Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen [*]

Kangho Kim Cheiyol Kim Sung-In Jung
Electronics and Telecommunications Research Institute (ETRI)
{khk,gauri,sijung}@etri.re.kr

Hyun-Sup Shin
ETRI/UST
superstarsup@etri.re.kr

Jin-Soo Kim
Computer Science Dept., KAIST
jinsoo@cs.kaist.ac.kr

Abstract

Communication performance between two processes in their own domains on the same physical machine gets improved but it does not reach our expectation. This paper presents the design and implementation of high-performance inter-domain communication mechanism, called XWAY, that maintains binary compatibility for applications written in standard socket interface. As a result of our survey, we found that three overheads mainly contribute to the poor performance; those are TCP/IP processing cost in each domain, page flipping overhead, and long communication path between both sides of a socket. XWAY achieves high performance by bypassing TCP/IP stacks, avoiding page flipping overhead, and providing a direct, accelerated communication path between domains in the same machine. Moreover, we introduce the XWAY socket architecture to support full binary compatibility with as little effort as possible.

We implemented our design on Xen 3.0.3-0 with Linux kernel 2.6.16.29, and evaluated basic performance, the speed of file transfer, DBT-1 benchmark, and binary compatibility using binary image of real socket applications. In our tests, we have proved that XWAY realizes the high performance that is comparable to UNIX domain socket and ensures full binary compatibility. The basic performance of XWAY, measured with `netperf`, shows minimum latency of 15.6 μsec and peak bandwidth of 4.7Gbps, which is superior to that of native TCP socket. We have also examined whether several popular applications using TCP socket can be executed on XWAY with their own binary images. Those applications worked perfectly well.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance; D.4.7 [*Operating Systems*]: Distributed Systems

General Terms Performance, Experimentation

Keywords Virtual machine, socket interface, high performance, socket binary compatibility, Xen

[*] This work was supported by the IT R&D program of MIC/IITA. [2005-S-119-03, The development of open S/W fundamental technology]

1. Introduction

Virtual Machine (VM) technologies were first introduced in the 1960s, reached prominence in the early 1970s, and achieved commercial success with the IBM 370 mainframe series. VM technologies allow to co-exist different guest VMs in a physical machine, with each guest VM possibly running its own operating system. With the advent of low-cost minicomputers and personal computers, the need for virtualization declined [1, 3, 2].

As a growing number of IT managers are interested in improving the utilization of their computing resources through server consolidation, VM technologies are getting into the spotlight again recently. Server consolidation is an approach to reducing system management cost and increasing business flexibility by putting a lot of legacy applications scattered across network into a small number of reliable servers. It is recognized that VM technologies are a key enabler to achieve server consolidation.

When network-intensive applications, such as Internet servers, are consolidated in a physical machine using the VM technology, multiple VMs running on the same machine share the machine's network resource. In spite of the recent advance in the VM technology, virtual network performance remains a major challenge [10, 5]. Menon et al. [6] reported that Linux guest domain showed far lower network performance than native Linux, when an application in a VM communicates with another VM on a different machine. They showed the performance degradation by a factor of 2 to 3 for receive workloads, and a factor of 5 for transmit workloads [6].

Communication performance between two processes in their own VMs on the same physical machine (inter-domain communication) is even worse than we expected. Zhang et al. pointed out that the performance of inter-domain communication is only 130Mbps for a TCP socket, while the communication performance between two processes through a UNIX domain socket on a native Linux system is as high as 13952Mbps [10]. Our own measurement on Xen 3.0.3-0 shows the inter-domain communication performance of 120Mbps per TCP socket, which is very similar to Zhang's result. In the latest version of Xen 3.1, the inter-domain communication performance is enhanced to 1606Mbps (with copying mode), but still significantly lagging behind compared to the performance on native Linux. The poor performance of inter-domain communication is one of the factors that hinders IT managers from moving toward server consolidation.

We believe that TCP/IP processing, page flipping overhead, and long communication path between two communication end-points collectively contribute to performance degradation for inter-domain communication under Xen. It is not necessary to use TCP/IP for inter-domain communication as TCP/IP was originally developed to transfer data over unreliable WAN, not to mention that it requires non-negligible CPU cycles for protocol processing. The page flipping mechanism is useful to exchange page-sized data between VMs, but it causes a lot of overhead due to repeated calls to the hypervisor. In the current Xen network I/O model, inter-domain communication is not distinguished from others and every message should travel through the following modules in sequence: TCP stack/IP stack/front-end driver in the source domain, back-end driver/bridge in domain-0, and front-end driver/IP stack/TCP stack in the destination domain.

In this paper, we present the design and implementation of high-performance inter-domain communication mechanism, called XWAY, which bypasses TCP/IP processing, avoids page flipping overhead, and provides an accelerated communication path between VMs in the same physical machine. XWAY is implemented as a network module in the kernel. XWAY enables network programs running on VMs in a single machine to communicate with each other efficiently through the standard socket interface. XWAY not only provides lower latency and far higher bandwidth for inter-domain communication, but also offers full binary compatibility so that any network program based on the socket interface does not have to be recompiled to enjoy the performance of XWAY. We have tested several representative network applications including ssh, vsftp, proftp, apache, and mysql, and these programs work quite well without any recompilation or modification.

The rest of this paper is organized as follows. After a brief overview on Xen in Section 2, we discuss related work in Section 3. In section 4, we cover several design issues and solutions for achieving high-performance inter-domain communication while providing full binary compatibility with the existing socket interface. Section 5 describes the implementation detail of XWAY. We present experimental results in Section 6 and conclude the paper in Section 7.

2. Background

2.1 Xen

Xen is an open source hypervisor which is developed for x86/x64 platform and ported for IA64 and PPC. By using Xen, several operating systems can be run on a single machine at the same time. The hypervisor running between hardware and operating systems virtualizes all resources over the hardware and provides the virtualized resources to operating systems running on Xen. Each operating system is called guest domain and one privileged domain for hosting the application level management software is termed domain 0. Xen provides two ways of virtualization which are full virtualization and para-virtualization. On full virtualization, the hypervisor provides full abstraction of the underlying physical system to each guest domain. In contrast, para-virtualization presents each guest domain with an abstraction of the hardware that is similar but not identical to the underlying physical hardware. Figure 1(courtesy[**??**]) illustrates the overall architecture of Xen.

2.2 Xen network I/O architecture and interfaces

Domain 0 and guest domain have virtual network interfaces which are called as back-end interface and front-end interface, respectively. All network communications on Xen are carried out through these virtual network interfaces. Figure 2(courtesy [**??**]) illustrates the network I/O architecture in Xen. Each front-end network interface in the guest domain is connected to a corresponding backend network interface in the domain 0, which in turn is connected to the physical network interfaces through bridging.

