An Efficient End-host Architecture for Cluster Communication Services

Xin Qi, Gabriel Parmer and Richard West

Computer Science Department
Boston University
Boston, MA 02215
{xqi, gabep1, richwest}@cs.bu.edu

Abstract

Cluster computing environments built from commodity hardware have provided a cost-effective solution for many scientific and high-performance applications. Likewise, middleware techniques have provided the basis for large-scale applications to communicate and exchange data across the various end-hosts in a distributed system. Unfortunately, middleware services are typically encapsulated in user-level address spaces that suffer from scheduling delays and communication overheads induced by the host kernel. For various high performance distributed computing applications such overheads are unacceptable. This paper therefore addresses the problem of providing an efficient end-host architecture to support application-specific communication services at user-level, without the need to explicitly schedule such services or copy data via the kernel.

We briefly describe a sandboxing mechanism that allows applications to configure and deploy services at user-level, that may execute in the context of any *address space. Using Linux as the basis for our approach, we focus specifically on the implementation of a user-space network protocol stack, that avoids copying data via the kernel when communicating with the network interface. Our approach enables services to efficiently process and forward data via proxies, or intermediate hosts, in the communication path of high performance data streams. Unlike other user-level networking implementations, our method makes no special hardware requirements. Results show that we achieve a substantial increase in throughput, and a reduction in jitter, over comparable user-space communication methods.*

1. Introduction

Cluster computing environments have proved a cost-effective solution to many distributed computing problems by leveraging inexpensive hardware. Likewise, middleware implemented on general-purpose systems has provided the basis for large-scale applications to communicate and exchange data across the various end-hosts in a distributed system. Unfortunately, general-purpose systems provide a generic set of abstractions that are not well-suited for efficient middleware services needed by high performance applications. Specifically, abstractions such as BSD sockets and kernel-based network protocols are common to modern systems, but they are not tailored to the needs of applications that require low latency, high-bandwidth communication (e.g., involving real-time data streams). These mechanisms are necessarily general so that they can provide fair, consistent, and simple abstractions of the base hardware to all application processes. With this generality, fine grained control is sacrificed. For instance, there is little support for using efficient custom communication protocols in common systems. These generic mechanisms do not provide the power to efficiently utilize modern networking systems such as Gigabit Ethernet [22], ATM [12], and Myrinet [6].

To efficiently use many of the modern high throughput, low latency networking systems, specialized approaches must been taken that depart from the traditional operating system abstractions. Streamlined network processing stacks, zero-copy data movement, and asynchronous network processing are now seen as necessary for the highest degree of networking performance [25, 26]. A minimal networking path is important to reduce the latency involved in communication. For example, zero-copy techniques avoid superfluous memory usage, while execution at interrupt time avoids unnecessary scheduling overheads.

Many systems have been devised that combine a number of the above optimizations to provide a platform for high demand communication [10, 19, 25]. Most of these systems, however, require non-trivial changes to their host kernel and specific functionality built into the hardware of the network interface to achieve enhanced performance. This is acceptable when the environment is controlled, as is the case in a research or government lab for scientific applications, but

not feasible on the scale of the Internet where commercial-off-the-shelf (COTS) systems must be considered.

The basic objective of this paper is to provide a method by which efficient middleware services can be implemented on COTS systems. In providing such a method, we wish to empower applications with the ability to configure and deploy services for their specific needs, while avoiding the traditional costs associated with general-purpose systems. Specifically, the aim is to provide support for the construction of services at user-level that have the same capabilities and privileges of traditional kernel services, with the exception that the kernel may always revoke access rights to any service abusing its capabilities. In effect, an efficient mechanism is required with which an application can receive more control of the underlying hardware while still maintaining safety and isolation. Just as a kernel service may be invoked without scheduling overheads (e.g., at interrupt-time) and may directly access hardware, certain user-level services should be treated similarly.

In effect, this is related to work in the area of extensible systems, that allow applications to customize services for their specific needs. Some systems allow untrusted application extensions to be executed in the context of the kernel, by enforcing safety using type-safe languages [4]. Others implement minimal kernels that provide interfaces for higher-level abstractions in user-space [11]. Linux provides the ability for trusted code to link directly into the address space of the kernel, without protection. Each of these mechanisms permit a greater degree of control of the hardware than would be otherwise available.

As part of the work presented in this paper, we introduce a sandboxing mechanism [29] to support efficient and configurable user-level services. We use this mechanism to implement a user-level network stack that can be executed at interrupt-time, and that has regulated access to the network interface. Increased control of the network interface card (NIC) allows for zero-copy communication which reduces memory bus usage. Even with these privileged capabilities, sandboxed services execute at user-level without access to the kernel address space. Furthermore, because they execute in the application domain, they may link with user-level libraries. By allowing the kernel to control access rights to user-level sandboxed services, regulated access to I/O devices is guaranteed, whereas this would be difficult to achieve with extensions executing in the kernel.

