國 立 交 通 大 學

電 機 資 訊 學 院

資 訊 科 學 系

碩 士 論 文

低負荷虛擬機器內部通訊

Lightweight Inter-Virtual-Machine Communication

指導教授：張瑞川 教授

研 究 生：張明絜

中華民國九十四年六月

低 負 荷 虛 擬 機 器 內 部 通 訊
Lightweight Inter-Virtual-Machine Communication

研 究 生：張明絜　　Student：Ming-Chieh Chang

指導教授：張瑞川　　Advisor：Ruei-Chuan Chang

國 立 交 通 大 學

資 訊 科 學 系

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

低負荷虛擬機器內部通訊

研究生： 張明絜　　指導教授： 張瑞川 教授

國 立 交 通 大 學 資 訊 科 學 研 究 所

摘 要

複雜的應用與伺服器系統一般會利用 multi-tier 搭配 machine redundancy 來達到 load balance 與提升系統的 reliability。這不但會造成硬體成本的提高，系統管理的困難，也會使得 CPU 使用率大幅降低。除此之外，也因各系統均需要相互通訊，而使得內部與外部的通訊大量提升。虛擬機器技術提供了一個方式來降低機器數量與提升 CPU 使用率。此外，它也提供了降低內部通訊負載量的機會。

一般而言，這些內部的通訊還是利用傳統的網路通訊協定來達成。然而，有許多虛擬機器實際上是處於同一個實體機器上。根據這一點，我們可以利用簡易的通訊協定來提升通訊的速度。

在此論文中，我們利用同一個實體機器的特性，設計了 fast Inter Virtual-machine Communication (fast-IVC)的機制。該機制會在建立 TCP/IP 連線時，主動判斷收發端是否處於同一台實體機器上。若是，則改用較簡單的通訊協定來取代原本 TCP/IP 通訊協定。根據實驗，利用 fast-IVC 傳遞資料可以提升 50%到 150％的效能。

Lightweight Inter-Virtual Machine Communication

Student: Ming-Chieh Chang Advisor: Dr. Ruei-Chuan Chang

Department of Computer and Information Science
National Chiao Tung University

Abstract

Complex applications or Internet services are usually constructed by multiple tiers and machine to improve reliability. However, it not only requires high hardware cost and expensive management overhead but also incurs both inter and intra machine communication overhead. Moreover, the CPU utilization of each machine is usually low since many services are I/O-bound and each machine performs a less-complicated function in the multiple-tier architecture.

The virtual machine technology was proposed for consolidating the machines and improving the CPU utilization. In addition, it also provides an opportunity to reduce the overhead of the inter-machine communication.

Generally, machines communicate with each other via network protocols. However, virtual machines reside on the same physical host, making it possible to improve the performance of the communication among them.

In this thesis, we propose a mechanism called fast Inter Virtual-machine Communication (fast-IVC), which transparently turns the TCP/IP based network communication between two virtual machines on the same physical host into shared memory based communication. Fast-IVC will detect whether the communication endpoints reside on the same physical machine. If they are, the complex TCP/IP stack will be skipped transparently and the communication performance will be improved.

We implement the fast-IVC on Xen. According to the performance results, the performance improvement ranges from 50% to 150%.

Acknowledgements

I am so grateful to have much guidance from my advisor Professor R. C. Chang. He taught me the essential of research, guided me the way of thinking. I also very appreciate Dr. Da-Wei Chang. He advised me so that I can finish my thesis.

Besides, thanks to each member of the computer system laboratory for their encouragement and kindly help. I would like to thank my parents for their unlimited love. Finally, I thank all friends for all the joyous things that inspire my life.

Ming-Chieh Chang

Department of Computer and Information Science

National Chiao Tung University

2005/6

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Complex applications or Internet services are usually composed of multiple tiers. For example, a web service can be organized as three tiers: web server, application server, and data storage server. In order to improve the system reliability, machine redundancy can be usually used in each tier. Figure 1 shows a typical architecture of a three-tier web service. In this architecture, inter-tier traffic (e.g., request/response messages) can be large.