There are two modes for front-end network interface to receive data from back-end network interface: copying mode and page flipping mode. On copying mode, data contained in the page of domain 0 are moved to a page in a guest domain through memory

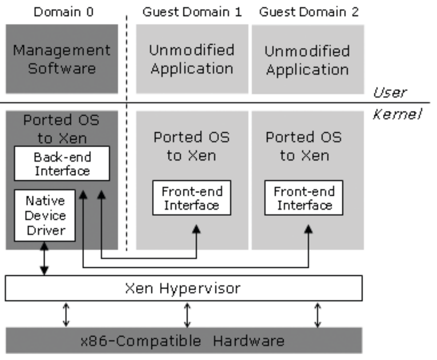


Figure 1. Overall architecure of Xen (courtesy [7])

copy operations. On page flipping mode, the page holding data in domain 0 is exchanged with an empty page that the guest domain provides.

Xen provides three means of communication among domains: event-channel, grant table, and Xenstore. Event-channel is a signaling mechanism for domains on Xen. A domain can deliver an asynchronous signal to another domain through event-channel. It is similar to hardware interrupt and includes one bit information. Each domain can receive a signal by registering event-channel handler.

Each domain needs to share the memory space on local memory area with another domain. However, no domain can directly access the memory area of another domain. Therefore, Xen provides a grant table to each domain. The grant table contains the references of granters and a grantee can access granter's memory by using the reference on the grant table.

Xenstore is a space to store or reference the information for setting up event channel and shared memory that is mentioned above as a grant table. It is organized as hierarchical structure of key value pairs and each domain has its own directory including data related to its configuration.

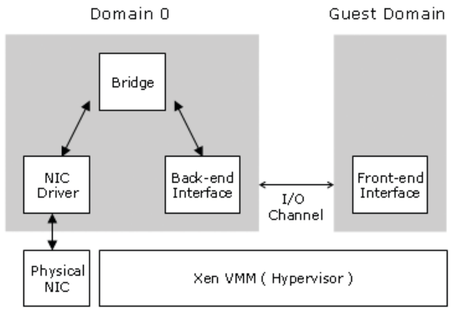
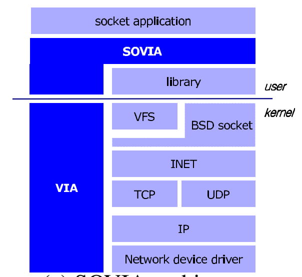


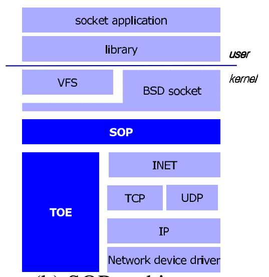
Figure 2. Xen network I/O architecture

3. Related Work

There have been several previous attempts to reduce the overhead of network I/O virtualization that affects the performance of network-intensive applications. One of the most general way is to redesign the virtual network interface. Menon et al. [5] redefined the virtual network interface and optimized the implementation of the data transfer path between guest and domain-0. The optimization avoids expensive data remapping operations on the transmission path, and replaces page remapping with data copying on the



(a) SOVIA architecture



(b) SOP architecture

Figure 3. Design alternatives for XWAY

receiving path. Although this approach can be also used for inter-domain communication, communication among VMs on the same physical machine still suffers from TCP/IP overhead and the intervention of domain-0.

Zhang et al. proposed XenSocket for high-throughput inter-domain communication on Xen [10]. XenSocket avoids TCP/IP overhead and domain-0 intervention by providing a socket-based interface to shared memory buffers between two domains. It is reported to achieve up to 72 times higher throughput than standard TCP stream in the peak case. Despite its high throughput, XenSocket only proved the potential ability of shared memory for high-throughput inter-domain communication. Its socket interface is quite different from the standard in that the current implementation not only requires the grant table reference value be passed to `connect()` manually, but also lacks such features as two-way data transmission and non-blocking mode reads & writes. Thus, XenSocket cannot be used as a general purpose socket interface.

Ensuring full binary compatibility with legacy socket-based applications is essential requirement for server consolidation, but it is not easy due to complex semantics of socket interface. The architecture of XWAY is largely influenced by our two previous attempts to support the standard socket interface on top of a high-speed interconnect, namely SOVIA and SOP.

SOVIA [4] is a user-level socket layer over Virtual Interface Architecture (VIA). The goal of SOVIA is to accelerate the performance of existing socket-based applications over user-level communication architecture. Since SOVIA is implemented as a user-level library, only source code-level compatibility is ensured and the target application should be recompiled to benefit from SOVIA. In some cases, the application needs to be slightly modified as it is hard to support the complete semantics of the standard socket interface at user level. Figure 3(a) illustrates the overall architecture of SOVIA.

Many TOEs [8] do not support binary compatibility with existing socket applications, even though they provide their own socket interface. SOP [9] presented a general framework over TOE to maintain binary compatibility with legacy network applications using the standard socket interface. SOP is implemented in the Linux kernel and intercepts all socket-related system calls to forward them to either TOE cards or legacy protocol stack. As illustrated in Figure 3(b), the intercept takes place at between BSD socket layer and INET socket layer. Note that it is necessary to intercept file operations as well as socket operations in order to achieve full binary compatibility, which requires a lot of implementation efforts.

4. Design Issues

In this section, we discuss several design issues that arise in the development of XWAY. Our main design goal is to make XWAY as efficient as UNIX domain socket on native Linux environment, while providing complete binary compatibility with legacy applications using the standard socket interface. XWAY should allow any socket-based applications to run without recompilation or modification to the application.

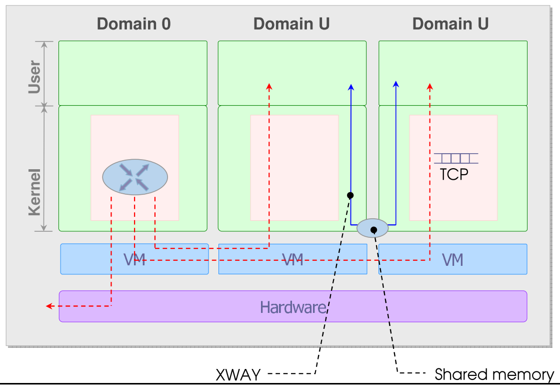


Figure 4. Inter-domain communication path for high performance

4.1 High performance

We focus on TCP socket communication between two domains in the same physical machine. It is expected that such inter-domain communication achieves higher bandwidth and lower latency than communication between different machines, as IPC and Unix domain socket significantly outperform TCP socket communication. Unfortunately, Xen falls far short of our expectations. As mentioned in Section 1, three notable overheads contribute to the poor inter-domain performance: (1) TCP/IP processing overhead, (2) page flipping overhead, and (3) long communication path in the same machine.

For high-performance inter-domain communication, XWAY bypasses TCP/IP stacks, avoids page flipping overhead with the use of shared memory, and provides a direct, accelerated communication path between VMs in the same machine, as shown in Figure 4. In Figure 4, the dotted arrows represent the current communication path, whereas the solid arrows indicate the simplified XWAY path through which both data and control information are exchanged from one domain to another in a single machine.

It is apparent that we can bypass TCP/IP protocol stacks for inter-domain communication as messages can be exchanged via the main memory inside a machine. This not only minimizes the latency, but also saves CPU cycles.