In summary, the contributions of this paper center around support for efficient user-level implementations of network services. The aim is to provide a means by which high performance distributed computing applications can customize network services for their specific needs. We leverage our ongoing work on *user-level sandboxing* to support the implementation of a network subsystem in user-space that avoids unnecessary intervention of the kernel. Moreover, our approach allows high performance applications to communicate with the network interface without either: (1) the need to copy data via the kernel, or (2) scheduling overheads. This is achieved on commonly available hardware. We compare various implementations of a networking stack, traditionally implemented in the kernel, that forward data between end-hosts in a distributed system. This scenario would be applicable for efficient peer-to-peer routing of high bandwidth data streams, or in situations where a proxy server is handling remote procedure calls. Results show our system is flexible enough to allow applications to customize networking services for their specific needs, while providing efficient throughput comparable to kernel-level methods of forwarding network data.

The remainder of the paper is organized as follows: Section 2 briefly describes the sandboxing mechanism required for our user-level networking services. Section 3 then discusses the implementation of a networking stack in a user-level sandbox. This is followed by Section 4 that compares the performance of various networking implementations. Finally, related work is described in Section 5, followed by conclusions and future work in Section 6.

2. User-level Sandboxing

Our *user-level sandboxing* mechanism [29] is the basis for an extensible end-host architecture. It involves modification to the address spaces of all processes, or logical protection domains, so they contain one or more shared pages of virtual memory. The virtual address range shared by all processes provides a sandboxed memory region into which extensions may be mapped. Under normal operation, these shared pages will be accessible only by the kernel. However, when the kernel wishes to pass control to an extension, it assigns user-level privileges to the shared page (or pages) containing the extension code and data. This prevents the extension code from violating the integrity of the kernel. The extension code itself can run in the context of *any* user-space process, even one that did not register the extension with the system, thereby eliminating scheduling overheads.

There is potential for corrupt or ill-written extension code to modify the memory area of a running process. To guard against this, we require extension code registered with the system to be written by a trusted programmer. By virtue of running at user-level, the kernel itself is always shielded from any extension software faults.

2.1. Hardware Support for Memory-Safe Extensions

Our approach assumes that hardware support is limited to page-based virtual memory (i.e., processors with an

MMU). [1] This minimum hardware requirement is met by many processors made today. These relaxed requirements will allow wide deployment across a heterogeneous environment such as the Internet.

On many processors, switching between protection domains mapped to different pages of virtual (or linear) addresses requires switching page tables stored in main memory, and then reloading TLBs with the necessary address translations. Such coarse-grained protection provided at the hardware-level is becoming more undesirable as the disparity between processor and memory speeds increases [23]. This is certainly the case for processors that are now clocking in the gigahertz range, while main memory is accessed in the 10^8Hz range. In practice, it is clearly desirable to keep address translations for separate protection domains in cache memory as often as possible. User-level sandboxing avoids the need for expensive page table switches and TLB reloads by virtue of the fact that the sandbox is common to all address spaces.

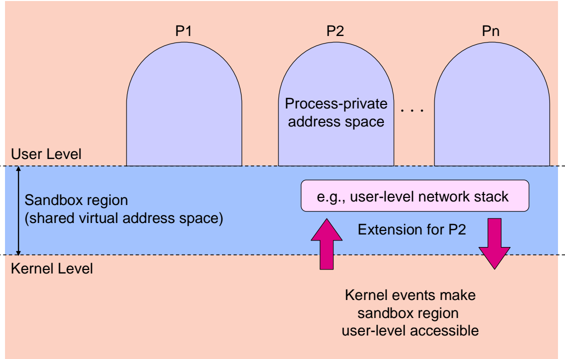

Figure 1. User-level sandboxing: each process address space has a shared virtual memory region for mapped extensions, activated by upcalls from the kernel.

With user-level sandboxing (Figure 1), each process address space is divided into two parts: a conventional process-private memory region and a shared, but normally inaccessible, virtual memory region. The shared region acts as a sandbox for mapped extensions. Kernel events, delivered by upcalls to sandbox code, are handled in the context of the current process, thereby eliminating scheduling costs.

2.2 Sandbox Regions

In our current implementation on the Intel x86 processor, the sandbox consists of two $4MB$ regions of virtual memory that are identically mapped in every address space to the same physical memory. For convenience, the two regions are assigned to adjacent (extended) page frames. That is, regions employ the x86 page size extensions and each occupy one 4 megabyte page directory entry [2].

In this paper, the two $4MB$ regions form a single larger region that is permanently assigned read-write permission at kernel-level, but by default is inaccessible at user-level. The sandbox memory area can be made accessible to user-level by toggling the user/supervisor flags of the corresponding page directory entries, and invalidating the relevant TLB entries. This is only allowed when an upcall occurs from the trusted kernel.

2.3 Sandbox Threads

If code in the sandbox is allowed to invoke system calls, it is possible for an extension registered by one process to block the progress of another process. For example, if process p_i registers an extension e_i that is invoked at the time process p_j is active, it may be possible for e_i to affect the progress of p_j by issuing 'slow' system calls. The current solution to this problem is to execute extensions that issue blocking system calls in their own thread context. Sandbox threads have scheduling costs comparable to those of kernel threads [29].

