XenSocket: A High-Throughput Interdomain Transport for Virtual Machines

Xiaolan Zhang[1], Suzanne McIntosh[1], Pankaj Rohatgi[1], and John Linwood Griffin[2]

[1] IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
{cxzhang,skranjac,rohatgi}@us.ibm.com
[2] Arlington, Virginia, USA

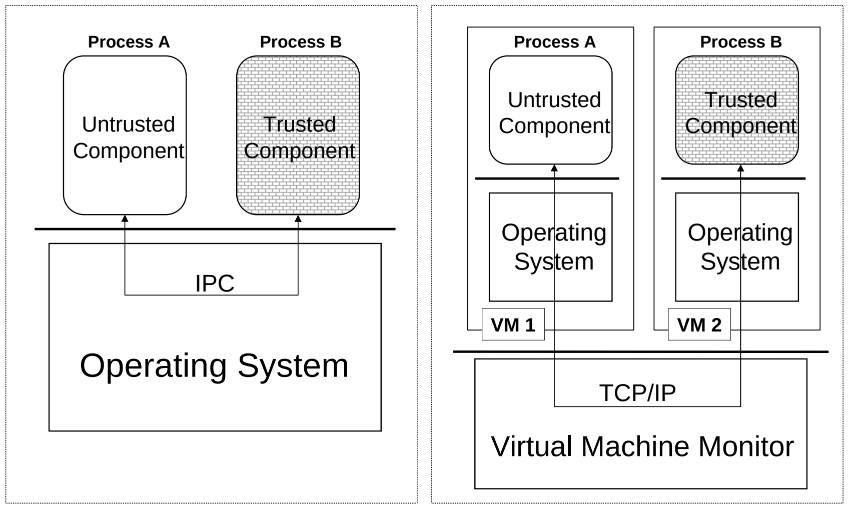
Abstract. This paper presents the design and implementation of XenSocket, a UNIX-domain-socket-like construct for high-throughput interdomain (VM-to-VM) communication on the same system. The design of XenSocket replaces the Xen page-flipping mechanism with a static circular memory buffer shared between two domains, wherein information is written by one domain and read asynchronously by the other domain. XenSocket draws on best-practice work in this field and avoids incurring the overhead of multiple hypercalls and memory page table updates by aggregating what were previously multiple operations on multiple network packets into one or more large operations on the shared buffer. While the reference implementation (and name) of XenSocket is written against the Xen virtual machine monitor, the principle behind XenSocket applies broadly across the field of virtual machines.

Key words: shared-memory IPC, interdomain communication, virtual machine, stream processing, security architectures, Xen

1 Introduction

Virtual machine technologies offer a number of benefits in the design of middleware. These include the ability to make more efficient use of hardware resources and to minimize network overhead by colocating multiple parties acting on the same data on the same physical machine. In addition, virtualization can provide increased robustness and security by isolating different applications and critical system components into separate protection domains within the same physical system. Finally, virtual machine technologies facilitate efficient monitoring and resource control of these different protection domains or partitions to ensure that adequate resources are available to critical domains. Figure 1 illustrates, at a conceptual level, how security can be improved by employing virtualization.

Unfortunately, the disappointing I/O performance of virtual machines has limited their adoption in application domains that require data-intensive, high-throughput network computing. Even with the recent advances in virtualization technology, virtual network and interdomain communication performance remain a problem. Taking the Xen [3] version 3.0.2 virtual machine monitor as an



(a) Monolithic Approach (b) Virtualization Approach

Fig. 1. Improved Security via Virtualization. In (a), the untrusted component might compromise the Operating System on which it is running which in turn leads to compromise of the trusted component running on the same OS. By isolating the trusted and untrusted components into separate Virtual Machines in (b), it is significantly more difficult for the untrusted component to affect the integrity of the trusted component. We are assuming that, because the Virtual Machine Monitor (VMM) is much smaller compared to a modern monolithic kernel, it is therefore much harder to break.

example, Figure 2 shows the transport throughput of two guest domains on the same machine communicating through a TCP connection. For comparison, the figure also shows the throughput of two Unix processes communicating through a UNIX domain socket stream on a native Linux system. As shown in the figure, the disparity is enormous, 13952 Mb/s for a UNIX domain socket vs. a mere 130 Mb/s for a TCP socket.

Analyses of the literature, combined with our own empirical observations, led us to speculate that a large source of overhead in Xen's interdomain networking was caused by the overhead of the TCP/IP stack as well as the repeated issuance of hypercalls to invoke Xen's page flipping mechanism. As described by Barham et al. [3], the Xen virtual machine monitor supports an atomic operation that updates the page tables in two virtual machines to swap the mapping of a pair of pages between the domains. This operation is used to implement a zero-copy network transmission from one domain to another: data in a network packet is page-aligned in one domain, the operation is invoked, and the data is now resident in the other domain. While this is a useful general solution for low-bandwidth messaging between domains, we speculated that it led to low throughput and high processor overhead for inter-domain communication bound applications.

To address these problems we designed and built XenSocket—a specialized interdomain transport based on memory buffers that are shared statically between a pair of domains. Applications inside a domain access this shared memory segment using the standard POSIX socket API, from which we derived the name XenSocket. A XenSocket is conceptually similar to a UNIX domain socket as would be provided by an operating system for interprocess communication; we note that we cannot simply use UNIX domain sockets for same-system component-to-component communication due to the need for virtualization-based isolation as described above.

The idea of using shared memory buffers for interprocess communication is obviously not new. However, one critical design issue for the virtualized environment is that information leakage is a sincere concern in a scheme that involves the direct sharing of memory resources between two dissimilar virtual machines (such as Domain-0 and an unprivileged domain). Special care must be taken to ensure that the integrity of the interdomain protections are maintained after the shared memory channel is torn down or after a workload is complete.