Along with the overhead of TCP/IP stacks, Zhang et al. attributed the low performance of Xen's inter-domain communication to the repeated issuance of hypercalls to invoke Xen's page flipping mechanism [10]. The page flipping mechanism enables a hypervisor to send page-sized data from one domain to another with only one memory pointer exchange. However, it leads to lower performance and high CPU overhead due to excessive hypercalls to remap and swap pages. Page tables and TLBs also need to be flushed since the mappings from virtual to physical memory addresses have changed. This is why the latest Xen 3.1 recommends "copying mode" over "path flipping mode" for receive data path, even though both modes are available. Instead, a simple shared memory can be used for transmitting data between domains, which does not require any hypervisor call.

The direct communication path provided by XWAY makes the latency shorter by eliminating the intervention of domain-0. This is a simple but efficient way to improve performance. In the current Xen design, two virtual interrupts and two domain switchings are required to send data from a socket application s_1 in a domain d_1 to a peer socket application s_2 in another domain d_2. When s_1 invokes `send()`, data in s_1's user space are copied to the kernel

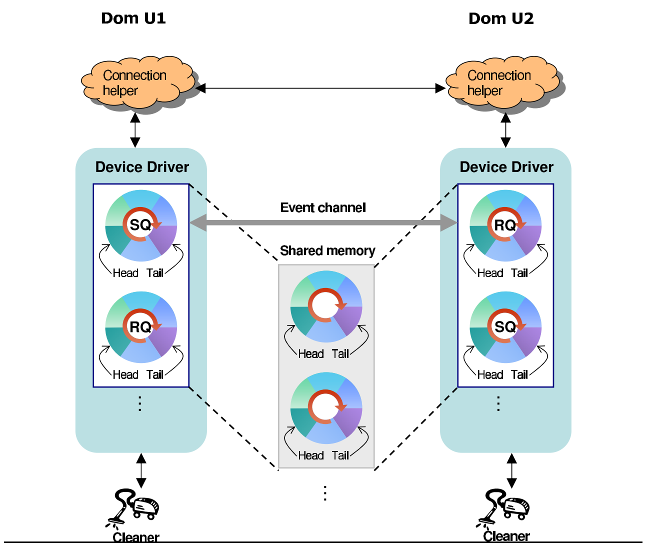


Figure 5. An XWAY channel

space in d_1 and the front-end driver in d_1 raises an event toward domain-0. The event triggers the back-end driver in domain-0 and the back-end driver fetches the page filled with s_1's data through page flipping. The bridge module determines the domain (d_2 in this case) where the data are destined to go. The rest of the steps from domain-0 to d_2 follow the same steps from d_1 to domain-0 except that the order is reversed. The direct communication from d_1 to d_2 cuts the communication cost nearly in half by removing one virtual interrupt and one domain switching.

Let us consider the same scenario under XWAY. First, s_1's data are copied into a buffer in the shared memory, after which XWAY in d_1 notifies XWAY in d_2 that some data have been stored in the shared memory, allowing s_2 to get data from the shared buffer. When XWAY in d_2 receives the notification, XWAY wakes s_2 up and copies data in the shared memory to the corresponding user buffer of s_2 directly. After the transfer is completed, XWAY in d_2 sends a notification to XWAY in d_1 back. When XWAY in d_1 receives the notification, it wakes s_1 up if s_1 is blocked due to the lack of the shared buffer space.

We have defined a virtual device and a device driver to realize our design on Xen. The virtual device is defined as a set of XWAY channels, where each channel consists of two circular queues (one for transmission and the other for reception) and one event channel shared by two domains, as depicted in Figure 5. Note that those two queues contain the real data instead of descriptors representing the data to be transferred. The event channel is used to send an event to the opposite domain. An event is generated if either a sender places data on the send queue or a receiver pulls data out of the receive queue.

While the XWAY virtual device just defines the relevant data structures, the XWAY device driver is an actual implementation of the virtual device. In our design, the XWAY device driver serves upper layers as one of network modules which supports high-performance, reliable, connection-oriented, and in-order data delivery for inter-domain communication. The XWAY device driver manages XWAY channels, transfers data, and informs upper layers of any changes in the status of queues.

Creating an XWAY channel is a complex task. To open an XWAY channel, both XWAY device drivers in Figure 5 should agree to share some memory area and an event channel. This means that they have to exchange tokens of the shared memory and the port number of the event channel through another control channel

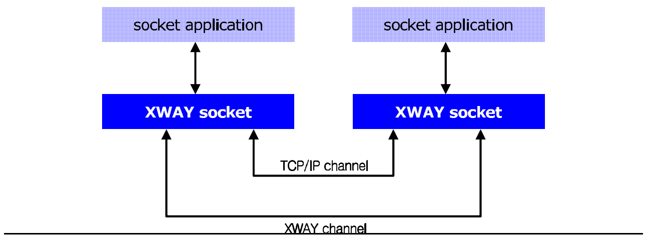


Figure 6. XWAY socket

across two arbitrary domains. We have decided to utilize a native TCP channel for the purpose of exchanging control information. XenStore would be used for the control channel since it provides information storage space shared between domains. However, since XenStore is mainly used for sharing semi-static data (configuration and status data) between domain-0 and guest domains, it is not appropriate for our environment where multiple kernel threads access the shared storage concurrently and tokens are generated dynamically for every channel creation request.

The next problem we face is to handle requests for establishing and closing an XWAY channel. Since they arrive at the destination domain asynchronously, we need to have separate threads dedicated to servicing those requests. Our design introduces *connection helper* and *cleaner* which handle the channel creation and destruction requests, respectively. The detailed mechanism will be discussed in Section 5.

4.2 Binary compatibility

There could be several different approaches to exporting the XWAY channel to user-level applications. The simplest approach is to offer a new set of programming interface that is similar to Unix pipes or to socket interface with its own address family for XWAY. Both approaches can be implemented either as user-level libraries or as loadable kernel modules. Although they provide inter-domain communication service with minimal performance loss, it is not practical to enforce legacy applications to be rewritten with the new interface. To be successful, XWAY should be compatible with the standard socket interface and allow legacy applications to benefit from XWAY without source code modification or recompilation.

A tricky part of implementing the socket interface on a new transport layer is to handle various socket options and to manage socket connections. Handling socket options requires full understanding of each option's meaning and how they are implemented in the network subsystem of the target guest OS. It generally takes a lot of effort to write a new socket module which implements all of socket options on a new transport layer. To reduce the development effort as much as possible, we introduce a virtual socket termed *XWAY socket* as shown in Figure 6. An XWAY socket consists of an XWAY channel and a companion TCP channel. The XWAY channel is responsible for sending and receiving data, and the TCP channel for maintaining socket options and checking the socket state. In fact, XWAY places most of its burden on the native TCP channel, and the XWAY channel handles data transmission only, sometimes referring to socket options stored in the TCP channel.

When an application requests a socket creation, the guest OS kernel creates an XWAY socket unconditionally, instead of a native socket. Since the kernel has no idea whether the socket will be used for inter-domain communication or not, the creation of the XWAY channel is delayed and the XWAY socket is treated as if it were a native socket.