2.4 Pure Upcalls

Traditionally, signals and other such kernel event notification schemes [2, 17] have been used to invoke actions in user-level address spaces when there are specific kernel state changes. Unfortunately, there are costs associated with the traversal of the kernel-user boundary, process context-switching and scheduling. The aim is to implement an upcall mechanism with the speed of a software trap (i.e., the mirror image of a system call), to efficiently vector events to user-level where they are handled by service extensions, in an environment that is isolated from the core kernel. A *pure upcall* occurs when a sandbox extension is invoked without any scheduling costs. It should be noted that sandbox extensions cannot be invoked other than via the trusted kernel.

3 User-level Networking

The motivation for this work is to support high performance distributed applications that require system services configured for their specific needs. In meeting this goal, we have implemented an entire networking subsystem in a user-level sandbox that can be customized to support application-specific protocols and services. This section describes the issues involved in constructing this subsystem.

[1] A series of caches, most notably one or more untagged translation look-aside buffers (TLBs) is desirable but not necessary.

[2] The 32-bit x86 processor uses a two-level paging scheme, comprising page directories and tables.

- Memory management: Memory resources in the sandbox area must be managed efficiently to oversee packet placement and to ensure all allocations occur within the sandbox. The `malloc` interface provided with glibc is not satisfactory for this high performance role. It is beneficial to have a slab allocator [7] that has apriori knowledge of objects such as packet descriptors (`sk_buff_heads`).
- Kernel bypass: An abstraction for passing control to an extension via a bottom half asynchronous execution path must exist. This abstraction must include hooks which are executed during packet reception in the kernel, to trigger execution in the sandbox. This allows the network execution path in the kernel to be bypassed in favor of the more specialized extension's code. The abstraction allows insertion of customized network code into the critical networking path. This is seen as primary to obtaining high networking performance [25, 10].
- NIC interaction: An interface between the network interface card (NIC) and the sandbox allowing packets to be received into, and sent from, the sandbox directly using 'Direct Memory Access' (DMA). This is essential in data stream processing because the packets will be very large, and we will benefit from the zero-copy afforded to us by this interface.

In consideration of all these criteria, we chose User-mode Linux (UML) [24] as the basis for network service extensions. UML is, in essence, the Linux kernel ported to user-space. It is a type of virtual machine that executes as an application on top of a host Linux kernel. All of the hardware of a real machine is emulated using the system-call, signaling, and ptrace interfaces of the host kernel. It provides all of the services that Linux itself does including, most importantly, memory allocation, a modular device interface, and a fully functional, well tested, efficient and modular networking stack. Both UML and extensions to its networking code can be loaded into the sandbox. Furthermore, because UML is a user-space application, it is much easier to port to the sandbox than a normal kernel. UML satisfies the memory management requirements with its internal `kmalloc`-based interfaces. Using UML allows us to manage the memory resources in the sandbox with the same efficiency and granularity as in the host kernel. A benefit of this decision is that memory is strictly controlled within the $8MB$ sandbox region.

The design choice to use UML makes the conditions for kernel bypassing easily satisfiable. Driver abstractions exist within the Linux kernel, and therefore in UML, to receive and transmit packets. These abstractions make it simple to write a driver in UML that functions as an intermediary between the host kernel bottom half and the extension networking stack. It is only because the sandbox is mapped across all process address spaces that it is possible to support efficient bottom half execution. Typically, it would not be efficient or desirable to execute a user-level handler in the context of a bottom half because it is impossible to guarantee that a process's virtual address space will be currently loaded. Executing the process's code could require a context switch. This is a costly operation to do for every interrupt caused by network hardware. Similarly, scheduling decisions based on process priorities may ordinarily cause unbounded delays before a specific address space is executed. Such issues are eliminated using user-level sandboxing.

UML, however, does provide abstractions and mechanisms that are unnecessary in a sandbox environment. Namely, the UML analogy of user-level processes are superfluous to our purposes of providing a customizable networking stack in the sandbox. Only a few hundred lines of UML code needed alteration to eliminate unwanted 'virtual' user-level processes, and include a specific driver to interface with the host kernel.

Because extensions running in the sandbox cannot issue privileged instructions, they cannot directly modify or influence the networking hardware to copy packets directly to and from the sandbox using DMA. Instead, the host kernel and sandbox interface through a well defined set of communication channels to pass the desired memory location for packet arrival or transmission. In this way, all protected instructions and all kernel memory remains protected from sandbox extensions. Independent of which process is running at any given moment, the networking card can DMA directly to or from the sandbox region. This type of communication is only possible because the sandbox exists in every process virtual address space. Consequently, no special requirements are placed on the networking hardware. We exploit the DMA capabilities common to modern networking cards, but this is not a requirement of our user-level networking subsystem. The aim is to deploy our end-system architecture on an Internet-scale, so support for diverse hardware is important.

In this manner the NIC interaction criteria for providing a high performance, user-level, networking interface is fulfilled. This interface is leveraged to provide a minimal copy capability where packets coming in from the network can be copied using DMA directly into the sandbox and, after processing by the customizable stack, can be copied directly back to the network. This ability has been used to provide a direct proxying service within the sandbox. Consequently, host-based processing and routing of packets is possible without unnecessary copying via the kernel.