Another contribution of this paper is our exposition of one of several potentially useful techniques for high-throughput interdomain messaging in a virtual machine environment. Our sockets-based interface to the shared-memory-based transport provides a straightforward integration mechanism for large applications that require a mix of intra-machine and inter-machine communications.

We have realized an implementation of XenSocket against the Xen 3.0.2 release. Beyond Xen, the technique of using shared memory buffers for high-throughput communications applies generally across the field of virtual machines as well as other low-level resource protection schemes such as microkernels.

We summarize the key contributions of our paper below:

1. We designed and implemented an interdomain transport on Xen using shared memory. Our approach requires no modification to Xen or the Operating System.
2. Our design takes special care to maintain the interdomain protection provided by the original security architecture.
3. We measured the performance of our implementation and compared with previous approaches.
4. We demonstrated that security can be achieved with marginal performance loss—we were able to achieve throughput close to that of a native Unix Domain Sockets with much better security and robustness guarantees than we otherwise could with a monolithic kernel approach.

The remainder of this paper is organized as follows. Section 2 presents an example of a complex middleware application and details the existing performance problems we encountered with Xen. Section 3 describes our high-level design objectives with XenSocket. Section 4 discusses details of our Xen-based

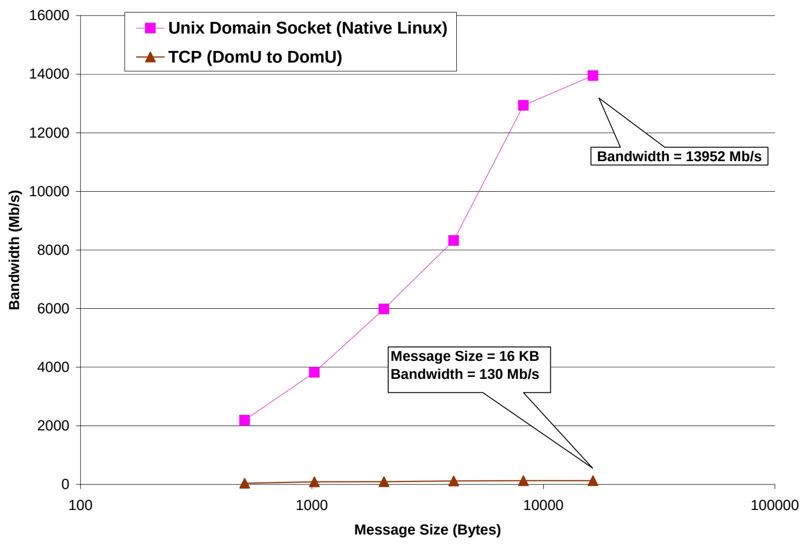


Fig. 2. Performance Comparison of TCP vs. Unix Domain Sockets as a Function of Message Size.

implementation. Section 5 presents the performance of our reference implementation. Section 6 describes related work. Section 7 discusses current status, open issues and future work surrounding XenSocket. Section 8 presents our concluding remarks.

2 Background and Motivation

2.1 A Motivating Example: System S

Outside the context of this paper, we are involved in building the security architecture for the large-scale distributed stream processing system known as System S being developed at IBM Research. The goal of System S is to extract important information by analyzing voluminous amounts of unstructured and mostly irrelevant data. Example applications for System S include analyzing audio, video and data feeds carrying information about financial, business and current events in order to support trading activities in financial institutions, and supporting responses to disasters such as Hurricane Katrina, based on analysis of vehicular movements, traffic and other sensors, news reports etc. System S has been designed to simultaneously address a number of challenges including

- **Rapid Reconfiguration**: The system must be quick to adjust to external events and the changing requirements and priorities of its users to the rapidly evolving data forms and types.
- **Perpetual Overload**: The system is required to "process" orders of magnitude higher data rates than existing systems, so a design goal has been to ensure that it functions well at high load. In fact, the system is designed to operate under a perpetual state of overload and must adjust its resource allocations to support the highest priority activities. This means that there will not be enough processing resources to completely analyze all the data being ingested, nor the bandwidth to transmit all the intermediate results, nor the storage to store all the data, so applications have to be designed to be resilient to variations in processing resources and to operate despite missing data.
- **System Security and Information Confidentiality** The system must be resilient against compromise from data-driven attacks originating from the ingested data and must adequately protect the confidential information being processed within it from unauthorized disclosure.
- **Heterogeneity** System S has to be designed to be a distributed system running on a heterogeneous collection of platforms, each specialized for particular types of processing.

The Stream Processing Core (SPC) is the middleware component of System S that hosts the distributed stream processing applications over heterogeneous hardware platforms and manages the stream connections, resources and dataflow autonomically. From a logical perspective, applications running on System S consist of multiple software-based processing and analysis components known as Processing Elements (PEs) which can communicate with each other via a unidirectional data stream abstraction. Each application can therefore be viewed as a directed graph with the PEs as nodes and the streams as edges, and at any point in time multiple applications could be running concurrently within System S. The SPC is responsible for providing both the execution environment for the PEs running in the system as well as the underlying data transport mechanism that implements the streams abstraction. The actual PE's are processes that are scheduled throughout a large physical installation and communicate with the rest of the system via the abstractions provided by a Streams Library (SL) that is linked into the PE executable. This library is also responsible for providing the streams API to the PEs, with the actual data transport across PEs managed by a separate data routing and transport component known as the Data Fabric. Each PE takes in chunks of data (known as stream data objects or SDOs) from one or more incoming streams, operates atomically and collectively on the input SDOs, and passes out results in the form of SDOs into one or more outgoing streams. These output SDOs are then transferred by the PE's streams library to the Data Fabric, which is then responsible for transporting them to the Streams Library of the subsequent PEs that need to consume them.[3]