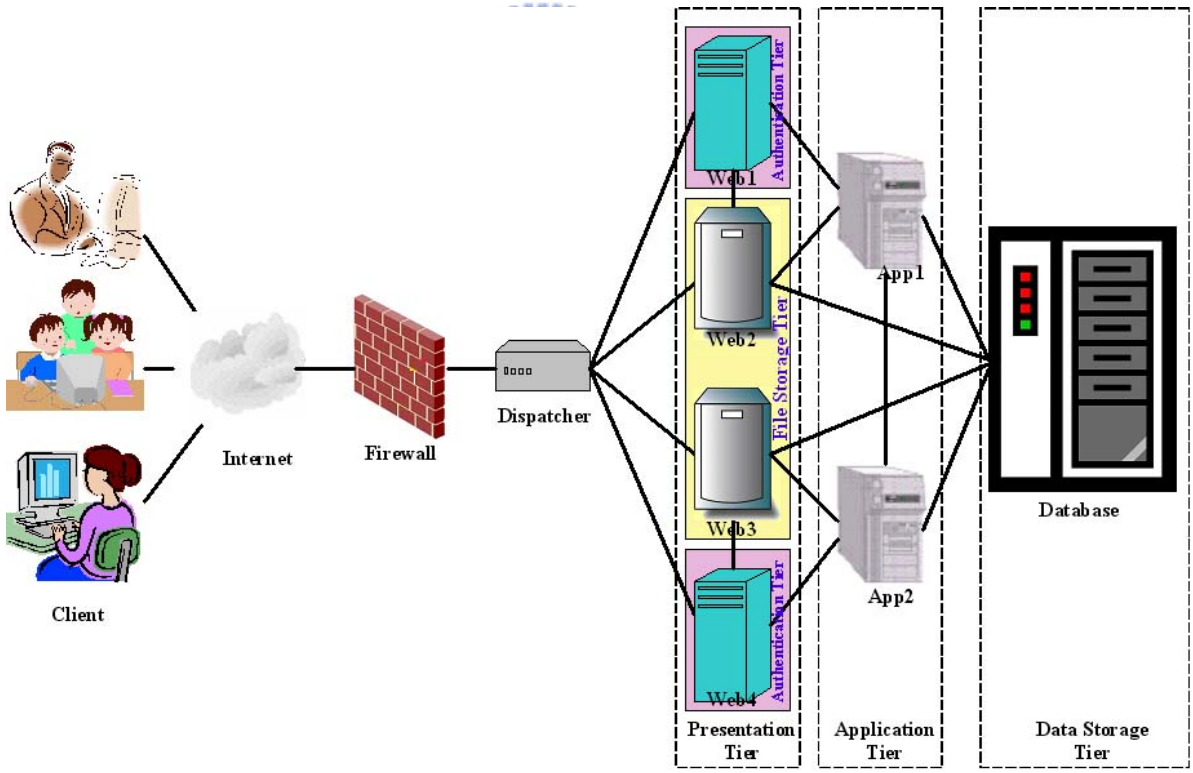


Figure 1 A 3-Tier Architecture

Some fault tolerant techniques based on machine redundancy, such as process pair [9], can results in large intra-tier traffic (e.g., state checkpointing/syncronization). The process

pair technique requires a (primary) server to synchronize its state with the backup server so that the backup can take over the job when the primary fails.

In addition to the inter-tier and intra-tier traffic, implementing a reliable service based on multiple machines also requires both high hardware cost and expensive management overhead. Moreover, the CPU utilization of each machine is usually low since many services are I/O-bound and each machine performs a less-complicated function in the multiple-tier architecture. Therefore, some companies consolidate the machines and improve the CPU utilization by using the virtual machine technology [34].

Virtual machine technology, which was introduced by IBM in 1960 [4][5][6][16][30], can provide many virtual machines, which are isolated with each other, on top of the a single hardware machine. Thus, a multi-tier service based on virtual machine technology results in less hardware and management cost and more effective CPU utilization.

In addition to the benefits mentioned above, the virtual machine technology also provides an opportunity to reduce the overhead of the inter-machine communication. Generally, machines communicate with each other via network protocols such as TCP/IP. However, domains[1] reside on the same physical host, making it possible to improve the performance of the communication among them. Specifically, the complex TCP mechanisms such as packet re-ordering, packet retransmission, RTT measurement, checksum, and etc., which are useful for Internet communication, can totally be skipped since the communication endpoints reside on the same physical machine. In this thesis, we propose a mechanism called fast Inter Virtual-machine Communication (fast-IVC), which transparently turns the TCP/IP based network communication between two virtual machines on the same physical host into shared memory based communication. Fast-IVC will detect whether or not the communication endpoints reside on the same physical machine. If they are, the complex

[1] A domain is a virtual environment in the virtual machine, and we will describe the details in Section 2.2

TCP/IP stack will be skipped transparently and the communication performance will be improved.