Demultiplexing: The demultiplexing of packets is one of the challenges when developing a user-level networking stack. Other technologies rely on programmable NICs which have apriori knowledge of the destination of incoming packets so they can transfer them directly to the correct

destination. We do not take this approach because we wish to target non-specialized NICs. Instead, a light-weight classifier can be written either in the lowest levels of the host kernel or in the sandbox, which then dispatches packets appropriately. In either case, all incoming packets must still be allocated and transfered (perhaps via DMA) to the sandbox address range. The sandbox networking scheme is not intended to be an architecture for the efficient processing of *all* packets. Rather, we restrict efficient user-level processing to those packets with corresponding network service extensions. This is not a serious limitation because it is unlikely that *every* networking stream will require customizable high-performance communication services.

3.1. Control Flow With Sandboxed Networking

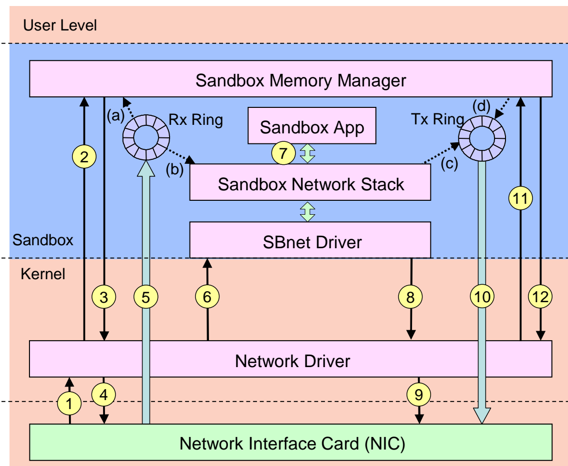

Figure 2. User-level asynchronous networking in a sandbox.

The control path, for the case when sandboxed network services are invoked asynchronously with respect to the current execution context, is shown in Figure 2. This is the control path experienced by pure upcalls into the sandbox, that are executed in the context of the active address space at the time of the upcall. We modified the kernel network driver so that packet processing and interaction with the NIC (via the SBnet driver) take place in the sandbox. That said, the various stages of asynchronous computation involving the networking stack are as follows:

1. When a packet is received by the NIC, an interrupt service routine (top half) is invoked in the network driver. This is a basic notification that a packet is ready and a minimal amount of processing is undertaken. No modifications to the top half of the default driver are made, so that it remains as efficient as possible.
2. When the top half returns, interrupts are re-enabled and bottom half execution proceeds in the network driver. Space is then allocated from a receiver ring buffer (label (a)) in the sandbox, by an upcall that directly invokes the sandbox memory manager.
3. The return address of this allocation is passed to the network driver. A check is performed to verify that the memory location is within the sandbox region.
4. The network driver informs the NIC of the location into which it can DMA the packet. Because this network driver is executed in the kernel domain, it has full I/O permissions for trusted communication with the NIC.
5. The NIC copies the received packets into the allocated sandbox memory using DMA.
6. After the new packet is resident in the sandbox, an upcall to the SBnet driver and, hence, the protocol stack occurs. Packets can be accessed from the ring buffer (label (b)) in the context of a bottom half, so execution is unaffected by host scheduling. Recall that when a pure upcall is triggered, the handler runs with user permissions.
7. At this point in the configurable networking stack, application specific handlers can execute. For example, we provide an application that performs transport-level forwarding of packets to another end-host.
8. After the network stack's processing is complete, the memory address of a packet awaiting transmission is placed in an outgoing buffer (label (c)). Control then returns to the kernel.
9. Now full I/O permissions are restored, the NIC is notified of the packet it should transmit.
10. The NIC uses DMA to retrieve and send the packet onto the network.
11. After the DMA is complete, the network driver notifies the sandbox extension that it can free the memory formerly taken up by the packet (as in label (d)).
12. Upon return from this pure upcall, we have completed the bottom half and can return to the previously executing thread.

Though this entire control path seems complex, it is highly optimized and yields significant performance improvements over conventional user-space network protocol stacks confined to process-private address spaces. As will be seen in Section 4, experimental results confirm the benefits of our approach.

If the network control path attempts to block when requesting a timeout, as is the case in TCP processing, then execution cannot be completed in a totally asynchronous manner. We support synchronous network processing by allowing any control path that may block to continue in the context of its own sandbox thread. Recall that a sandbox thread shares the page tables of the currently running process, but has its own execution state, thus having relatively inexpensive context switching costs. This processing

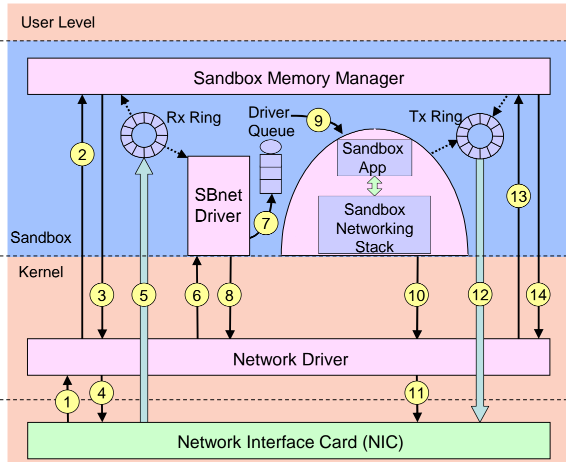

Figure 3. User-level synchronous networking in a sandbox.