[3] The interested reader will find more information on the scale and scope of the System S components in the treatment by Amini et al. [1] and on the Web; only those

2.2 Security Requirement of System S and Virtualization

Virtualization technology is an important component of the System S architecture. Although virtualization is not yet pervasively used throughout the system, it is critically needed in select places for the purposes of aggregation, colocation and most importantly for security and robustness. Given the large attack surface of applications analyzing large quantities of unstructured data of all types, it is highly likely that application PEs and even the operating systems hosting them could be compromised by exploits within the ingested data. The security architecture for System S (see Section 6 in [4]) therefore requires that application PEs that are not robust enough to handle low-integrity external data be confined and restricted in the way they can interact with other PEs as well as the rest of the system. This could be done by exploiting the protections provided by a virtualization layer; for example PEs operating on streams with different security or privacy labels could be isolated from each other and the virtualization protection could also be used to ensure the integrity of System S itself, in that trusted portions of the system (such as the Data Fabric modules that route SDOs between PEs) exist either on standalone physical machines or in isolated virtual machines, so as to be protected from damage from poorly designed or compromised PEs. In particular, the security architecture calls for the PEs and the Data Fabric to be resident in separate partitions or nodes, whereas in the current implementation these reside on the same node and the Streams Library transports the SDOs back and forth from the Data Fabric using Unix Domain Sockets.

2.3 Performance Requirement of System S

The fundamental performance bottleneck of System S is designed to be the saturation of the network links between each processing component [7]. In other words, throughput—the number of chunks per second passing through the system (or between components)—is a key metric of goodness for our purposes.[4]In contrast, the latency incurred by chunks moving from component to component does not have an important impact on the overall performance of the Stream Processing Core, and the fraction of processing time consumed for each chunk is expected to be negligible in comparison with the fraction of the network capacity consumed by each SDO.

2.4 Problem Statement

Unfortunately, our empirical experience with Xen as a virtualization platform for System S showed that interdomain communication using the Xen virtual

details that motivate the design of our high-throughput messaging system are included in this paper.

[4] Actually, the metric of goodness in System S is the utility of the work done within the system [2], but networking throughput remains a key bottleneck.

network fell well short of the throughput metrics identified for the project. In addition, the processor overhead consumed by the virtual network infrastructure was substantial enough to take away resources needed by each SDO. Specifically, domain-to-domain throughput capped out at around 130 Mb/s to 142 Mb/s (13-14% of the available raw physical network capacity), while maxing out the CPU utilization of the guest domains and requiring 18-20% of the processor in Domain-0.

The inefficiency of virtual machines as regards same-system networking performance is a well-known problem. In the literature, Menon et al. use profiling to explain some inefficiencies in the Xen virtual network [13], and in follow-up work Menon, Cox, and Zwaenepoel discuss performance optimizations to the network stack that leverage the fact that two domains on the same system are not constrained by physical network effects such as small packet sizes and the need to calculate and verify checksums [12]. Through these optimizations the authors achieve a maximum receive throughput of 970 Mb/s and transmit throughput of 3310 Mb/s. While these improvements are noteworthy, the performance of the resulting system still falls short compared to that of Unix Domain Sockets (over 10,000 Mb/s, see Figure 2).

In order to support the use of virtualization in our distributed stream processing application, our objective is to achieve throughput performance on both the send and receive paths at speeds approaching those of a UNIX domain socket for mid-sized messages (tens or hundreds of kilobytes).

3 Design

XenSocket provides a sockets-based interface to one or more large shared memory buffers for domain-to-domain communication. We make the design assumption that a XenSocket provides a one-way tunnel between a sender domain and a receiver domain. As discussed in Section 7, this assumption is not a requirement of a shared-memory-based transport; rather, the choice was made to conserve memory in the event that only one-way communications are needed.

Our shared-memory-based system is especially appropriate for asymmetric broadcast communications, where one domain sends a lot of information to multiple other domains on the same system (perhaps including an I/O domain for retransmission to other physical machines) without expecting to receive anything in return other than an acknowledgment of receipt.

3.1 Shared Memory and Circular Buffers

XenSocket was designed to test our hypothesis that per-packet page flipping is a large source of inefficiency in the Xen virtual network design. This was inspired in large part by the work of Menon, Cox, and Zwaenepoel [12] who demonstrated substantial performance gains in the Xen virtual network by transmitting more information per hypercall and, notably, replacing some instances of page flipping between domains with a memory copy between the domains.

XenSocket uses shared memory for message passing. There are two types of memory pages shared by each endpoint of a XenSocket: a descriptor page and buffer pages. The descriptor page is used to store control information. The buffer pages together form the circular buffer. When a socket connection is established between two domains, a shared memory region is reserved by one domain and mapped by the other domain. This shared memory is treated as a circular buffer: the sender writes data into this buffer, and the receiver reads directly from the buffer in FIFO order. This design differs from the well-utilized method of using Xen communication rings and page flipping, where data are first placed onto a memory page by the sender and then that page is remapped into the receiver's address space.

3.2 Sharing and Security

When sharing pages between domains of different trust levels (say, between an unprivileged domain and Domain-0), it is important that pages are only shared from the less-trusted domain and only mapped by the more-trusted domain. This design is to prevent the less trusted domain from launching a Denial of Service (DoS) attack on the more trusted domain by repeatedly establishing XenSocket connections to the more trusted domain without tearing them down, eventually exhausting the resources of the more trusted domain. We refer to the less-trusted domain as the *server* domain, since it provides the pages used for the circular buffer. The more-trusted domain is then the *client* domain, since it maps these pages into its own memory space. Note that the label of server and client is independent of which domain acts as the sender of data and which as the receiver.

