A Study of Inter-domain Communication Mechanisms on Xen-based Hosting Platforms

Nan Li, Yiming Li, Member, IEEE

Abstract—In a virtualized multi-hosting environment, communication exists on different levels of abstractions and between different entities. Guest domains on the same machine communicate with each other via shared memory and message passing while guest domains on different machines communicate with each other over the local network. Significant performance overhead still exists in part of the communication architecture. Within the current literature, various research groups have accomplished domain-specific solutions in response to such issues. In our paper, we conduct a comprehensive survey, with sufficient analysis, on current inter-domain communication mechanisms on Xen-based hosting platforms. Techniques proposed by recent researchers to enhance the communication performance are compared and discussed from three perspectives, which are locations, access and management of shared memory regions. In addition, we also present detailed overviews of various topics regarding virtualization. At the end of paper, we present with the readers what we have learned during the investigation and what we suggest for future research in this area.

Index Terms—Virtualization, Xen, Communication, Inter-domain

I. INTRODUCTION

Recent advances in software and architectural support have led to a pervasive ongoing research activities in virtualization technology. This fact also further spurs a variety of applications where virtualization constitutes the primary role in implementation or performance optimization. Coined earliest in the 1960s as a term to refer to a virtual machine (sometimes called pseudo machine), virtualization has gained its resurgence within the past several years under the explosive development of modern computer technologies which enables sufficiently powerful machines to use virtualization to re-represent the underlying hardware and present the illusion of many co-existent virtual machines (VMs) [2]. Ever since the advent of this technology, numerous systems have been designed which use various virtualization techniques to subdivide the ample resources of modern computers [2], [9], [23], [33], [35], amongst which Xen [2], [10], [17], [19] serves to be one of the most representative virtual machine monitors (VMMs) for x86-compatible computers. In our paper, we conduct a comprehensive survey and present a detailed and informative overview accompanied by sufficient analysis, upon various issues related to communication between guest domains on Xen-based platforms.

Communication mechanisms constitute the heart of the study in our survey. On a multi-hosting virtualized environment, communication exists at different abstraction levels and between different entities. For example, guest domains on the same machine communicate with each other via either message passing or shared memory, which will be discussed in later sections. While at the same time, guest domains on different machines communicate with each other over the local network [10]. Besides domain-level communication, communication also exists between guest domains and VMM as well as between VMM and physical hardware. In our study, we focus on the inter-domain communication mechanisms. In our definition, inter-domain communication refers to communication between various guest domains co-existing on the same physical machine. Considering the pervasive research work conducted upon Xen-based platforms, we further refine our research goal as investigating various inter-domain communication mechanisms, their characteristics and their implementations on Xen-based multi-hosting environments.

The rest of the paper is organized as following. Section II provides a review of virtualization, including currently studied virtualization techniques, motivations for virtualizing hardware, applications that adopt virtualization as part of the methodology and an introduction to the Xen VMM. An overview of different communication mechanisms existing in the current Xen-based platforms is presented in Section III, where the drawbacks of those existing techniques are also discussed. Section IV constitutes the heart of our study. It discusses techniques that are utilized within the current literature to enhance inter-domain communication in Xen, in terms of performance and reliability. We include what we have learned during the survey together with what we might suggest to researchers for future research in the discussion in Section V. Section VI concludes the paper.

II. AN OVERVIEW OF VIRTUALIZATION

As aforementioned, before an exposition of existing work within the current literature regarding communication mechanisms in Xen is presented to the readers, an overview covering various conceptions related to virtualization is given in this section. We primarily focus on currently studied virtualization techniques, motivations for virtualizing hardware and applications that adopt virtualization as part of the
methodology. At the end, we provide an introduction to the extensively-studied VMM within the current literature, the Xen VMM.

A. Virtualization and Different Approaches

We first probe into the conceptions of virtualization, virtual machine, virtual machine monitors as well as different approaches to implement virtualization.