model is demonstrated in Figure 3. It is similar to the asynchronous communication model demonstrated in Figure 2 except that after a limited amount of interrupt-time processing, packets are placed into a queue (as shown in step (7)). Control returns back to the kernel, as in step (8). A sandbox thread is scheduled in step (9), to complete the network processing of the packets in the queue. Steps (10) to (14) are essentially the same as steps (8) to (12) in Figure 2.

4. Experimental Evaluation

This section describes experiments on a cluster of 8 IBM x series 305 e-servers, each with a tigon3 gigabit Ethernet card, interconnected by a gigabit Ethernet switch. Each machine has a 2.4GHz Pentium IV CPU and 1024M RAM, running the Linux 2.4.20 kernel.

4.1. UDP Forwarding

We use a simple *UDP forwarding agent* to test the performance characteristics of our implementation. This application forwards UDP packets for specific streams of data from one host, which we will call *A*, through the forwarding host, *B*, to the receiving host, *C*. More generally, an application-specific service could be used to forward data along a P2P overlay [3], or over a grid infrastructure [13]. Nonetheless, our UDP forwarding agent is similar to that found in web proxies such as Squid [21], and serves to demonstrate the high performance capabilities of our end-host architecture.

To generate traffic and measure throughput and jitter, we used the *Iperf* network performance tool [16] . Iperf generates packets at the source, *A*, and measures the perceived throughput at the destination, *C*. Note that with UDP forwarding, the perceived throughput can be affected by lost packets e.g., at host *B*. All bandwidth figures shown in this section are of the perceived throughput. Iperf also measures the jitter of the arrival time of the packets. That is, it measures the deviation from the average transfer time of each of the packets and it maintains a running total of this average jitter.

4.2. Comparison of Networking Implementations

User-level Networking: The first set of experiments compare the throughput of two different UDP forwarding agents running on host *B*. One agent involves User Mode Linux mapped into a conventional process address space, while the other involves UML mapped as an extension in the sandbox. MTU sized packets are routed from *A* to *C* and the throughput is noted. The process-based forwarding agent is similar to our sandboxed approach, although processing and routing is done in the UML equivalent of a bottom half, rather than a host kernel bottom half. Additionally, the process-based agent must be scheduled by the host kernel and cannot take advantage of the zero-copy benefits of the sandbox approach. In this scenario, the sandbox approach uses an asynchronous processing model, as in Figure 2.

Figure 4 shows the throughput corresponding to both of these cases. Background threads, each mapped to a separate process address space, are run to measure the effect of system load on performance. These background threads are simple while loops with minimal working-set sizes. It is important to measure the throughput of our methods in the presence of background threads, because we target COTS systems on the Internet that could be performing other simultaneous tasks.

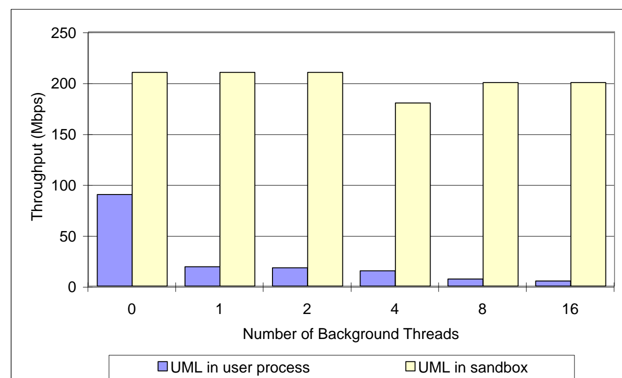

Figure 4. Throughput comparison of UML in a sandbox versus user-process using an asynchronous processing model with UDP.

The results show that with no background threads, the sandbox forwarding agent demonstrates an improvement of 130% over the process-based approach. With more background threads, the process-based agent has more competition for the CPU and suffers from increased scheduling delays. The sandbox agent does not suffer from scheduling delays and therefore maintains high throughput irrespective of the number of background threads.

User- versus Kernel-Level Networking: The previous experiments demonstrated that with minimal porting effort, a user-level code base can be made to execute in the sandbox, thus reaping performance benefits. However, using UML for sandbox network services is not necessarily the best approach for high-performance communication, because of virtualization costs.

UML virtualizes interrupts using signals from the host kernel. When synchronous access to a shared resource is required, UML makes a system call that blocks the delivery of signals, thereby disabling virtualized interrupts. However, our approach allows for delivery of synchronous upcalls from bottom halves, in place of more expensive asynchronous signals. Since only one bottom half (softirq in Linux) can execute per processor at any one time, the synchronization-related system calls within UML are unnecessary.