It is currently necessary for the designer of a shared-memory-based transport to consider this, as Xen does not support the forced revocation by the hypervisor of a mapped page by a domain. This security design must be enforced by the existing security architecture of the application using an explicit policy, as Xen by default does not assign trust labels to domains (other than the hardwired fact that Domain-0 is more trusted than any other domain). One way of implementing this is through sHype, the secure hypervisor architecture for Xen [16].

Each XenSocket uses one descriptor page per one-way connection. The descriptor page is mapped read-write by both domains and is used for transmission of control information passing between the domains. An alternate design is possible where each domain provides its own descriptor page that is readable by both domains but only writable by itself. However, such a split design is both undesirable (from the standpoint of only wanting less-trusted domains to expose pages) and unnecessary, as long as none of the domains use the information in an unsafe, unchecked manner. In other words, if there is operationally-sensitive information that domain S shares with domain R, then S should keep the authoritative copy of the information in memory that is not shared or visible to R. In this way, if R overwrites the copy of this information on the shared page, it will not affect the correct operation of S—R's overwrites will only impact R itself.

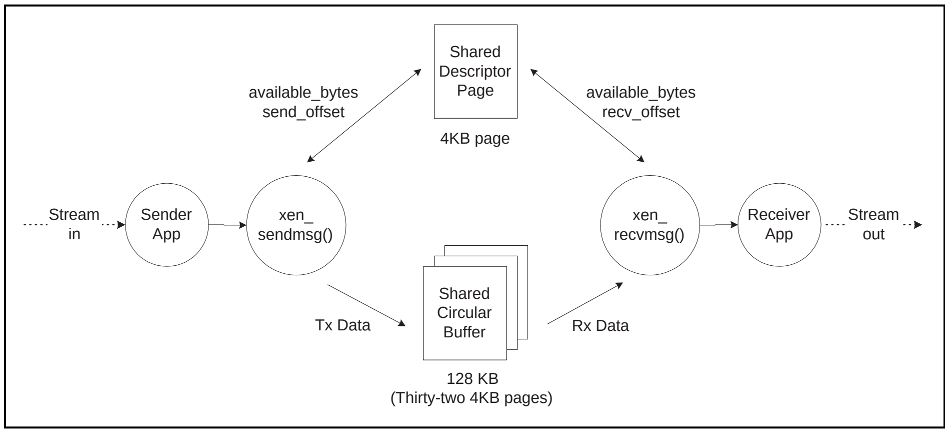


Fig. 3. XenSocket Architecture

3.3 A Sockets Interface to Shared Memory

In XenSocket, a sender application in one domain can create a socket (just as it would create a socket for TCP/IP-based communication or Unix Domain Sockets) and use `send()` (or `write()`) to push data into the socket. A receiver application in another domain can also create a socket and use `recv()` (or `read()`) to pull data from the socket. The choice of a sockets-based interface was initially made because existing System S components such as the Stream Processing Core (SPC) already make use of a sockets interface in their communications, but we believe that a sockets-based interface is a generally useful one to support the migration of distributed applications into a virtual machine environment.

In our design, a domain does not specify whether it will be a sender or a receiver on a XenSocket; this choice is indicated the first time the domain issues a `write()` or `read()` operation on the socket—after issuing a command of one type (e.g., `write`), any commands of the opposite type (`read`) will immediately return with a failure code.

4 Implementation

XenSocket is a socket-based solution for increasing interdomain throughput in Xen. Its APIs follow from standard socket APIs. Underneath this socket API, XenSocket uses shared memory for implementing high-throughput, interdomain data transfer. Our implementation is based on Xen version 3.0.2. XenSocket compiles into a kernel module and currently requires no changes to Xen or Linux. Work is in progress to port the implementation to newer versions of Xen.

XenSocket allocates two shared memory regions accessible by both the sender and receiver. One region consists of just one 4KB page for storage of state and control variables shared by the sender and receiver, which is called the descriptor

Server (Receiver)	**Client (Sender)**
`s = socket();`	`s = socket();`
`...`	`...`
`gref = bind(s, xaddr);`	`...`
`...`	`connect(s, xaddr);`
`recv(s);`	`send(s);`
`...`	`...`
`...`	`shutdown(s);`
`shutdown(s);`	

Fig. 4. XenSocket Usage Example

page. The second region is comprised of multiple 4KB buffer pages that form one shared circular buffer. In our present implementation, thirty-two pages are allocated to realize a 128 kilobyte circular buffer. Figure 3 shows the architecture of the XenSocket implementation.

4.1 User Perspective

From a user perspective, XenSocket has a simple sockets-based interface, so chosen because of its simplicity and because existing components in our application already communicated over a socket interface. In this section we show how an application would use the XenSocket API, highlighting the differences between XenSocket API and standard socket API.

Figure 4 illustrates the time sequence whereby XenSockets are established and used by a client (sender) and server (receiver) in a typical scenario [5]. The receiver in one domain first creates a socket by calling the `socket()` API. It then calls the `bind()` API to bind the socket to an address (`xaddr`). Additionally, it allocates physical memory to establish both a descriptor page and a shared circular buffer. Unlike the normal `bind()` call, which returns an error code indicating success or failure, the XenSocket `bind()` call returns the grant table reference to the descriptor page (`gref`) on success so that the sender can later use it to establish the sharing of that page. The `bind()` API also allocates an event channel to be used for communication with the sender whose identity, its domain number, is passed in as part of the socket address (`xaddr`) parameter. The receiver then calls `read()` or `recv()` for receiving data. The receiver blocks until it detects data in the circular buffer. The receiver calls `shutdown()` upon detecting that the sender has ended the connection.