Figure 7 illustrates the final architecture of XWAY, which provides full binary compatibility with the standard socket interface by inserting two additional layers right on top of the XWAY device

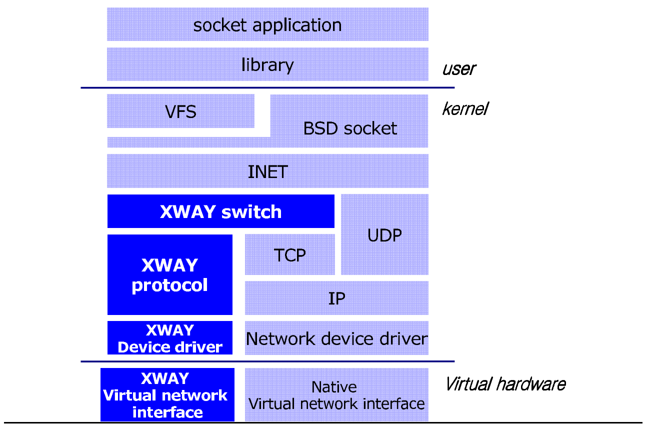


Figure 7. XWAY architecture

driver. Unlike SOP, we intercept socket-related calls between the INET and TCP layer.

A role of the XWAY switch layer is to determine whether the destination domain resides in the same physical machine or not. If it is in the same machine, the switch tries to create an XWAY channel and binds it to the XWAY socket. Otherwise, the switch layer simply forwards INET requests to the TCP layer. The switch layer classifies incoming socket requests from the INET layer into the following two groups:

- **Data request group**: send(), sendto(), recv(), recvfrom(), write(), read(), recvmsg(), sendmsg(), readv(), writev(), sendfile()
- **Control request group**: socket(), connect(), bind(), listen(), accept(), setsockopt(), getsockopt(), close(), shutdown(), fcntl(), ioctl(), getsockname(), select(), poll(), getpeername()

In principle, the switch layer redirects inter-domain data requests to the XWAY protocol and inter-domain control requests to the TCP layer. Unlike data requests, redirecting control requests are not trivial since the switch layer needs to take appropriate actions to maintain the socket semantics before redirecting the request. The actual actions that should be taken vary depending on the request, and they are explained in more detail in Section 5.

The XWAY protocol layer carries out actual data transmission through the XWAY device driver according to current socket options. For instance, the socket option `MSG_DONTWAIT` tells whether the XWAY protocol layer operates in non-blocking I/O mode or in blocking I/O mode. Socket options come from diverse places: function arguments, INET socket structure, TCP sock structure, and file structure associated with the socket. `MSG_PEEK`, `MSG_OOB`, and `MSG_DONTWROUTE` are some example socket options that are specified by the programmer explicitly, while `MSG_DONTWAIT` comes from either function argument or the file structure, `SO_RCVBUF` from socket structure, and `TCP_NODELAY` from TCP sock structure. The XWAY protocol layer should understand the meaning of each socket option and perform actual data send and receive reflecting the meaning.

Binary-compatible connection management is another important issue for XWAY because of the subtle semantics of socket interfaces related to connection management. For example, we have to take care of such a situation that `connect()` successfully returns even though the server calls `listen()` but has not reached `accept()`. In Section 5, we will conceptually describe how the XWAY channel interacts with the native TCP channel when an application tries to establish or close an inter-domain socket connection. Thanks to the XWAY device driver that hides most of the complexity concerning to connection management, it is relatively simple to create an XWAY channel and bind it to the XWAY socket. The switch layer only needs to call proper predefined interface of the device driver.

4.3 Minor issues

We think that high performance and binary compatibility feature of XWAY are just minimal requirement in practice. For XWAY to be more useful, we must take into account minor design issues, which we often miss. Those issues would be about portability, stability, and usability of XWAY.

No modification to Xen

It is not a good decision to make VMM provide inter-domain communication facility to the guest OS kernel just as the kernel provides IPC to user-level applications. This approach could deliver the best performance to the kernel but it makes VMM bigger which could result in unreliable VMM. Moreover, writing a code in VMM is more difficult than in kernel. Thus, we have decided to implement XWAY using only the primitives exported by Xen to the kernel. Those primitives include event channel, grant table, and XenStore interfaces. XWAY is not affected by changes in Xen internals as long as Xen maintains the consistent interface across upgrades or patches.

Minimal guest OS kernel patch

Since it is easier to develop and to distribute applications or kernel modules than kernel patches, kernel patches should be regarded as the last resort. The XWAY design enables us to implement the XWAY layer with minimal kernel patch. Currently, the use of a few lines of kernel patch is inevitable in implementing a small part of the XWAY switch layer. The rest of the switch layer and the other layers of XWAY are implemented with kernel modules or user-level applications.

Coexistence with native TCP socket

It does not make sense if the XWAY socket prevents applications from using the TCP socket for communication with another application outside of a machine. In our design, XWAY intercepts only inter-domain TCP communication, acting transparently to other TCP communication as if XWAY did not exist in the system. Though the peer domain of a TCP socket is in the same machine, XWAY does not intercept the socket connection if the peer domain's IP address is not registered.

Live migration

Live migration is one of the most attractive features provided by VM technologies. Thanks to the XWAY socket, it is not difficult to support live migration in XWAY. When one of domains communicating through XWAY is migrated to another machine, the domain can still use the TCP channel because Xen supports TCP channel migration. In fact, this is an additional benefit of the XWAY socket which maintains the native TCP channel inside. The implementation detail is beyond the scope of this paper.

5. Implementation

This section describes implementation details of XWAY. XWAY is composed of three major components: a few lines of kernel patch, a loadable kernel module, and a user-level daemon. Figure 8 shows the relationship among these components. We have implemented XWAY on Xen 3.0.3-0 with the Linux kernel 2.6.16.29.

5.1 Socket creation

Figure 9 outlines data structures for a socket that are used throughout XWAY layers. The normal socket structure is extended with the XWAY data structure. These data structures are created when `socket()` is invoked by an application. The underlined fields in the box surrounded by dotted lines in Figure 9 represent those fields as-