After removing synchronization system calls, performance increases by nearly a factor of three as can be seen in Figure 5. Here, we compare the throughput of the sandbox networking stack with two other implementations: (1) a simple forwarding agent that uses a kernel thread to send from one socket to another without copying the data (denoted 'kernel' in the figure), and (2) a user-level application that simply transfers data between a pair of sockets (denoted 'socket' in the figure). It should be noted that the user-level application is mapped to its own process address space, similar to conventional middleware approaches.

The same testing environment as before is maintained and maximum throughput from *A* to *C* through *B* is measured with a certain number of background threads. One can see that the sandboxed networking stack's bandwidth remains nearly constant as the number of processes increases, whereas with the other two approaches this is not the case. Observe that the kernel approach still requires a thread to be scheduled, and so there is competition for the CPU with background threads. Even with no background threads, our user-level sandboxing approach performs better than conventional middleware, and only slightly worse than the kernel approach.

4.3. Transfer Time Jitter

Jitter measures the variation of the transit time of the packets over a period of time. The reduction or elimination of jitter is especially important in environments which require quality of service (QoS) constraints to be met. If network performance is unpredictable (i.e., high jitter is present) then guaranteeing QoS constraints becomes increasingly difficult if not impossible.

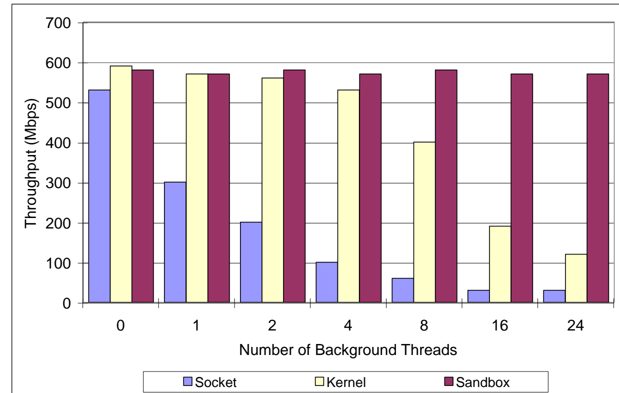

Figure 5. UDP Throughput comparison of an optimized sandbox stack versus both user-level sockets and kernel implementations.

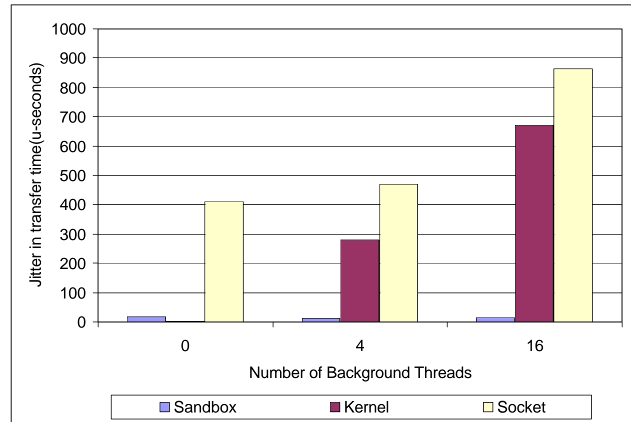

Figure 6. Maximum jitter in transfer time.

Running in the context of bottom halves gives the sandbox pure upcall code the ability to immediately process each incoming packet, which results in a very small amount of variation in the transfer time of those packets. In contrast, the kernel and process-based forwarding agents must suffer from scheduling delays. The amount of deviation from the average transfer time is a function of the size of the scheduler's run queue. Figures 6 and 7 show that a nearly constant amount of jitter is demonstrated by the sandboxed networking scheme, while the other two approaches show larger and more variable jitter as the number of background threads increases.

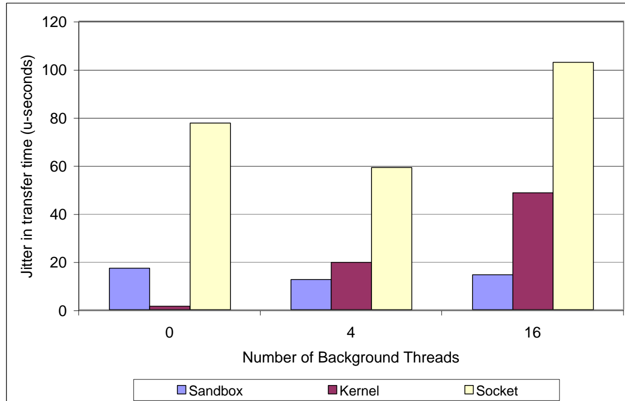

Figure 7. Average jitter in transfer time.

4.4. Microbenchmarks

Operation	Cost in CPU cycles
Null Pure Upcall	1370
Sandbox Packet Processing Time	6360
Kernel Packet Processing Time	4800

Table 1. Sandbox Overheads.

Table 1 shows microbenchmarks measured by using the x86 timestamp counter. The round-trip time for a pure upcall that jumps to the address of a sandbox function and immediately returns to the kernel is 1370 clock cycles. Due to various optimizations that avoid identifying the process responsible for registering an upcall function, this value improves upon the costs of upcalls in our earlier work [29].