The sender similarly calls `socket()` to create a socket just as the receiver does. The sender then calls `connect()`, supplying the receiver's domain ID and

[5] Note that although in this example, the server acts as a receiver and the client as a sender, the mapping between the server and the receiver (similarly the client and the sender) is not fixed, as discussed in Section 3.2.

```
Procedure: xen_sendmsg
Input     : target_bytes
Output    : written_bytes
begin
    num_bytes ← 0;
    written_bytes ← 0;
    while written_bytes < target_bytes do
        num_bytes ← atomic_read(available_bytes);
        num_bytes ← min(num_bytes, target_bytes);
        if num_bytes = 0 then
            wait with timeout;
            continue;
        end
        write num_bytes into circular buffer;
        send_offset ← (send_offset + num_bytes) mod BUFFER_SIZE;
        atomic_sub(available_bytes, num_bytes);
        signal receiver of newly available data;
        written_bytes ← written_bytes + num_bytes;
    end
    return written_bytes;
end
```

Algorithm 1: Send Algorithm. The use of atomic operations eliminates the need for conventional locks and thus improves performance.

the grant table reference of the shared descriptor page [6], both are part of the `xaddr` parameter. The `connect()` call gets the addresses of the physical pages of the shared circular buffer, which were placed in shared memory when the receiver called bind(), and maps them into the virtual address space of the sender. Additionally, it establishes the other end of the event channel facilitating communication of events between the client and server. With all this in place, the sender can now transmit data by calling `send()` or `write()` to deposit data into the circular buffer. The sender shuts down when all data has been sent.

4.2 Data Transfer

One core piece of the implementation is an efficient data transfer algorithm using atomic operations provided by the Linux kernel. A sketch of the send and receive algorithms is shown in Algorithm 1 and Algorithm 2 respectively. Pseudo procedures starting with `atomic` indicate atomic operations. The send and receive algorithms use one shared control variable, `available_bytes`, which indicates the number of bytes available for write in the circular buffer. Both the sender and the receiver maintain local read/write offsets into the circular buffer, which are not shared.

[6] In our current implementation, the grant table reference value is passed to the connect() call manually, however in the future we intend to automate this.

```
Procedure: xen_recvmsg
Input      : target_bytes
Output     : read_bytes
begin
    num_bytes ← 0;
    read_bytes ← 0;
    while read_bytes < target_bytes do
        num_bytes ← atomic_read(available_bytes);
        num_bytes ← min(num_bytes, target_bytes);
        if num_bytes = 0 then
            wait with timeout;
            continue;
        end
        read num_bytes from circular buffer;
        recv_offset ← (recv_offset + num_bytes) mod BUFFER_SIZE;
        atomic_add(available_bytes, num_bytes);
        signal sender of newly available space;
        read_bytes ← read_bytes + num_bytes;
    end
    return read_bytes;
end
```

Algorithm 2: Receive Algorithm.

Currently, our implementation supports only blocking reads and writes. When there is no room in the circular buffer for writing, `send()` will block. The sender will remain in a wait loop, awaking periodically, until space becomes available in the circular buffer. Similarly, `recv()` blocks when the buffer is empty. It remains in the blocking state until data is available for read. The sender signals the receiver of available data via the event channel when more data is written to the buffer. Similarly, the receiver signals the sender of available space when more data is consumed from the buffer.

4.3 Connection Teardown

Unlike Unix Domain Sockets, where either endpoint of the connection can shut down independent of the other, care must be taken to tear down a connection in XenSocket to ensure a smooth unmapping process because of the shared resources between the two endpoints. Since the server is the one that allocates the shared resources, our current implementation of XenSocket ensures that the client shut down first. If we had allowed the server to shut down first, the descriptor page, event channel, and circular buffer would all have been torn down, making communication between the client and server for the purpose of synchronization impossible. On the other hand, the shutdown API provides the user applications with the capability of initiating a shutdown at either endpoint. To support this, our shutdown implementation uses two shared control variables to serialize the shutdown. `shutdown()` first detects whether the client or the server is the caller. In the former case, the shutdown proceeds as usual, and one shared variable is set to indicate that the client has shutdown. The server application is notified of this condition after all data sent by the client has been emptied from the circular buffer. The server application can then issue a `shutdown` call, which properly deallocates all shared resources. If the server application issues a `shutdown()` call first, a second shared control variable is set to indicate that the server has initiated a shutdown and waits for the client to shut down first. When the client detects such a situation, it immediately stops sending data and returns an error code to the application, which in turn will eventually issue a `shutdown()` call. The shutdown process then proceeds as if the client had initiated the shutdown first.

Our XenSocket implementation is resistant against misbehaving server domains. We assume that the high integrity client domain is trustworthy and therefore can be relied upon to behave correctly. In our implementation, the client domain is non-blocking – it merely notifies the server domain that it has initiated a shutdown. Therefore, if the server misbehaves, it will only hurt itself.

5 Performance Evaluation

We evaluated our XenSocket implementation on an IBM HS20 blade with dual 2.8GHz Pentium Xeon processors and 4GB RAM. We use netperf version 2.4.2 as our primary benchmark. All data reported was run on Xen version 3.0.2 and Linux version 2.6.16.18. Each test was run 3 times, with the average reported. All experiments were run in single CPU mode with hyper-threading disabled to minimize performance variation.

5.1 Performance for Common Message Sizes

Figure 5 shows the reported throughput as a function of message size for XenSocket between two guest domains, as compared to that for Unix Domain Sockets of two processes on native Linux, and that for unmodified TCP between two DomUs. As demonstrated in the figure, XenSocket achieves up to 72 times the throughput of standard TCP stream in the peak case (message size = 16 KB). However, XenSocket still lags Unix Domain Sockets by 33% in this case. We are very encouraged by this initial performance result and are continuing to optimize XenSocket further.