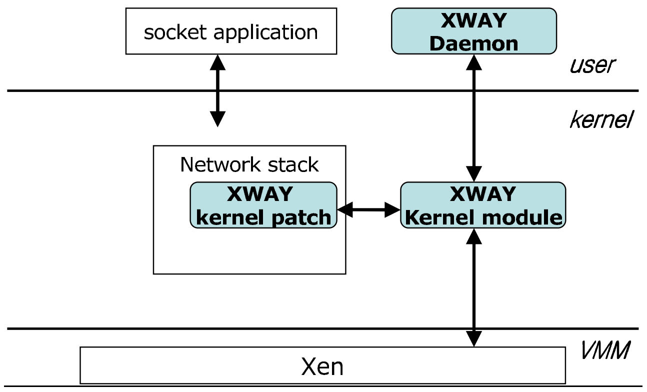


Figure 8. Code structure of the XWAY implementation

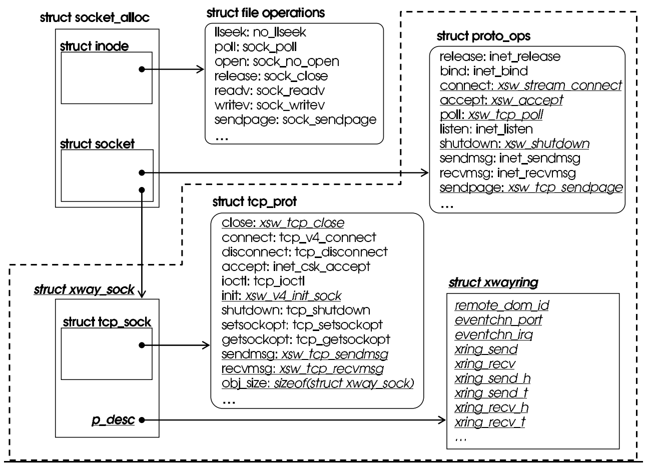


Figure 9. Socket data structures for XWAY

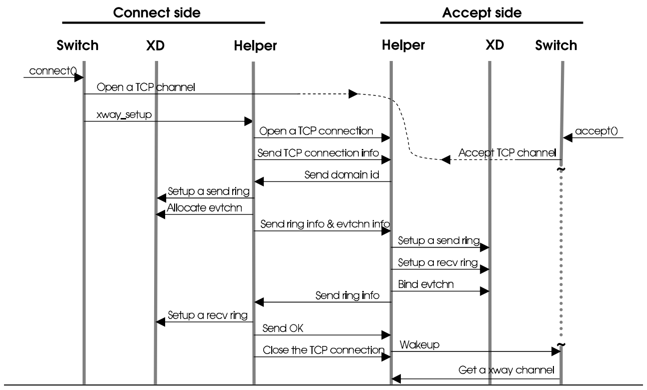


Figure 10. Establishing an XWAY channel

sociated with XWAY. The rest of the fields are borrowed from data structures representing the normal socket.

XWAY defines two main data structures, `struct xway_sock` and `struct xwayring`, and several functions whose names are underlined in `struct tcp_prot` and `struct proto_ops`. Note that XWAY functions are scattered throughout both `tcp_prot` and `proto_ops` strcutures. Those two structures hold a list of function pointers that are necessary to handle TCP requests and INET requests, respectively. The `p_desc` field is set to NULL when the corresponding `xway_sock` structure is created. If the socket has an XWAY channel, this field will point to the address of the `xwayring` structure. The XWAY channel is not created until the application requests an inter-domain connection establishment via `accept()` or `connect()`.

It is the `xway_sock` structure that realizes the XWAY socket without making any trouble for the TCP code. In fact, the TCP code is not able to recognize the presence of the XWAY subsystem. Although the structure maintains both a TCP channel and an XWAY channel, the TCP code treats it merely as a normal `tcp_sock` structure. Only the XWAY code is aware of the extended fields, following the `p_desc` field, if necessary, to access the XWAY channel that is associated with the TCP channel.

XWAY functions shown in Figure 9 intercepts INET requests from the BSD layer and TCP requests from the INET layer, and force the TCP code to create `xway_sock` instead of `tcp_sock`. The XWAY switch layer is implemented by replacing appropriate functions in the tables with XWAY functions. According to our design explained in Section 4, the switch layer lies between the INET layer and the TCP layer, so that TCP requests from the INET layer can be intercepted. In spite of this design, XWAY hooking functions are placed not only in `tcp_proto`, but also in `proto_ops` to make the result code more neat.

The `xwayring` structure describes an XWAY channel. It is created if the peer resides in the same machine when an application calls `connect()` or `accept()`.

5.2 Establishing a connection

When the switch layer receives `connect()` request, it first creates a native TCP channel and then tries to make an XWAY channel only if the peer is in the same machine. XWAY identifies the physical location of a domain using IP address(es) assigned to it. Since IP addresses for each domain are already stored into a branch of XenStore by the system manager, the switch layer simply contacts the IP address storage in XenStore.

Establishing an XWAY channel implies that the device driver builds a pair of circular queues and an event channel shared by both end-points. Figure 10 illustrates the sequence of interactions during connection establishment between the client and the server. The switch layer leaves all the jobs required for creating an XWAY channel to the helper. Initially, the helper at the server side opens a TCP channel with the helper at the client side. The TCP channel is used as a control channel and is destroyed immediately after the connection is established. Through the TCP channel, both helpers exchange information required to share a certain memory region and an event channel. The information includes the domain identifier, memory references of the shared memory, the event channel port, and the TCP channel identifier. The TCP channel identifier indicates the TCP channel created by the switch layer, not the TCP channel created by the helper. The TCP identifier consists of four attributes: source IP address, source port number, destination IP address, and destination port number, which is also used as an identifier for the XWAY channel as well as the TCP channel.

The next step is to set up send/receive queues and an event channel. Each end-point allocates a queue individually, and the other end-point maps the queue into its kernel address space, symmetrically. A send queue in one side plays a role of a receive queue in the other side and vice versa. As a result of this, both sides have the identical data structures.

Finally, the client-side helper registers the XWAY channel to `xway_sock`, while the server-side helper puts the channel on the list of unbound XWAY channels. When the switch layer of the server performs `accept()`, it looks for an XWAY channel that corresponds to the TCP channel returned by `accept()`. If found, it registers the XWAY channel to `xway_sock`. Otherwise, it sleeps until the corresponding XWAY channel becomes available.

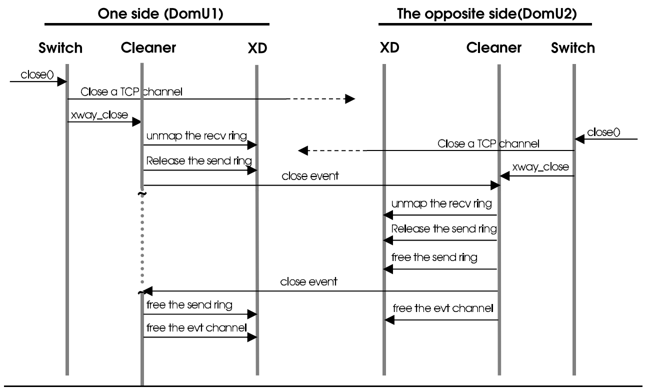


Figure 11. Tearing down an XWAY channel

5.3 Closing a connection

Closing a connection is a little bit harder than establishing one because each side is allowed to issue `close()` at any time without prior negotiation with the peer. In our implementation, we do not have to care about the TCP channel, but we do need to pay attention to the XWAY channel so that it can be destroyed correctly. Recall that the send queue at one side is mapped into the receive queue in the kernel address space of the other side, and vice versa. Figure 11 depicts general steps for closing an XWAY socket. The point is that the send queue in one side should not be freed until the corresponding mapping in the other side is removed.

When the switch layer receives `close()`, it closes the native TCP channel and asks the cleaner to destroy the XWAY channel. As the first step, the cleaner tries to remove the mapping of the receive queue from its kernel address space. It is very simple to unmap the receive queue because the cleaner does not have to check the state of the send queue at the other side. One thing to care about is that the cleaner must mark its receive queue with a special flag just before unmapping it. The flag is marked within the receive queue itself. When the other side sees the flag, it knows that the send queue is unmapped at the peer, hence it is safe to deallocate the queue.