Virtual Machine and Virtual Machine Monitor

Generally speaking, virtualization refers to the creation and management of a virtual version of a resource or an entity, such as an operating system (OS), a server, a storage device, network resources, etc [10]. In our study, we use the term virtualization to refer to the virtualization of a physical machine at the operating system level. This is obtained by establishing an intermediate software layer called the virtual machine monitor (also known as hypervisor [19]) between the operating system and the underlying hardware. This VMM layer serves to be a re-presentation of the physical hardware, where how the hardware is represented depends on which virtualization technique is used. A VMM virtualizes the resources of a physical machine, such as processors, storage devices and network resources, and supports the co-location and co-execution of multiple virtual machines [2], [9], [24]. Each VM, sometimes also called as a guest domain [2], [10], [20], runs a separate operating system within it and the VMM provides safety and isolation to the overlying operating systems [10]. The term VM sometimes is referred to, by some authors, as the software layer that is actually the VMM in our paper. However in our study we refer to VM as the isolated environment that contains a commodity operating system and multiple applications and that runs on top of VMM in a shared machine. Fig. 1 visualizes a typical software layer structure of a VMM and its overlaying VMs [2], [5], [6], [20].

![Fig. 1. VMM virtualizes the hardware resources, partitions them into isolated VMs, on which multiple OSes and applications can run.](image)

Various Virtualization Techniques

As aforementioned, how the physical hardware layer is virtualized and represented to the guest domains in the higher level is contingent upon the virtualization techniques being used. There are a variety of different virtualization approaches and each of them has its own characteristics. On the x86 platform, the main virtualization techniques include binary translation based full virtualization, para-virtualization, hardware-assisted virtualization, simulator, pre-virtualization and so forth [4], [5], [6] [25]. Here we give simple descriptions to three of the typical virtualization approaches, full virtualization, para-virtualization and hardware-assisted virtualization.

- **Binary translation based full virtualization.** Full virtualization can be implemented using different techniques, one of which is to employ binary translation [1], [4]. Hardware assistance serves as another alternative to implement full virtualization, which will be discussed later. In full virtualization, the VMM simulates enough hardware resources to enable an unmodified guest domain to run in isolation [6]. Full virtualization combined with dynamic binary translation was first employed by VMware [6], [21], [35], [18]. It refers to the process of directly executing the unprivileged instructions. For those privileged operations, a dynamic binary translator is utilized to translate them into an equivalent block of unprivileged instructions. Full virtualization does not require either the OS modification or the hardware to be modified, therefore has the best compatibility. In addition to VMware, other virtualization implementations that are based on this approach include QEMU [27], Virtual PC [28], etc.

- **Para-virtualization.** Para-virtualization is a virtualization approach that has been proved to have a better performance and lower virtualization overhead [5], [29]. Para-virtualization establishes a software layer to the overlaying VMs, which is slightly modified from the underlying hardware. It allows VMM to be simpler and VMs that run on it to achieve performance closer to non-virtualized hardware. The primary reason for enhanced performance in para-virtualization is due to para-virtualized drivers. A para-virtualized driver knows it is used in a virtualized environment. As a result, the driver can generate optimized instructions to talk to the hardware directly [29]. The most acknowledged limitation about para-virtualization is that OSes have to be explicitly ported or modified, to be able to run on top of para-virtualized VMs [5]. The concept of para-virtualization was first introduced by Denali [23] and later popularized by Xen [2].

- **Hardware-assisted virtualization.** In hardware-assisted virtualization, the hardware facilitates VMM to host guest domains in an isolation manner through adding a new privilege layer to the processor [6], [22]. Due to the resurgence of virtualization research, hardware vendors start to embrace this technology by providing architectural support [6]. On the x86 platforms, both AMD and Intel have developed virtualization extensions to enable unmodified guest OSes to run on top of the hypervisor [5]. A typical example of hardware-assisted virtualization is the proprietary software Virtual Iron [30].

B. Motivation for Virtualization

Robert P. Goldberg describes in his 1974 paper that: "Virtual machine systems were originally developed to correct some of the shortcomings of the typical third generation architectures and multi-programming operating systems - e.g., OS/360" [31],
ever since the advent of this conception, virtualization researchers have been motivated by a variety of reasons, such as virtualization enables easier and faster application migration as well as secure co-location of antagonistic applications [10]. [31] specifies several reasons as following why virtualization technology is in demand.