The gradual increase of the throughput as the message size increases indicates that at small message sizes, the performance is dominated by the per-message call overhead (one system call plus one Xen Hypercall each side). When the message size increases, the performance becomes dominated by the overhead of actually transferring the data. At the message size of 16 KB, XenSocket reaches a peak throughput of 9295 Mb/s. At this rate, the CPU utilizations of both guest domains reach 100%, whereas Domain-0 remains at near zero CPU utilization.

A direct comparison with the results of Menon et al. [12] is not illustrative, as described below, but it is useful to point out the design choices that cause

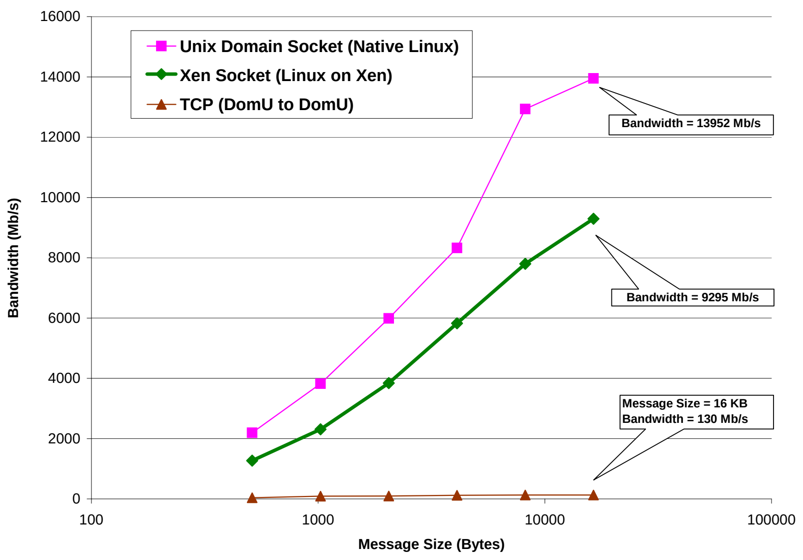


Fig. 5. Throughput Comparison of XenSocket vs. Unix Domain Sockets and TCP for Message Sizes Between 512 Bytes and 16 KB. XenSocket achieves up to 72 times the throughput of standard TCP stream at message size of 16 KB.

our results to differ. Their results are asymmetric, with a maximum receive performance of 970 Mb/s and a maximum transmit performance of 3310 Mb/s. In addition, there is a big difference between running the benchmark in the driver domain and in the guest domain. In our case, since we run both the receiver and the sender on the same machine, we only look at the maximum bandwidth that can be achieved between the two. Additionally, since XenSocket does not require Domain-0 to be involved in the data exchange, it does not make much difference whether the sender (or receiver) resides in the driver domain or the guest domain.

To make the comparison more complete, we also look at the CPU utilization of XenSocket at performance close to 3310 Mb/s, 970 Mb/s and 130 Mb/s, the maximum transmit and receive throughputs achieved in Menon et al., and in unmodified TCP on Xen (see Section 2). We modify netperf to sleep at a certain rate so as to bring down the performance to the specific target level. Since the throughput varies at each run, it is difficult to fix the throughput at exactly a static value. Thus, we chose the throughput level that is closest to the target level. The CPU utilization at the sender and receiver is taken from the statistics reported by netperf. For Domain-0, we use the percentage of total processor time spent idle reported by the `vmstat` tool. Table 1 lists the CPU utilization of the sender and receiver guest domains, and domain-0. At 3320 Mb/s, the CPU utilization is around 6% for the sender and 11% for the receiver. At 972 Mb/s,

Throughput	CPU Utilization		
	Sender	Receiver	Domain-0
3320 Mb/s	6%	11%	1%
972 Mb/s	3%	4%	0%
136 Mb/s	0%	2%	0%

Table 1. CPU Utilization vs. Achieved Throughput in XenSocket. As discussed in Section 2, the existing Xen virtual network requires 18-20% CPU usage in Domain-0 to transfer only 130-142 Mb/s between two guest domains.

the CPU utilization is around 3-4% for the sender and the receiver. At 136 Mb/s, the CPU utilization is close to 0% for the sender and about 2% for the receiver. In all cases, Domain-0 is mostly idle.

Note that this is not an exact apples to apples comparison for two reasons: First, we run the sender and the receiver on the same machine, whereas in the case of Menon et al., the sender and the receiver are evaluated separately. Secondly, we have different assumptions on the intended usages of XenSocket than that of Menon et al. Our intended applications are high-throughput distributed stream systems, thus we relax the latency requirement, and can do batching at the receiver side. In contrast, Menon et al. have to support interactive networking, and therefore have to dispatch any network packet received from the network immediately to the receiver.

Despite these differences, we believe that the comparison is still meaningful in that it highlights the unique features of our approach and shows how the differences in the two approaches affect performance.

5.2 Performance for Larger Message Sizes

Figure 6 shows the throughput of XenSocket, Unix Domain Sockets and TCP for large message sizes (ranging from 16 KB to 64 MB). It's interesting to note that for both XenSocket and Unix Domain Sockets, the throughput starts to drop off after a certain message size (16 KB for XenSocket and 64 KB for Unix Domain Sockets), then stabilizes when the message size is larger than 512 KB. Interestingly, XenSocket performs about 33% better than Unix Domain Socket (6534 Mb/s vs. 4907 Mb/s for message size of 2 MB). For TCP, the throughput is virtually unchanged at about 141 Mb/s.

