information, please see the EROS home page at `http://www.cis.upenn.edu/~eros`

8 Acknowledgments

Many of the ideas in EROS are descendants of KeyKOS, and the KeyKOS design team, especially Norm Hardy, Bill Frantz, and Charlie Landau have been active participants in electronic discussions preceding commitment to a particular solution used in EROS. In addition, Bryan Ford of the University of Utah has actively participated in the design discussions that led to the current system.

Mike Laskin, Geoff Milord, and Mike Berry have been putting together a binary-compatible Linux environment atop EROS, and their early discovery of implementation issues has saved us a great number of headaches in the current work.

Sam Weber and Angelos Keromytis provided feedback on earlier drafts that rescued us from several mistakes. Any remaining errors, of course, are entirely our own.

References

[1] B. Bershad, *et al.*,, "Extensibility, Safety and Performance in the SPIN Operating System," Proc. 15th SOSP, pp. 267–284, December 1995.

[2] "Explicit Network Scheduling," R. J. Black, Ph.D. Dissertation, University of Cambridge Computer Laboratory Technical Report No. 361, April 1995.

[3] D. D. Clark, V. Jacobson, J. Romkey and H. Salwen, "An Analysis of TCP Processing Overhead," in *IEEE Communications Magazine*, **27**(6), June 1989, pp. 23–29.

[4] "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism," D. D. Clark, S. Shenker and L. Zhang, in *Proceedings, SIGCOMM '92 Conference*, pp. 14–26, Baltimore, MD, August 17-20, 1992

[5] "Fbufs: A High-Bandwidth Cross-Domain Transfer Utility," P. Druschel and L. Peterson, in *Proc. 14th SOSP*, December 1993.

[6] D. Engler, *et al.,* "Exokernel: An Operating System Architecture for Application-Level Resource Management," Proc. 15th SOSP, 1995.

[7] "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," D. Engler and M. F. Kaashoek in *Proceedings, SIGCOMM '96 Conference*, Stanford, CA, August 28-30, 1992

[8] "RealTime Communication in an Internetwork," D. Ferrari, in *Journal of High-Speed Networks*, Vol. 1, 1992, pp. 79–103.

[9] "The KeyKOS Architecture" Norman Hardy in *Operating Systems Review*. Oct. 1985, pp 8-25.

[10] "Operating System Support for Quality of Service," E. Hyden, Ph.D. Thesis, University of Cambridge Computer Laboratory Technical Report No. 340, February, 1994.

[11] "AIX Version 3.1. RISC System/6000 as a Real-Time System", *Technical Documentation Number GG24-3633-0*, International Technical Support Center, Austin, Texas.

[12] "Integrated Management of Priority Inversion in Real-Time Mach." Takuro Kitayama, Tatsuo Nakajima, Hiroshi Arakawa, and Hideyuki Tokuda. *IEEE Real-Time Systems Symposium.* December 1993

[13] "Reflections on an Operating System Design." Butler W. Lampson and Howerd E. Sutrgis. *Communications of the ACM* Vol 19, No 5, May 1976.

[14] Charles R. Landau. "The Checkpoint Mechanism in KeyKOS," *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* IEEE. September 1992. pp 86-91.

[15] "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden in *IEEE Journal on Selected Areas in Communications* September 1996, **14**(7), pp. 1280–1297.

[16] "Capability-Based Computer Systems," H. M. Levy, Digital Press, 1984.

[17] "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", C. L. Liu, J. W. Layland, in *Journal of the Association for Computing Machinery*, **20**(1), January 1973, pp. 46–61.

[18] "Processor Capacity Reserves: An Abstraction for Managing Processor Usage," C. W. Mercer, S. Savage and H. Tokuda, in *Proc. 4th Workshop on Workstation Operating Systems*, October 1993.

[19] "Processor Capacity Reserves: Operating System Support for Multimedia Applications," C. W. Mercer, S. Savage and H. Tokuda, in *Proc. IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[20] "The Packet Filter: An Efficient Mechanism for User-level Network Code," J. Mogul, R. Rashid and M. Accetta, in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 39–51, Austin, TX November 8–11, 1987.

[21] "The BSD Packet Filter: A New Architecture for User-level Packet Capture," S. McCanne and V. Jacobson, in *Proc. USENIX Technical Conference*, January 1993, pp. 259–269.

[22] A. B. Montz, *et al.,* "Scout: A communications-oriented operating system," Technical Report 94-20, Dept. CS, University of Arizona, June 1994.

[23] "Design, Implementation and Experiences of the OMEGA End-Point Architecture," K. Nahrstedt and J. M. Smith, in *IEEE Journal on Selected Areas in Communications (Special Issue on Multimedia Systems)* **14**(7), September, 1996, pp. 1263–1279.

[24] "The Structure of a Multi-Service Operating System," T. Roscoe, University of Cambridge Computer Laboratory, (Ph.D. Thesis) Technical Report No. 376, August 1995.

[25] "Extending a Capability Based System into a Network Environment," R. D. Sansom, D. P. Julin and R. F. Rashid, in *Proceedings, 1986 ACM SIGCOMM Conference*, August 1986.

[26] "State Caching in the EROS Kernel – Implementing Efficient Orthogonal Persistence in a Pure Capability System," Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith, in *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J. 1996

[27] "The Measured Performance of a Fast Local IPC," Jonathan S. Shapiro, David J. Farber, Jonathan M. Smith, in *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, Seattle, Washington, November 1996, IEEE

[28] "A Survey of Active Network Research," D. L. Tennenhouse, *et al.*, in *IEEE Communications*, January, 1997 **35**(1), pp. 80–86.

[29] "Real-Time Mach: Towards a Predictable Real-Time System," H. Tokuda, T. Nakajima, P. Rao, *Technical Report* School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1993.