- **Server consolidation and multi-hosting.** Virtualization enables multiple physical servers to be replaced by one larger physical server, thus increasing the utilization of expensive hardware resources. However, each OS is still able to run in an isolated environment known as a VM. The virtualization technology enables multi-hosting, which means a large server can host many guest VMs.

- **Easy control and inspection.** It is easier to control and inspect a VM from outside than a physical one, and its configuration is more flexible. Therefore it enables powerful debugging and efficient monitoring. This is very useful in kernel development and for teaching operating system courses [16]. In other words, virtual machines are great tools for research and academic experiments, since they provide isolation.

- **Easy relocation and migration.** A new VM can be dynamically created on demand without an up-front hardware purchase. Besides, a VM can easily be relocated from one physical machine to another as needed, therefore can be used in disaster recovery scenarios.

- **OS-level protection.** VMM provides an intermediate software layer between underlying hardware and overlaying guest domains, therefore it can isolate what they run on. As a result, it provides a fault and error containment, where faults can be injected deliberately and proactively into software to study its subsequent behavior. For example, virtual private servers (VPSes) [14] can serve as honeypots [31].

### C. Applications of Virtualization

Due to its powerful monitoring, isolation environment and easier migration, virtualization facilitates a variety of different applications. For example, due to its functions of isolation and security, virtualization is used to implement server consolidation [5], [10], where several independent services running in different OSes can be put in the same physical host. Some researchers also extend the application of virtualization by using it to build up intrusion detection systems [7], [8]. Virtualization has also been incorporated for OS kernel debugging [15]. As VMs constitute packages of complete machines, this can benefit mobile computing as well [5]. A research group at Stanford designs and develops the Collective system [3], which delivers managed desktops to personal computer users running in VMs.

### D. The Xen VMM

Introduced by Barham et al. at University of Cambridge in 2003 [2], Xen has been one the most widely used and studied VMMs on x86 platforms. Xen allows multiple commodity OSes to share real physical hardware in a secure and resource managed fashion, but without significant compromise of either performance or functionality. Xen plays an important role in popularizing the conception of para-virtualization [2], [5], which was first introduced by Denali [23]. Even though Xen started up as incorporating para-virtualization, later versions after Xen 3.0 have also adopted hardware-assisted virtualization to implement full virtualization [5]. Fig. 2 shows the architecture of Xen 3.0, which incorporates both para-virtualization and hardware-assisted virtualization. Various guest domains can run in different virtualization modes on the same physical server [5].

As a hypervisor, Xen is categorized as the “bare-metal” type [33], which means it runs directly on top of the physical hardware to present a virtualized hardware representation to the overlaying guest domains. Xen is the hardware control as well as the guest operating system monitor.

In Xen implementation, a special domain is created during boot time, which is responsible for hosting the application-level management software. This initial domain is termed as Domain0 [2], [5], [20], [32], sometimes also as the driver domain, the privileged domain or the control domain [5], [20]. Any other guest domain in Xen that is not Domain0 is termed as DomainU or unprivileged domain [20], [32]. According to Barham et al., the control interface used by Domain0 provides the ability to create and terminate other domains and to control their associated scheduling parameters, physical memory allocations and the access they are given to the machine’s physical disks and network devices [2].

Within the current literature, Xen has been extensively studied by various research groups, due to its open source property. For the next section, we will discuss various mechanisms related to inter-domain communication on Xen-based platforms and their limitations. Section IV will present innovative techniques to improve inter-domain communication in response to the addressed issues in Section III.

### III. AN OVERVIEW OF INTER-DOMAIN COMMUNICATION MECHANISMS IN XEN

On Xen-based virtualization-based hosting platforms (VHPs) [10], as shown in Fig. 3, communication on a guest domain level can be roughly categorized into two types, intra-node communication and inter-node communication [11]. A node refers to a physical server in a local network. Intra-node communication refers to the inter-domain communication on the same machine, while inter-node communication refers to
communication between domains located on different machines in the network.

![Diagram of Xen-based VHP architecture](image)

Fig. 3. Illustration of hosting in a Xen-based VHP (figure from [10]).

As previously mentioned, our study concentrates on the first type, namely the intra-node communication or the inter-domain communication on Xen-based VHPs. Intra-node communication can be further classified into data transfer and control transfer.