We investigated the cause of the performance drop off for large message sizes using the OProfile tool [10] and its extension to Xen [13]. Our initial results indicate that there is a strong correlation between the throughput performance and the L2 cache hit ratio. We thus believe that the drop off is caused by some caching effects of the L2 cache. Another indication that this is due to cache effects comes from the fact that performance of the Unix Domain Socket varies across identically configured hardware platforms. A precise characterization of the performance variation and pinpointing the causes will require further research. It

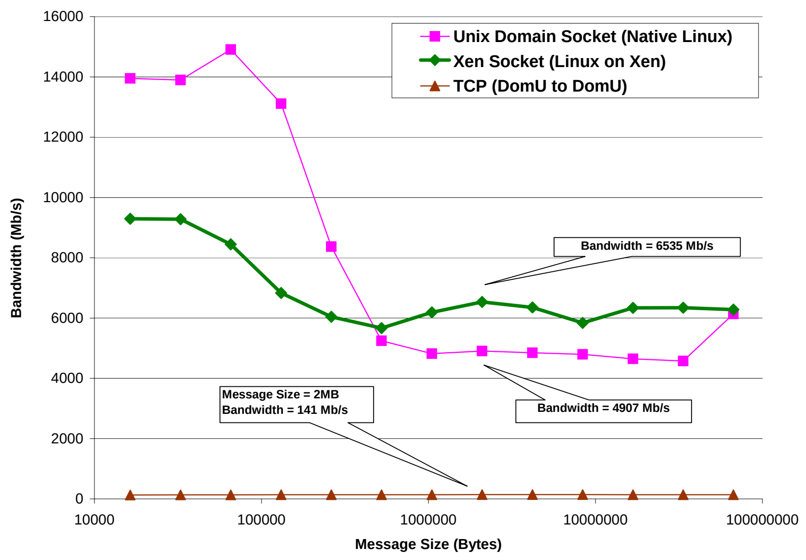


Fig. 6. Throughput Comparison of XenSocket vs. Unix Domain Sockets and TCP for Large Message Sizes. Both XenSocket and Unix Domain Sockets see a large drop-off when the message size reaches 512 KB and then stabilize around 5-6 Mb/s. The performance curves invert at message size of 512 KB where XenSocket outperforms Unix Domain Sockets.

suffices to say that the variation is comparable to the performance difference between XenSocket and Unix Domain Socket, confirming that XenSocket indeed achieves throughput close to that of Unix Domain Socket.

6 Related Work

Our approach is inspired by previous research on using shared memory buffers for interprocess communication. As an earlier example, we note the use of cached fast buffers by Druschel, Peterson, and Davie [5] in their optimization of the OSIRIS network adaptor. More recently, Götz implemented a shared-memory-based transport for high-throughput data transfer in the L4 microkernel [6]. There are also examples other than our work on System S that motivate high-throughput communication in a VM environment, such as the virtualization of a transaction processing system that contains multiple front-end web servers, interconnected database servers, and back-end storage system nodes.

The Xway project [8] also uses a sockets interface over a shared-memory transport to improve Xen interdomain communications throughput. The Xway and XenSocket projects were developed independently but share similar designs.

The core difference is the type of socket interface presented to the user or application. With Xway, applications create sockets using the existing AF_INET protocol family. Modified INET socket code creates a shared-memory transport whenever both endpoints are on the same physical host. The Xway design allows deployment of the shared-memory transport without requiring changes to existing applications. With XenSocket, sockets are created using a new AF_XEN protocol family. The XenSocket design enables communication between domains that do not have virtual network devices or that do not share a common Internet Protocol-based network interface. This level of isolation is important for System S security and in such architectures as that described by Payne [15].

The PROSE System prototype developed by Van Hensbergen and Gross [17] uses shared buffers for low-latency IPC in a hybrid microkernel-and-virtual-machine environment. Their work focuses on latency and no performance details are available for bandwidth benchmarks. In addition, their approach uses polling at the receiving side, which leads to more CPU usage than a non-polling algorithm.

Liu et al [11] looked at improving device I/O of Virtual Machines by leveraging the virtualization capabilities of the device itself and bypassing the VMM all together for performance critical operations. The idea was inspired by early work on OS-bypassing I/O where user-level applications can directly access physical devices in order to improve performance. While their approach shares similar principle with ours in that both try to improve performance by minimizing the involvement of the VMM, there are two fundamental differences between the two approaches. In our approach, the VMM is always involved in the communication (e.g., it's never bypassed). In addition, our approach does not involve physical devices. Rather, it only concerns the communication of two VMs on the same physical platform.

An orthogonal area of memory sharing research on virtual machines focuses on improving the *spatial* efficiency of memory usage. For example, Kloster et al [9] employs hashing to locate identical pages that belong to different VMs and transparently share the page among VMs, thereby reducing the total number of required physical pages. Because the pages are identical, and sharing is performed transparently from the VM's perspective, there is no security implication of this optimization, except for the possibility of opening up potential side channels.

7 Discussion

As described above, our design of a XenSocket is a one-way communications pipe between two domains. While the traditional view of a socket is a two-way mechanism, we chose the one-way design as a balance between our desire to minimize overall system impact and our interest in ensuring a large circular buffer to avoid stalling by the sender or receiver. A more complete design would include variable-size circular buffers whose logic is capable of adapting the buffer reservation size to the actual usage of the buffer. In this way a two-way socket could be the norm, where the initial circular buffer size is small but grows to most efficiently match the demand. A variant on this idea would be to dynamically move pages between the two circular buffers in order to adapt the buffer size to the workload while maintaining a constant amount of memory reservation per XenSocket.

An unexplored aspect of our design for XenSocket is its use in a local multicast environment; i.e., in the case where one domain sends identical messages to a constant set of multiple other domains on the same system. When one or more of these other domains act as an external network bridge, this could represent a multicast to applications running both in local domains and on remote systems. The descriptor page in our design could be extended to include acknowledgments from each of the receiving domains. This would reduce the memory and computational pressure on the sending domain—both in comparison with the design presented in this paper and with the original Xen virtual network—as the sending domain would only have to copy each message once into a shared memory buffer instead of performing work for each receiving domain. However, it remains to be explored the degree to which such an approach is open to denial-of-service attacks when one domain chooses not to acknowledge on a timely basis the data it receives, filling up the circular buffer and therefore halting the information flow.