The second value in Table 1 measures the time it takes to process a packet using our user-level networking code. A measurement is taken upon reception of a packet and again when we transmit that packet. This overhead compares favorably to the cost of executing a network bottom half handler in the kernel. Hence, the overheads of using our sandboxing scheme do not impose excessive costs on the implementation of network services at user-level.

4.5. TCP Forwarding

This section shows the performance of sandbox networking support for synchronous TCP forwarding. Part of the network processing is done in the context of a sandbox thread, as described in Figure 3. A series of experiments measure the achievable throughput of an end-host while servicing a number of background threads. As before, a conventional process-based implementation of User-mode Linux is compared to a sandboxed network approach, to forward packets. Additionally, we compare the sandbox thread approach using the POSIX.4 normal (SCHED_OTHER) and real-time (SCHED_RR) scheduling policies. In all cases, throughput results were captured using *wget* to receive data from an *Apache* web server through an intermediary node similarly to the UDP testing environment.

Figure 8 shows that the sandbox implementation can achieve as much as 30% better throughput than the typical UML implementation when performing synchronous processing in the context of a sandbox thread. Further, it can be seen that if we prioritize the network thread (using SCHED_RR with the highest real-time priority), even more than 50% higher throughput can be attained, irrespective of the number of background threads. This is similar to the case in which UDP forwarding is performed in the context of a bottom half.

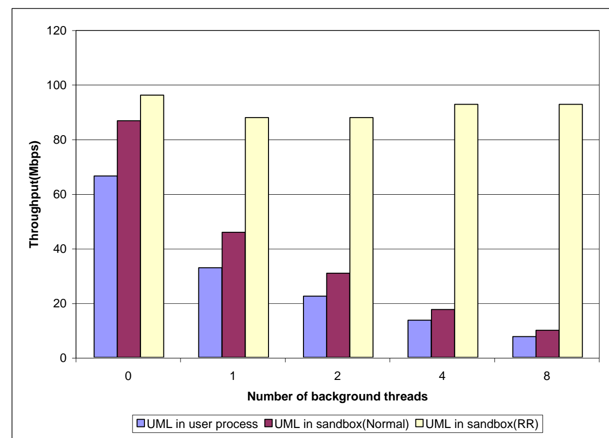

Figure 8. Throughput comparison of UML in a sandbox versus user-process using a synchronous processing model with TCP.

Summary: In summary, sandbox-based communications yields improved throughput and efficiency compared to traditional user-level approaches. The flexibility of our sandboxing system allows users to customize and configure network protocols and services for their needs without having to embed code in the kernel. For the most part, sandbox threads execute with the same low costs associated with kernel threads, in that they never require heavyweight address space switches. Additionally, by exposing the networking interface to a user-level sandbox, we are able to avoid unnecessary data copying. This is the same as if we executed code inside the kernel, because there would be no need to perform additional copies to user-space. Collectively, the elimination of unnecessary data copying and heavyweight address space switching results in improved service as seen by the throughput results in this section. We envision our approach as laying the foundations for a method of imple-

menting first-class user-level services that are tailored to the needs of specific applications. By first-class, we mean any service that has the same capabilities as traditional kernel services, with the exception that the kernel may always revoke access rights to any service abusing its privileges.

5. Related Work

Many researchers have explored various methods for high performance communication. For example, Active Messages [26] can be used to implement an efficient RPC [5] mechanism by running as handlers in the context of the currently active protection domain. This avoids scheduling and context switching overheads to implement network services, as does our approach using user-level sandboxing.

In effect, our work is similar to that of U-Net [25], Ethernet Message Passing (EMP) [19], and the Virtual Interface Architecture (VIA) [10], that all provide abstractions for user-level network implementation. In these alternative approaches to ours, the network interface card (NIC) is virtualized and multiplexed across applications. Additionally, if hardware permits, zero-copy capabilities are available. EMP requires programmable NIC interfaces to offload message processing to hardware and provide zero-copy capability. While U-Net and VIA are also able to take advantage of advanced hardware, they can only run on non-programmable NIC cards at the cost of efficiency. In contrast, our work allows user-level extensions to run efficiently enough to be invoked as handlers for networking events, without the need for special hardware support.

There have been a number of related research efforts that focus on OS structure and extensibility, safety, and service invocation. Extensible operating systems research [20] aims to provide applications with greater control over the management of their resources. SPIN [4], for example, is an operating system that supports extensions written in the type safe Module-3 programming language. By using type safety and interface contracts to provide protection, extensions can be injected into the operating system and run at kernel level. In addition to being able to extend kernel functionality, like memory management and scheduling, they show their approach provides improved network latency and lower CPU utilization over user-level implementations of protocol forwarders and video servers. Our work attempts to bridge the performance gap between user and kernel-level network implementations evident in the SPIN experiments.

A transaction-based approach to system extensibility is employed by the VINO [18] operating system. Unsafe kernel extensions (or grafts) may be aborted, to allow the system to return to consistent state. We are currently working to provide CPU constrained execution for extensions running in the sandbox, using techniques found in the VINO and SafeX [28] work. Such a method would allow us to avoid the situation where an extension executing as a bottom half uses more than a constrained amount of resources.