A. Data Transmission in Xen

Both shared memory and message passing mechanisms exist in data transfer between guest domains implemented in Xen [10], [20]. Inter-domain data transmission in Xen is implemented basically through a data structure called grant table [2], [20]. Each domain in Xen has its own grant table, which is a data structure shared with the VMM. Grant table allows the domain to inform the VMM what kind of permissions other domains have on its pages. Grant table is used to implement both shared memory and message passing between domains to enable data transmission.

**Shared Memory**

Each entry in the grant table is identified by the grant reference, which serves as a capability which the grantee can use to perform operations on the grantor's memory [20]. This capability-based system enables shared memory communication between unprivileged domains.

**Message Passing**

According to [10], Xen uses a shared-page mechanism called network-I/O-rings for inter-domain message passing. The detailed information of a shared page is also encapsulated in a grant reference inside a grant table [20]. Each domain has a reception ring and a transmission ring, implemented in circular queue data structures.

B. Notification Mechanism of Control Signals in Xen

Communication of control information between Domain0 and DomainUs happens frequently in the context of Xen. On Xen-based platforms, Domain0 hosts backend drivers [20] to access native device drivers, which access the hardware directly. Whereas all other unprivileged domains are only provided with frontend drivers, also known as the para-virtualized drivers or the virtual interfaces [20], [32], to enable them access virtual devices. When a DomainU intends to access a virtual device, it uses its virtual interface to connect to a corresponding backend network interface in the Domain0, which is in turn connected to the physical device driver (native driver) [10], [20]. Fig. 4 shows the network I/O architecture of Xen.

![Network I/O Architecture in Xen](image)

Fig. 4. The architecture of network I/O in Xen (figure from [20]).

As shown in Fig. 4, data transfer between virtual interface in DomainU and backend interface in Domain0 is obtained over an I/O channel, which uses zero-copy mechanism to implement the transmission [20]. As mentioned by [20], [29] and [32], XenBus constitutes the primary virtual bus on which data transfer between frontend driver and backend driver is implemented. In practice the bus is used for configuration negotiation, which is used in combination with XenStore [20], [29], [32], to exchange control information between Domain0 and other domains. Located in Domain0, Xenstore is a centralized configuration database that is accessible by all domains. Management tools configure and control virtual devices by writing values into the database that trigger events in virtual drivers.

**Page Flipping in Xen**

Page flipping is a technique that is used for transmitting data between domains by the unit of page [20]. Originally, it is designed to exchange pages of data between DomainUs’ frontend drivers and Domain0’s backend drivers [10]. It makes the effect of transmitting data by transferring an ownership of the physical page. Fig. 5 illustrates the process of data transmission using page flipping.

![Page Flipping Illustration](image)

Fig. 5. An illustration of page flipping in Xen (figure from [20]).

The page flipping mechanism simplifies the procedure of sending a page size data from one domain to another and is often used in data transfer in kernel drivers. However, it leads to lower performance and high CPU overhead due to excessive hypcalls to remap and swap pages. Page tables and TLBs also need to be flushed since the mappings from virtual to physical
All techniques proposed take the advantage of grant table mechanism of Xen to create shared pages across different domains. While in [11], [12], [13] and [26], the region is created in kernel of Xen (Domain0), Youself et al. propose a new way to create shared buffer in user-level, by patching grant table of Xen and exposing the interfaces to programmers, the result of which is native execution speed in a para-virtualized setting [25].

### B. Access of shared memory regions

This subsection focuses on how different implementations make use of shared memory, also it serves as an overview of general designs of different implementations under our survey.

In [11], the authors first developed a VM-aware communication library (IVC) to support shared memory communication. IVC is responsible to manage primitive operations of creating shared memory and schedule communications in fine-grain. Then MVAPICH2-ivc, which is set of VM-aware MPI over IVC is developed. MVAPICH2 is a modified version of MVAPICH2 [34], a popular multi-method MPI-2 implementation. An unmodified MVAPICH2 can also run in VM environment, with all communication going through network if the processes are in different VMs, which is a potential source of overhead. On the other hand, MVAPICH2-ivc provides support of shared memory in different VMs for MPI applications.