After removing the mapping of the receive queue, the cleaner tries to deallocate the send queue. Figure 11 shows an example scenario where domU1 is not allowed to destroy the send queue because the receive queue in domU2 is not removed yet. In this case, though the connection is not closed yet, the cleaner returns the control back to the application immediately and it is the cleaner that is responsible for completing the remaining sequence. First, the cleaner waits for an event indicating the unmapping of the send queue from the peer. The domU1 is permitted to destroy the send queue after receiving the event. Finally, the event channel is removed at both sides. On the contrary, domU2 follows the same sequence as domU1 but it does not wait for any event to destroy the send queue. This is because the mapping of the receive queue in domU1 is already removed before domU2 tries to destroy its send queue.

Since one side can inform the other side of the unmapping of its receive queue through the shared memory and the event channel, we do not have to set up a separate control channel to close a connection.

5.4 Data send and receive

Once an XWAY channel is created, data transmission through the channel is relatively easy compared to the connection management.

Basically, data transmission on XWAY is performed as follows: a sender stores its data on the send queue (SQ) and the corresponding receiver reads the data from the receive queue (RQ) whenever it wants. Unlike the TCP protocol, it is not necessary to either divide the data into a series of segments or retransmit missing segments on timeout. We just perform a simple flow control around the circular queue; if SQ is full the sender is not allowed to write more data, and if RQ is empty the receiver is not allowed to read data.

When the switch layer receives `send()`, it redirects the call to the XWAY protocol layer. The protocol layer tries to add data on SQ by calling the device driver. If there is enough room in SQ, the device driver completes the send operation and returns back to its caller immediately. Otherwise, the next step depends on socket options. If `MSG_DONTWAIT` is specified for the socket, the protocol layer returns immediately, operating in non-blocking I/O mode. For blocking I/O mode, the device driver notifies the protocol layer of the state of SQ. Later, when the device driver detects changes in the state of SQ, it tells the protocol layer about the availability of free space (not necessarily enough space). The device driver sends a notification to the upper layer by calling a callback function. The callback function is registered to the device driver when the protocol layer is initialized. When the callback function is invoked, the protocol layer tries to send the remaining data again. The second try could fail if another thread sharing the channel fills SQ with its own data before the current thread accesses SQ.

Upon `recv()`, the switch layer also redirects the call to the protocol layer. The protocol layer tries to read data from the corresponding RQ through the device driver. As in `send()`, the next step is determined by the state of RQ and socket options. If any data is available in RQ, the device driver and the protocol layer returns with the available data immediately even though the amount of data is less than the requested size. If the device driver reports that RQ is empty, the protocol layer waits until any data is stored in RQ. The protocol layer wakes up when the device driver invokes a callback function to notify the availability of data. If RQ is empty and `MSG_DONTWAIT` is specified, the protocol layer returns immediately.

The XWAY device driver plays a basic role in exchanging data between two domains. It is responsible for the following four key functions: storing data to SQ, pulling data out of RQ, dispatching events to the other domain, and informing the upper layer of changes in the queues through a callback. Basically, the device driver dispatches an event to the other side whenever SQ is full or a timeout signal is caught. Again, socket options may alter the behavior of the device driver. For example, if the `TCP_NODELAY` option is specified, the device driver dispatches an event for each sending operation from the upper layer. If not, the event is deferred until SQ is full or it catches a timeout signal. This results in combining several small size of data to a larger one, which reduces event handling overhead.

Callback functions must be registered to the device driver when XWAY is initialized. The device driver informs the protocol layer of arriving an event indicating changes in SQ and/or RQ. Note that the event indicating a socket close should be delivered to the cleaner, not to the protocol layer. The protocol layer wakes up the process(es) expecting data in RQ or free space in SQ by catching the callback.

Unlike conventional socket read/write operations, all the operations of an XWAY socket should pass through the switch layer to make a decision whether they are redirected to the XWAY channel or the TCP channel. Since the switch layer requires only one integer-compare operation for each send, the overhead hardly affects the overall performance.

6. Experimental Results

In this section, we evaluate the performance and the binary compatibility of XWAY using various real workloads.

6.1 Evaluation methodology

We have evaluated the performance of XWAY on a machine equipped with HyperThreading-enabled Intel Pentium 4 (Prescott) 3.2GHz CPU (FSB 800MHz), 2MB of L2 cache, and 1GB of main memory. In our evaluation, we have used two versions of Xen: version 3.0.3-0 and 3.1. Xen 3.0.3-0 is used for evaluating XWAY and the native TCP socket operating on the virtual network driver. Xen 3.1 is for the native TCP socket with the improved virtual network driver and the enhanced hypervisor. We set up two guest domains for each test case. 256MB of main memory and one virtual CPU are assigned to each domain. We have used native binary images compiled for the standard socket interface for all applications tested. The binary images are neither modified nor recompiled for XWAY.

We have conducted five kinds of evaluations. First, to quantify the basic performance, we measured the bandwidth and latency of Unix domain socket, native TCP socket, and XWAY socket by using netperf-2.4.3. When evaluating the native TCP socket, we considered two receive data paths: copying mode and page flipping mode. Besides the socket performance, we also measured the bandwidth of moving a fixed size of data in memory from one place to another repeatedly within the same process address space. In our memory copy test, we moved 1GB of memory in total and the resulting memory bandwidth acts as the upper bound for inter-domain communication performance.

Second, we have performed several experiments to see how fast network-based applications can transfer data from one domain to another through XWAY socket. The tested workloads are selected from popular applications such as scp, ftp, and wget. This test shows the real performance that users will perceive.

Third, to prove that XWAY is useful in practice, we run DBT-1 test, an open source TPC-W benchmark which simulates the activities of a business-oriented transactional web server. Since DBT-1 test is popular in open source community, the improved benchmark result will demonstrate the usefulness of XWAY in practice.

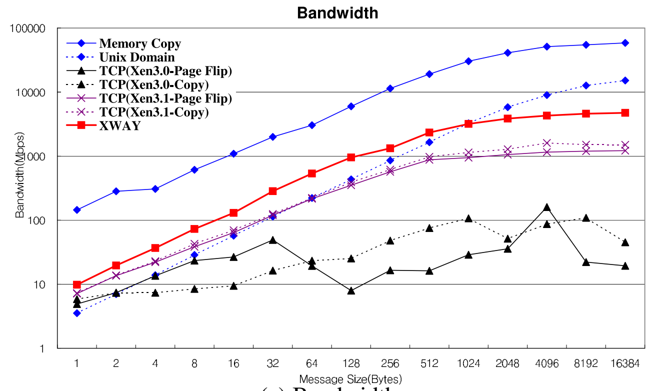
Fourth, we have experimented with a few other network-based applications to increase our confidence that XWAY ensures full binary compatibility with the existing socket interface. Finally, we examine the connection overhead in XWAY socket.