In contrast to safe kernel extensions, micro-kernels such as Mach [1], and also exokernels [11, 14] offer a few basic abstractions, while moving the implementation of more complex services and policies into application-level components. Separating kernel and user-level services can introduce inter-process (or protection domain) communication overheads. In particular, this has caused micro-kernels to fall out of favor despite substantial reductions [15] in communication costs. We alleviate communication costs arising from scheduling and context-switching, by supporting the execution of service extensions in arbitrary address spaces.

Finally, observe that our work differs from user-level resource-constrained sandboxing [8], by Chang et al. That work focuses on predictable resource management by instrumenting applications to intercept resource requests. The emphasis of our work is to develop an efficient execution environment at user-level for kernel extensions and system services, regardless of which address space is active at the time extension code is invoked.

6. Conclusions and Future Work

This paper describes an efficient end-host architecture for the deployment of application-specific services, suitable for high performance distributed computing. Specifically, we present the implementation of an efficient user-level network stack using our sandboxing mechanism. The resultant approach provides higher levels of throughput and lower levels of jitter than those of traditional middleware services implemented in process-private address spaces. In many cases, our architecture enables user-level services to outperform equivalent kernel-based services that require scheduling. Throughput gains are most significant with our approach when the end-host is not capable of saturating the link bandwidth. However, even in cases where throughput increases are not significant with our approach, the elimination of scheduling overheads (particularly due to the scheduling granularity of a timeslice in the millisecond range) results in lower jitter between service invocations. Furthermore, when end-hosts have relatively low-bandwidth network links (e.g., using DSL or cable modem connections to the Internet), it may be possible for a simple middleware service to saturate the link, but our approach still minimizes CPU utilization, due to the elimination of unnecessary context-switching overheads.

The user-level sandboxing scheme allows networking extension code to safely and efficiently access and influence lower-level abstractions such as bottom halves and the network hardware. It is through these exposed abstractions that such high performance is available. The sand-

boxing mechanism itself allows for customizable network processing units or stacks to execute with user-level permissions. These extensions, therefore, cannot compromise the integrity of the kernel's address space and are well encapsulated. Moreover, this efficiency and safety are provided without cumbersome hardware requirements. The only requisite is that of a paging memory management unit which is ubiquitous in modern computers. It follows that wide deployment across a heterogeneous environment such as the Internet is possible and desirable.

While a sandboxed service is isolated from the kernel, it must either be written by a trusted source or in a manner that is memory-safe. This is because sandbox services could otherwise access one another's memory space, or that of a process-private address space. Although out of the scope of this paper, we are currently studying various techniques including type-safe languages [9] and software-based fault isolation [27] for multiple services *within* the sandbox memory region. Future developments to our architecture will enable the kernel to revoke access rights to any service abusing its resource capabilities, including that of CPU usage. This will prevent a malicious service from consuming all available resources. Finally, using binary-rewriting techniques, we intend to eliminate modifications to the host kernel, thereby increasing the portability of our user-level sandbox approach.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *the Summer USENIX Conference*, pages 93–112, Atlanta, GA, USA, July 1986.

[2] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[3] Beep:. http://www.beepcore.org.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[6] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigbit-per-second local area network. In *IEEE Micro*, pages pages 29–36, 1995.

[7] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *the Summer USENIX Conf.*, 1994.

[8] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, 2000.

[9] Cyclone: http://www.research.att.com/projects/cyclone/.

[10] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. In *IEEE Micro*, 1998.

[11] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[12] A. Forum. *ATM User-Network Interface Specification Version 3.0*. Prentice Hall, Englewood Cliffs New Jersey, 1993.

[13] I. Foster and C. Kesselman. Globus: A toolkit-based architecture. *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278, 1999.

[14] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, 2002.

[15] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. Ther performance of μ-kernel-based systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*. ACM, October 1997.

[16] Iperf version 1.7.0: http://dast.nlanr.net/projects/iperf/.

[17] J. Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2001.

[18] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, 1996.

[19] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *The Intl. Conf. for High Perf. Computing and Communications, SC2001*, 2001.

[20] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *the USENIX Annual Technical Conference*, pages 41–54, 1996.

[21] Squid web proxy cache: http://www.squid-cache.org/.

[22] G.E.. Alliance. IEEE 802.3z. The emerging Gigabit Ethernet standard., 1997.

[23] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[24] User-mode linux: http://user-mode-linux.sourceforge.net/.

[25] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symp. on Op. Sys. Principles*, pages 40–53, December 1995.

[26] T. von Eicken, D. E. Culler, S. C. Golsdtein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. Report UCB/CSD 92/675, University of California, Berkeley, March 1992.

[27] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Software-based fault isolation. In *Proceedings of the 14th Symp. on Op. Sys. Principles*, Asheville, NC, USA, December 1993.

[28] R. West and J. Gloudon. 'QoS safe' kernel extensions for real-time resource management. In *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.

[29] R. West and J. Gloudon. User-level sandboxing: a safe and efficient mechanism for extensibility. Technical Report 2003-014, Boston University, June 2003. http://www.cs.bu.edu/techreports/pdf/2004-009-endhost-architecture.pdf.