Fig. 6 (a) and (b) is the comparison of implementations of MVAPICH2 and MVAPICH2-ivc, as we can see, shared memory communication channel is replaced with an IVC channel, also a communication coordinator level is added to manage IVC connections.

From the view of applications (programmers), all complex implementation details are hidden behind the interfaces exposed by MVAPICH2 and IVC. IVC provides a socket style interfaces, e.g. ivc_write and ivc_read, also it takes little effort to rewrite existing MPI applications to take the benefit of shared memory in VMs.

Basically, [12] is the continuing work of [11]. To address the problems found in during the implementation of MVAPICH2 [11], the authors of [12] enhance inter-VM communication by implementing a one-copy protocol, which drastically reduces the latency of large data transition.

In order to optimize their implementation, the authors also employ the idea of inter-VM grant/mapping cache [12] and fall-back mechanism [12], however, all these modifications are transparent to applications, which means programmers only
need to invoke interfaces exposed MVAPICH2-IVC, instead caring about the details of how the buffers are shared or how the connection is established. A typical scenario is like a client-server model demonstrated in [11],

Different from [11] and [12], the goal of [26] is to provide a VM-based environment for a data streaming processing system, which consists of multiple processing components and has a high workload. To serve for this purpose, the authors design a one way communication socket which is suitable for memory conservation and asymmetric broadcasting communications.

The socket was named Xen socket and export a set of API called XenSocket. Though it was claimed by the authors as Unix-domain-socket-like [26], it strictly follows the client-server model and no bidirectional communication is allowed. XenSocket uses shared memory for message passing, when a connection is established, a descriptor page and some buffer pages are reversed for sharing, while the former maintains control information and the latter actually holds data to share. Again XenSocket hides all details about how to manipulate shared buffer and control information behind its APIs, however a drawback of XenSocket is the grant table reference of shared pages has to be manually passed to connect() call, which is very inconvenient for users.

Based on the argument above, we can conclude that the usage of XenSocket is very specific in one way communications. Perhaps from the aspect of usability and compatibility, XWAY, a socket implement by [13] is the best practice.

XWAY is similar to XenSocket in that it also uses shared memory in kernel for data sharing to avoid page flipping. It also implements a set of socket-like interfaces. However, XWAY has some distinct features: it uses TCP channel to transmit control information, while uses shared memory to transfer data. In addition, since it makes it an explicit goal to achieve binary compatibility, in kernel level, it intercepts all TCP communication, and dynamically determines which channel should be used by the destination of packages. As illustrated by Fig. 7, the XWAY switch component is responsible for analyzing packages, if it found the destination IP address is the local machine, it will send the package through XWAY protocol, otherwise send it through traditional TCP protocol.

In [25], the authors promote memory sharing to a new level by allow user-level buffers sharing [25]. Instead of trying to hide all details in a library or socket API, they expose the shared buffer to programmers. The implementation or usage scenario is not very clear from the paper, except that they patched the grant table of Xen to enable memory sharing between different domains, instead of in the kernel. We may infer that they use standard grant table hyper calls (with their patch) to allocate shared buffers for applications.

To sum up, in [11], [12], [13], [26], some socket-like interface are developed, hiding the details of shared memory management. In [11] and [12] another layer above socket API is deployed for making parallel applications, while in [13] applications can run perfectly well without modifications.

C. Management of shared memory regions and socket communication

In this subsection, we discuss the detail implementation of the mechanisms mentioned above. Particularly, we focus on (1) how the shared buffers are manipulated because in Xen, shared buffers are valuable since the size and number of pages are both limited by grant table of Xen. (2) how socket connections are implemented, specially, initialization and shutdown of socket, because most of implementations define new communication protocols or use shared buffer to pass control message, instead of standard TCP channel, so synchronization of opening and closing connection needs careful designs.

In [11], after an IVC connection is established, a shared buffer (page-ring) in kernel is created, and the communications is conducted through producer-consumer algorithms. A ivc_write call puts data on producer segments and advances producer pointer, while ivc_read call read data from receiver segments and advances consumer pointer. Both read and write operations are non-blocking, and return immediately with the actual bytes read/write, so the computing processes have to care about data retry later if the buffer ring is not available.