6.2 Basic performance

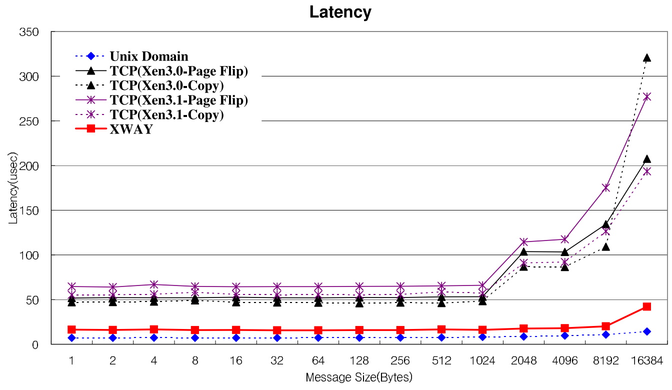
The netperf benchmark is composed of a netperf client (sender) and a netperf server (receiver). To measure the bandwidth, the sender issues a number of `send()`'s for a given time with best effort and waits until an acknowledgment message comes from the receiver. The latency is measured by a half of round-trip time in ping-pong traffic. The netperf results are shown in Figure 12. Note that the bandwidth in Figure 12(a) is represented in log scale for the convenience of comparing the results. It is obvious that the bandwidth of memory copy significantly outperforms other mechanisms, serving as the upper bound for inter-domain communication performance.

The bandwidth of Unix domain socket is similar to that of XWAY socket. We originally expected that Unix domain socket would be far superior to XWAY socket, but truth was different from our expectation as shown in Figure 12. XWAY socket even defeats Unix domain socket for data sizes ranging from 1 byte to 1024 bytes. Unix domain socket creates one socket buffer whenever it receives `send()` and appends the buffer to the socket buffer list. Moreover, it does not combine several small-sized data into a larger one before sending. These two drawbacks explain the lower bandwidth of Unix domain socket. For the large data, however, the overhead is getting amortized.

The bandwidth of XWAY socket is much higher compared to TCP sockets in all settings as we expected. XWAY socket shows 3.8 Gbps when sending 2048-byte data. TCP socket operating on



(a) Bandwidth



(b) Latency

Figure 12. Bandwidth and latency

virtual network driver with page flipping mode shows only 35.75 Mbps and 1056.89 Mbps on Xen 3.0.3-0 and Xen 3.1, respectively, for the same condition. In Xen 3.1, the TCP socket bandwidth is noticeably improved, presenting more predictable and stable behavior. We can also see that the TCP socket bandwidth with copying mode is better than with page flipping mode in Xen 3.1. In spite of this, the TCP socket bandwidth is still far lower than the XWAY socket bandwidth.

Figure 12(b) illustrates that the latency of XWAY socket is longer than that of Unix domain socket, but shorter than that of any TCP socket. The domain switching time explains the difference in the latency. In our test environment, two end-points of a Unix domain socket reside in the same domain, while two end-points of XWAY socket and TCP socket are in different domains on the same machine. The latency test causes switching between two domains or two processes in ping-pong fashion. Since the domain switching time is usually longer than the process switching time, the latency of XWAY socket is not comparable to that of Unix domain socket. On the other hand, at least two domain switchings are required for a message to travel to the peer domain through TCP socket, while one domain switching is enough for XWAY socket.

We can observe that the latency of XWAY socket at 16KB data has been off stride. Such a phenomenon occurs because the size of the circular queue that is used for XWAY socket is currently set to 8KB. Data larger than 8KB cannot be transmitted at once.

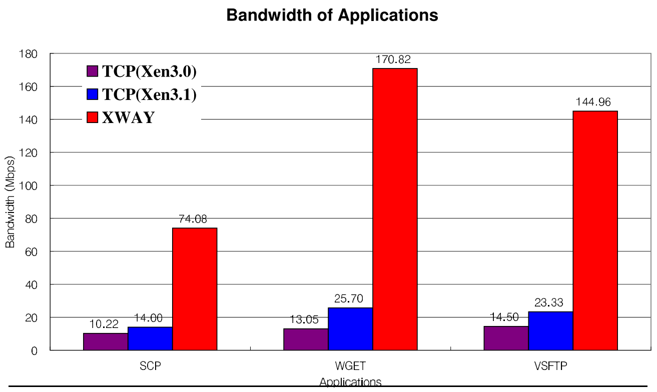


Figure 13. The application bandwidth on TCP socket and XWAY socket

6.3 Application performance

In this subsection, we measure the bandwidth of some real applications on Linux to evaluate the performance benefit that users will perceive. The tested workloads are scp, ftp, and wget, which are selected from popular Linux applications that are used for file transmission. In the test, we have transmitted 1GB of data from one guest domain to the other domain through TCP socket on Xen 3.0.3-0, TCP socket on Xen 3.1, and XWAY socket.

Figure 13 compares the bandwidth of each application under different inter-domain communication mechanisms. In line with the previous results shown in Figure 12, XWAY socket outperforms TCP sockets significantly, showing the higher bandwidth by a factor of 5.3 to 13.1.

Note that the actual application bandwidth is lower than the bandwidth measured by netperf. This is because the results shown in Figure 13 include the time for file I/O, while netperf measures the best possible bandwidth without any additional overhead.

6.4 OSDL DBT-1 performance

The configuration of DBT-1 benchmark is basically consists of three components: load generator, application server, and database server. Figure 14 illustrates the configuration of DBT-1 system on Xen for our test environment. The load generator is running on machine A, and both the application server and the database server are located in domain 1 and domain 2 of machine B, respectively. The load generator communicates with the application server through the native TCP socket. As application server relays incoming requests to and from the database server, the communication performance between these two servers can affect the overall DBT-1 benchmark result. Thus, we applied XWAY socket to the communication path between the application server and the database server.

Figure 15 compares DBT-1 benchmark results obtained under TCP socket and XWAY socket. In the graph, EUS denotes the number of users connected to the application server at the same time, whereas BT (Bogo Transaction) represents the number of web transactions serviced by the application server and the database server. The bigger BT value indicates the better performance for a specified EUS.

When EUS is 800, the performance of DBT-1 is improved by 6.7% by the use of XWAY socket; TCP socket shows 162.6 BT/sec and XWAY socket 183.5 BT/sec. The actual performance gain in this benchmark is much lower than the previous results. This is because DBT-1 benchmark is known to be CPU-bound instead of being network-bound. The CPU cycles saved by using XWAY socket make the database server perform more computation, but the benefit is not significant due to the CPU-bound nature of the benchmark.