There are other aspects of performance optimization that we have not yet explored. For example, we can offload control of memory transfers into and out of the shared memory space to the DMA controller or the I/O memory management units. Resource contention may become an issue for multiple instances of XenSocket running in parallel due to the extra memory copies needed into and out of the circular buffer. However, we note that even in the original Xen page-flipping scheme for virtual networks, it is still necessary to copy data into and out of the pages that are flipped. Another optimization is to implement the shared pages in the hypervisor memory, which is mapped to all VMs. An advantage of this approach is reducing the number of cache and TLB flushes due to context switches. A disadvantage is that it does not scale to large number of concurrent connections. A third optimization is using cooperative scheduling mechanisms such as gang scheduling [14], where the sender and the receiver are scheduled together to minimize waiting time.

A hardware trend that is relevant to our work on XenSocket is the emergence of multi-core processors. In a virtualized multi-core environment—where a currently-open question is "what are we going to do with all those cores?"—one class of applications that will map well to the environment contains those distributed applications that compute sequential analyses over large local data sets. Examples of these are image recognition or feature extraction applications. While such applications could be written as large multi-threaded programs with a common shared memory pool, we postulate that the preservation of isolation boundaries combined with a distributed message-passing paradigm will provide the most useful transition path for minimally-modified distributed software architectures into a multi-core environment.

8 Conclusion

As virtualization becomes more widely deployed, we foresee a growing number of applications that require high-performance interdomain communication. This is in part driven by security and reliability concerns—by separating components of a complex software system into different domains, one achieves better isolation among the components, thus improving security and reliability. For example, in our target application, a large-scale distributed stream processing system consisting of components with different trust levels, our security design mandates that components of different trust levels must be placed either on separate machines or on separate virtual machines, in both cases with a proper security-label-based gating of communication between the machines. Achieving this with only marginal degradation of communication performance in a virtual machine environment is particularly crucial for our target application whose success depends on the ability to transfer large amounts of data rapidly between the distributed application components.

In this paper we present XenSocket, a shared-memory-based construct that provides a POSIX sockets-based mechanism for high-throughput interdomain communications. XenSocket draws on best-practice work in this field and avoids incurring the overhead of multiple hypercalls and memory page table updates by aggregating what were previously multiple operations on multiple network packets into one or more large operations on the shared buffer. Our performance evaluation indicates that with XenSocket we have successfully achieved our goal of same-system interdomain transport throughput that approaches that of interprocess communication using UNIX domain sockets.

We have released the source code for our XenSocket reference implementation under the name **XVMSocket**. XVMSocket is freely available at the SourceForge open source software development web site [18] for use under the terms of the GNU General Public License.

9 Acknowledgments

We would like to thank our colleagues from IBM Research, Ronald Perez, Douglas Lee Schales, and Volkmar Uhlig, and our colleagues in the IBM Linux Technology Center, Anthony Liguori, Ryan Harper, Muli Ben-Yehuda, and Eric Van Hensbergen, and Jose Santos from Hewlett Packard for insightful discussions and comments. Reiner Sailer, Stefan Berger and Wesley Most helped in the initial Xen setup. And finally, we thank our anonymous reviewers for their careful review and helpful comments on improving this paper.

References

1. L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *DMSSP '06: Proceedings of ACM SIGKDD Workshop on Data Mining Standards, Services and Platforms*, Philadelphia, PA, USA, 2006.

2. L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS 2006*, 2006.
3. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
4. P. Cheng, P. Rohatgi, C. Keser, P. A. Karger, G. M. Wagner, and A. S. Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. Technical Report RC24190, IBM Research, Yorktown Heights, NY, USA, Feb. 2007.
5. P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM '94: Proceedings of the Conference on Communications Architectures, Protocols and Applications*, pages 2–13, New York, NY, USA, 1994. ACM Press.
6. S. Götz. Asynchronous communication using synchronous IPC primitives. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, May 2003.
7. N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 431–442, New York, NY, USA, 2006. ACM Press.
8. K.-H. Kim. Personal communication. May 1, 2007. http://lists.xensource.com/archives/html/xen-devel/2007-05/msg00122.html.
9. J. F. Kloster, J. Kristensen, and A. Mejlholm. Efficient memory sharing in the xen virtual machine monitor. Technical report, Aalborg University, Jan. 2006. https://services.cs.aau.dk/public/tools/library/files/rapbibfiles1/1136884892.pdf.
10. J. Levon and P. Elie. http://oprofile.sourceforge.net/about/.
11. J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *2006 USENIX Annual Technical Conference*, pages 29–42, Boston, Massachusetts, USA, June 2006.
12. A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *2006 USENIX Annual Technical Conference*, pages 15–28, Boston, Massachusetts, USA, June 2006.
13. A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE'05: First International Conference on Virtual Execution Environments*, pages 13–23, Chicago, Illinois, USA, June 2005.
14. J. K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS '82: 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.
15. B. D. Payne, R. Sailer, R. Cáceres, R. Perez, and W. Lee. A layered approach to simplified access control in virtualized systems. *Operating Systems Review*, 41(3):12–19, July 2007.
16. R. Sailer, T. Jaeger, E. Valdez, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. Technical Report RC23629, IBM Research, Yorktown Heights, NY, USA, June 2005.
17. E. Van Hensbergen and K. Goss. PROSE I/O. In *IWP9 2006: First International Conference on Plan 9*, Madrid, Spain, Dec. 2006.
18. XVMSocket. http://sourceforge.net/projects/xvmsocket/.