For multiple processes co-working successfully, a process has to register its magic_id to back-end driver (communication coordinator) which resides in dom0 and get a handler of
computing group, by query the handler, the process get a list of other processes under the same magic number. Then the process can connect to a peer by issuing a connect call and pass the handler of a peer in the list.

The communication coordinator also cares about tearing down of a connection by maintaining a list of active connections between processes, when an ivc_close call is invoked, communication coordinator is informed, then it stops all incoming send and write requests to the connection, and fulfill queued read and write requests, revoke the shared buffer and notify the other end of communication that the connection is closed.

[12] is the continuous work of [11], and it uses the same set of interfaces to handle socket connections, however, its main contribution is a new protocol for shared memory communications, named one-copy protocol.

In the traditional two copy approach, the sender first copies data to the shared region of IVC kernel, then notifies the receiver, after that, the receiver copies the data to its user buffer. However, it may result to performance overhead, so [12] implement another approach, as illustrated in Fig. 9, it only copies the handler of shared pages, after the receiver discovers sending requests, it directly maps the to its user space using the grant references from the sender. In this way, the receivers only need to copy the data once.

[Fig. 9. One-Copy Protocol (figure from [12]).]

However, there are still challenges that the authors face: granting page access to remote domains at the sender side and mapping user buffers at the receiver side are both kernel-privileged computationally complex operations. So to address the first problem, they use a pipelined sending protocol, so that they just grant several pages at a time and send them immediately, when sending is in progress, they grant more pages. By using this protocol, all sending cost except the first one is hidden.

The second problem is partly solved by introducing Inter-VM Grant/Mapping Cache (IGMC), the bottom line of which is keeping pages mapped as long as possible at the receiver side so that if the sender user buffer is re-used, the receiver doesn’t need to re-map new pages and data can be directly copied.

Though sounds easy, the logic of caching is complicated in that first, sending side should also keep corresponding pages granted as well, at the same time, the caches need to be evicted because the total number of granted pages is limited. The authors try to make this tradeoff by optimizing their algorithm to keep the record of caching at both side and do 2Q scheduling [12].

At the end of their argument, the authors admit that there are some cases that the caching scheme work not well [12], so they also employ a fall-back mechanism: if the overall hit ratio is under a certain threshold for a long period of time, the communicating processes will turn back to two-copy policy, which serves as a complementary of one-copy protocol.

Now let’s look at XenSocket, as mentioned above, there are two types of memory pages shared in XenSocket: descriptor pages for control information and buffer pages for data. In [26], it is not so clear that how actually pages are shared, the authors just state that “data are first placed onto a memory page by the sender and then the page is remapped into the receiver’s address space”. We can see that this may be very similar to one-copy protocol.

To establish connections, both the sender and the receiver need to call socket() API. The receiver calls bind() to bind the socket to a particular address, at the same time, it allocates shared memory and returns the reference of granted pages. It also establish an event channel (through TCP, though not explicitly state). The sender call connect() to get the reference of shared memory then communication begins. All read and write operations are blocked if the buffer is not available.

Since the receiver offer the shared pages, XenSocket enforces that only the sender can issue shutdown() call first, to prevent synchronization chaos. On the other hand, the connections in XWAY are more sophisticatedly implemented.

When a socket connection call is intercepted by XWAY switch layer resides between INET layer and TCP layer, the switch first determine of the destination is the same machine, if so, it will establish a TCP channel for event exchanging, an WXAY channel for data transmission. Next, both receiver and sender set up send/receive queues and map them to the kernel address spaces of the other end of connection. For instance, if A and B are the processes to communicate, the send queue of A maps to the receive queue of B and vice versa. From this we can see that XWAY actually performs a two-copy memory sharing.

If the read and write requests get blocked if the queues are empty depends on the options that applications use when creating sockets, e.g. if MSG_DOWNWAIT is specified, the protocol operates in non-blocking mode. In this way, the decision is made by users, which is more flexible.

To successfully close a connection, first switch layer close TCP channel and remove the receiver queue in its kernel(say A’s kernel), and mark it as cleaned, when the sender(B) sees the kernel, it can safely deallocate its sender queue, then A and B may continue one-way communication which A send data to B, when B issues close() and unmapped its receiver queue, A will stop send data and destroy its send queue, hence the communication is over.
The details of management of shared memory and connections are not available in [25], so we will omit them in this subsection.