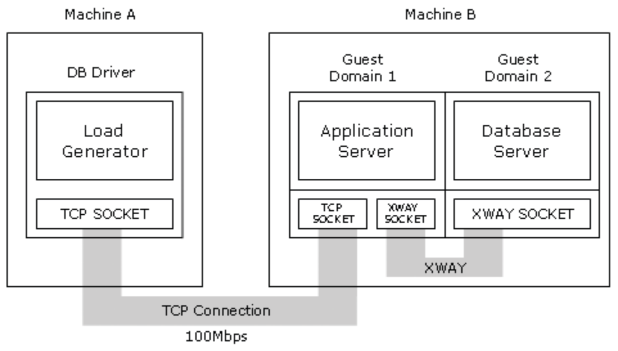


Figure 14. Configuration of DBT-1 benchmark on Xen

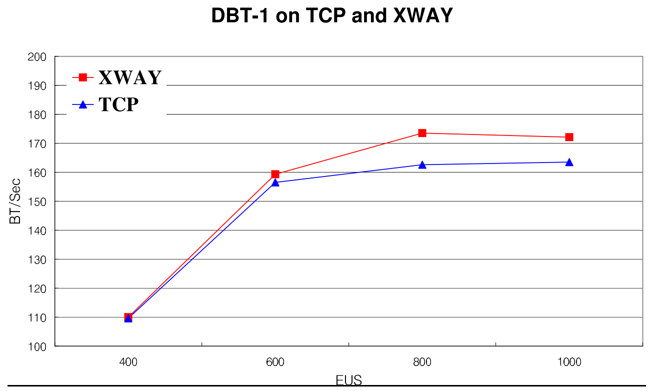


Figure 15. Results of DBT-1 benchmark

6.5 Binary compatibility

In XWAY, offering full binary compatibility with legacy socket-based applications is as important as achieving high performance. In order to test the binary compatibility, we have run a broad spectrum of network-based applications which use TCP sockets, as shown in Table 1. In these applications, the server is installed on one domain, and the client on another domain in the same machine. As can be seen in Table 1, all applications except SAMBA passed our compatibility tests.

SAMBA is different from the others in that it makes use of kernel-level socket interface. Since XWAY socket is not designed to support such applications that use kernel-level socket interface, the failure in SAMBA can be justified. We managed to make SAMBA work by modifying the source code of SAMBA client, but it is not stable yet. More work needs to be done to support the kernel-level socket interface.

Application	Pass/Fail	Application	Pass/Fail
SCP	Pass	WGET	Pass
SSH	Pass	APACHE	Pass
VSFTPD	Pass	PROFTPD	Pass
TELNET	Pass	MYSQL	Pass
NETPERF	Pass	SAMBA	Failed
FIREFOX	Pass		

Table 1. Results of binary compatibility tests

6.6 Connection overhead

So far we have focused on the overall bandwidth that XWAY socket achieves during data transfer through series of experiments. In

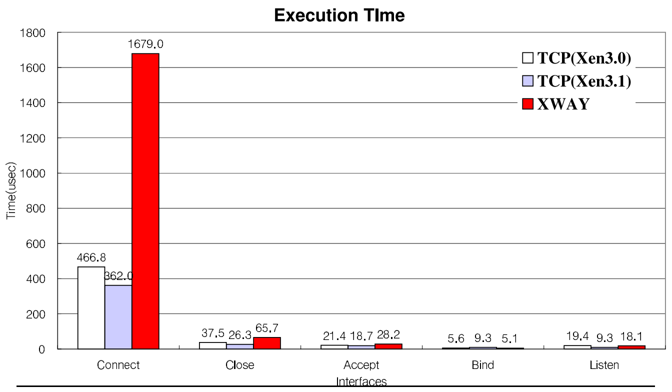


Figure 16. The execution times of socket interfaces related to connection management

this subsection, we measure the execution overhead of socket interfaces that are related to connection management: `connect()`, `accept()`, `bind()`, `listen()`, and `close()`. In general, the execution times of these interfaces with XWAY socket will be longer than the case with TCP socket since XWAY socket has a rather complicated connection management scheme.

Figure 16 compares the execution times of socket interfaces related to connection management. Note that `connect()`, `close()`, and `accept()` should take additional steps for XWAY as well as normal calls to the native TCP channel, while `bind()` and `listen()` does nothing special for XWAY. The time for `connect()` on XWAY socket has been nearly tripled compared to TCP socket. For `connect()` on XWAY socket, it is required to perform one TCP `connect()` to open an initial TCP channel, another TCP `connect()` and TCP `close()` to temporarily set up a control TCP channel, a few data exchange between helpers, the mapping for a receive queue, the allocation of a send queue, and the allocation of an event channel. The execution time of `accept()` on XWAY socket is slightly longer than on TCP socket. The difference mainly comes from the additional work that XWAY socket has to do such as looking for an XWAY channel from the unbound XWAY channel list. The execution time of `close()` on XWAY is roughly doubled owing to the fact that XWAY needs to unmap a receive queue, release a send queue, and deallocate an event channel during `close()`.

Although there is significant overhead in connection management under XWAY socket, these are the operations required only once for each connection. Since most network applications maintain their TCP sessions for a long time, the time taken by connection management does not dominate the overall network performance.

7. Conclusion

In this paper, we have presented the design and implementation of XWAY, an inter-domain communication mechanism supporting both high performance and full binary compatibility with the applications based on the standard TCP socket interface. XWAY achieves high performance by bypassing TCP/IP stacks, avoiding page flipping overhead, and providing a direct, accelerated communication path between VMs in the same machine. In addition, we introduce the XWAY socket architecture to support full binary compatibility with as little effort as possible.

Through a comprehensive set of experimental evaluations, we have demonstrated that two design goals are met successfully. XWAY socket exhibits the performance much superior to native TCP socket, and closer to Unix domain socket. The bandwidth and latency of XWAY for 2048-byte data are measured to be 3.8 Gbps and 15.61 μsec, respectively. Besides high performance, we have examined the binary compatibility of XWAY by running unmodified binary images of existing socket-based applications, such as scp, ssh, vsftpd, telnet, netperf, firefox, wget, apache, proftpd, and mysql. Since XWAY successfully runs these applications without any modification to source code or binary image, we are confident that XWAY ensures full binary compatibility with all the applications using the standard TCP socket interface.

We are currently working on supporting live migration over XWAY socket and plan to extend our design to support UDP protocol, Windows guest OS, domain crash resiliency, and so on. In addition, we will also optimize our implementation to match the performance of Unix domain socket and to use the shared memory more efficiently.

References

[1] R. Creasy. The Origin of the VM/370 Time-sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[2] R. Figueiredo et al. Resource Virtualization Renaissance. *IEEE Computer*, 38(5):28–31, May 2005.

[3] W. Huang et al. A Case for High Performance Computing with Virtual Machines. In *Proc. ICS*, 2006.

[4] J.-S. Kim, K. Kim, S.-I. Jung, and S. Ha. Design and implementation of user-level Sockets layer over Virtual Interface Architecture. *Concurrency Computat: Pract. Exper*, 15(7-8):727–749, 2003.

[5] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proc. USENIX Annual Technical Conference*, 2006.

[6] A. Menon et al. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. VEE*, 2005.

[7] I. Pratt. *Xen Virtualization*. Linux world 2005 Virtualization BOF Presentation, 2007.

[8] M. Rangarajan et al. TCP Servers: Offloading TCP Processing in Internet Servers. Technical report, Rutgers University, 2002.

[9] S. Son, J. Kim, E. Lim, and S. Jung. SOP: A Socket Interface for TOEs. In *Internet and Multimedia Systems and Applications*, 2004.

[10] X. Zhang et al. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Proc. Middleware (LNCS #4834)*, 2007.