D. Other individual goal-related issues

Generally speaking, the purposes of using virtual machines are isolation for performance and security, resource utilization, manageability et al. However, the papers we investigated have their own goal as well.

Migration

Since the motivation of [11] and [12] is building a VM-based environment for HPC using Xen, they also talk about their implementation of inter-node communications and live migration of processes.

IVC expose to users a set of callback functions, and let user decide how applications act when a process is about to migrate and when the task is done. In the kernel level, the communication coordinator is responsible to keep tracking the states of a connection, when a process is about to migrate, the coordinator temporarily removes the connection from active list, and block all send requests from peers, after migration is done, the coordinator re-sets up the connection, fulfill all pending requests and notifies all peers that the new state of a process, then move the connection to active list. In this way, the migration is done elegantly and all complexity is hidden in the library.

Security

Security concerns are required in XenSocket by System S, the motivation of the project. The bottom line is pages are only shared from less-trusted domains and mapped by more trusted-domains. That’s why only one-way communication and sender-first shutdown mechanism are implemented. However, more information about security of XenSocket is not available from the paper.

Binary compatibility

Without modifying, network applications can run perfectly in Xen if the most appealing idea of XWAY. To achieve their goal, the authors didn’t try to implement the entire socket interfaces, instead, they insert a layer called WWAY socket, which consists of a TCP channel and an XWAY channel. When an application requests a socket creation, the guest OS kernel creates an XWAY socket unconditionally, then the switch layer decide if the connection is a traditional TCP connection or it is an inter-VM connection. This implementation avoids the extra work to learn all existing socket options.

E. Evaluations of Experimental Results

It is not easy to do a thorough evaluations of a mechanism of inter VM-communication. Some papers describe measurement of bandwidth and latency in communication; others test their implementations against micro benchmarks and application-level benchmarks.

In [11], the authors first test basic bandwidth and latency by testing “pingpongs” message between two processes, the result shows that the latency of IVC is just a little higher than native environment, while is far low than No-IVC inter-VM communication. The bandwidth is also higher than No-IVC communications, though in case of large message passing, the performance degrades due to cache result because of large shared buffer size.

They also use four application-level benchmarks to test MVAPICH2, which are NAS Parallel Benchmark Suite, LAMMPS, NAMD, SMG2000 and HPL, the result is encouraging in that they can achieve 11% performance boost in the NAS.

By employing one-copy protocol, [12] accomplishes better performance than its original version. Especially, the bandwidth is 6.5GB/s when the messages size is 512K, also bandwidth is high even in a cyclic access pattern [12]. In NAS benchmark, one-copy approach meets or exceeds the performance of native and two-copy configurations in all programs. For IS program, a 15% improvement over native and 20% over two-copy is reported.

In [26], only bandwidth are compared between XenSocket, Unix-domain Socket(native Unix) and TCP connection(DomU to DomU) using netperf benchmark. The experiments are divided in two parts: messages smaller than 10K and larger than 10K. In the first experiment, the bandwidth is far better than TCP while the max gap between XenSocket and native Unix is not so huge(9295Mb/s Vs 13852 Mb/s), and when in peak time, the bottleneck of XenSocket is CPU instead of network. The sender and receiver CPU utilization is 100%, while the domain-0 is almost idle, which proves that their work in bypass domain-0 in communication is valid. With the increasing of message size, both performance of native Unix and XenSocket begin to drop, and the authors attribute this phenomenon to cache effect.

In [13], netperf is also used for benchmarking, however more configurations are participated in the tests, such as directly memory copying and different version of Xen (3.0 and 3.1), this time XWAY is comparable to Unix domain both in latency and bandwidth. Then application-level experiments are conducted using DBT-1 benchmark, and only XWAY and TCP performance are compared, though in bandwidth XWAY outperforms TCP by a factor of 5, the performance gap in user-login and bogo transactions is not apparent. To evaluate binary compatibility, several popular applications like SSH, TELNET and FIREFOX are tested on XWAY, only SAMBA fails due to it call some kernel-level socket functions, which is not the aim of XWAY.

In [25], only some home-made micro benchmark are used to test the novel way of user application level memory sharing, there are two main algorithms, the first is a “pingpong” like control transfer program, the second shared an integer variable using P/V synchronization. The result shows that the shared memory is 50 times faster than TCP socket connection.

Although every mechanism looks promising and does a lot of effort to improve inter-VM communication, it is really hard to
compare and tell which is better outside their papers. Because (1) they use different benchmarks, and (2) the hardware and Xen used are different. In addition, since they have different goal of their project, the methodology they use are not same, e.g. some focus on the comparison between native Unix and their implementation, some focus on bandwidth while others focus on CPU utilization. From the chapters of related works in the papers, we can see the authors are aware of other’s work, but they just mention the difference conceptually, there is no direct comparison in performance is available, or maybe because the distinct goals and backgrounds of their projects, there is no point to directly compare these implementations. However, the statement in [25] that the latency is “lower than under any existing inter-OS communication mechanism” is somewhat irresponsible, without more data in details.

V. DISCUSSION

This survey we have conducted in terms of inter-domain communication mechanisms in Xen is a highly rewarding process to enrich our understanding with respect to the specific topic. In this section, we present what we have learned during our study together with what we would like to suggest to other researchers to continue working on this topic, based on our investigation.

A. What We Have Learned

During the survey process, we truly feel that the biggest challenge we have been facing is to conduct comparative study on a fairly uniform platform. Within the current literature, different authors have published different works related to inter-domain communication in Xen. Each of them also presents convincing experiment results to support their conclusion. However, due to the heterogeneity of their experiment environments, such as heterogeneity in hardware, testing benchmark and performance measurement, it is almost infeasible to compare works from different research groups on a fair level.

Besides, we have discovered, through our study, that different optimization techniques might be adopted for different application scenarios. For example, as discussed in Section IV, researchers who work on enhancing inter-domain communication performance in Xen for HPC applications chose different optimization technique than others. It is hard to come up with a definite conclusion to evaluate which technique is the best, considering the variety of applications they are applied to.

B. What We Want to Suggest

The essence of a survey is to inform, by furnishing the readers with an exposition of existing work with respect to a specific topic. Based on our study, we have come up some suggestions that we want to present to researchers, for the hope of facilitating their research work.

A more generic, customizable, scalable and fine-grained approach to improve inter-domain communication in Xen is desired in the future. So far, various authors have implemented their solutions targeted at specific application or domains. However, a more generic approach can be designed for the purpose of solving common problems exhibited by current solutions. Such generic approach can also serve as the foundation of a variety of application-specific methodologies that run on top of it.

Secondly, not many authors have addressed the problem with the trade-off between memory sharing of guest domains and their isolation. In our view, security should be discussed more in this research topic, considering the fact that a primary reason for virtualization design is to implement isolated environments for un-trusted services to safely share resources on the same physical machine. Therefore, when researchers are exploring possibility of creating communication channels on a user space level, security should be given more consideration.

VI. CONCLUSION

In this paper, we conduct a survey into the current literature upon various inter-domain communication mechanisms and their performances on Xen-based platforms. Before specific techniques are discussed, we first provide overviews on virtualization, including different approaches and their applications. After that, we focus on the Xen VMM and discuss various forms of inter-domain communication mechanisms in Xen and their limitations. Typical techniques designed by past researchers to overcome the limitations addressed are discussed afterwards. Eventually, we show the readers what we have learned during the survey and what we can suggest for future research in this area in the discussion section.

ACKNOWLEDGMENT

We would like to express our deep and sincere gratitude to Professor Rich Wolski for instructing an amazing class as well as supervising upon our project. CS270 has been an interesting and rewarding journey, which has considerably enriched our spectrum of knowledge and enhanced our skills as independent researchers. We would also like to thank all the students in the class, who contribute to all the discussion sessions to help us develop a better understanding of the papers.

REFERENCES


[29] http://searchservervirtualization.technet.com-tip/0.289483.sdi94_gei1281856.00.html


[34] MVAPICH/MVAPICH2 Project Website, http://mvapich.cse.ohio-state.edu