Several approaches have already been proposed to improve the performance of such inter virtual machine communication [10][11][20][27]. Most of them only reduce the overhead of the data link layer instead of the whole protocol stacks [20][27]. However, according to previous study [7], TCP/IP processing which can not be improved or skipped by these efforts dominates the network communication overhead. Inter-User Communication Vehicle (IUCV) [10][11], which was proposed by IBM do skip the protocol processing overhead. However, it is not application transparent. That is, service applications should be modified to obtain the performance improvement. The contribution of the thesis is that we propose a mechanism that can automatically detect whether or not the communication endpoints are in the same host and transparently skip the whole network protocol processing.

We implement the fast-IVC on an open source virtual machine monitor, Xen [1]. According to the performance results, the performance improvement of fast-IVC ranges from 50% and 150%.

1.2 Thesis Organization

The rest of the thesis is organized as follows. We describe the related works in Chapter 2. In Chapter 3, we describe the design and implementation of the fast-IVC. Next, we show the performance results in the Chapter 4. Finally, we give conclusions and future work in Chapter 5.

CHAPTER 2

RELATED WORK

The related work can be classified into three categories: remote procedure call optimizations, virtualization technology, and optimizations on inter-virtual machine communication. We will describe these efforts in the rest of this chapter.

2.1 Remote Procedure Call Optimizations

Remote Procedure Call (RPC) was introduced by Birell and Nelson [3]. It allows the caller and callee to be distributed over heterogeneous systems. When the caller invokes a procedure on a remote callee, the stub programs on both the caller and the callee hosts are responsible for marshalling/un-marshalling the arguments and return values, and sending/receiving the marshalled data through the network.

Two kinds of techniques were proposed to optimize RPC in a physical machine: shared memory [2][12][22] and scheduling path changing [13].

There are four copy operations for each cross-domain RPC request (two on call, two on return). In order to reduce the overhead, DASH system [22], light weight procedure call [2] (LRPC) and Peregrine [12] eliminate unnecessary copying by sharing data between the kernel and user domains.

Mach [13] uses the *hand-off scheduling*, which schedule the CPU context from the sender thread to the receiver thread directly if the two threads are in the same domain. This results in better performance since the arguments can be placed in the registers.

2.2 Virtualization Technology

The virtualization technology is an important technology for improving server availability and consolidating servers. It has been researched and improved for nearly thirty years [5][16]. Basically, the techniques can be classified into three categories: single operating system image, fully-virtualized virtual machine, and para-virtualized virtual machine.

Ensim[31], Vservers[35], CKRM[17] and Jails[14] are based on Single Operating System Image (SSI) technique. These systems delegate untrusted or less-trusted parties to manage a part of the system. ,e.g. two different web sites which are maintained by different parties, by allowing some (but not all) administrative functions. To provide isolation, the systems group user processes into resource containers so that a user process can not access resources outside the resource container. The drawback of this technique is that it does not perform well in fault isolation. For example, a kernel crash will cause all services on top of it become unavailable. Therefore, it is not suitable for a system with high reliability requirement.

Virtual machine technology, which was introduced by IBM in 1960 [4][5][6][16][30], was defined by IBM as “a fully protected and isolated copy of the underlying physical machine’s hardware”. It is designed for a company to isolate the environment of each user, so that users can not interfere with each other.

Figure 2 shows the typical architecture of a virtual machine [19]. The Virtual Machine Monitor (VMM), which is used to virtualize or extend the underlying hardware, is a middle layer between the guests and the host. A domain (or, a virtual machine) is a virtual environment in the guest, and hardware access from a domain will trap into the VMM. The host layer can be a bare machine or a host OS, both of them are used currently (e.g., VMware GSX and VMware ESX [34]). The operating system of a domain is called a guest operating

system. Note that many domains can run concurrency on a VMM.

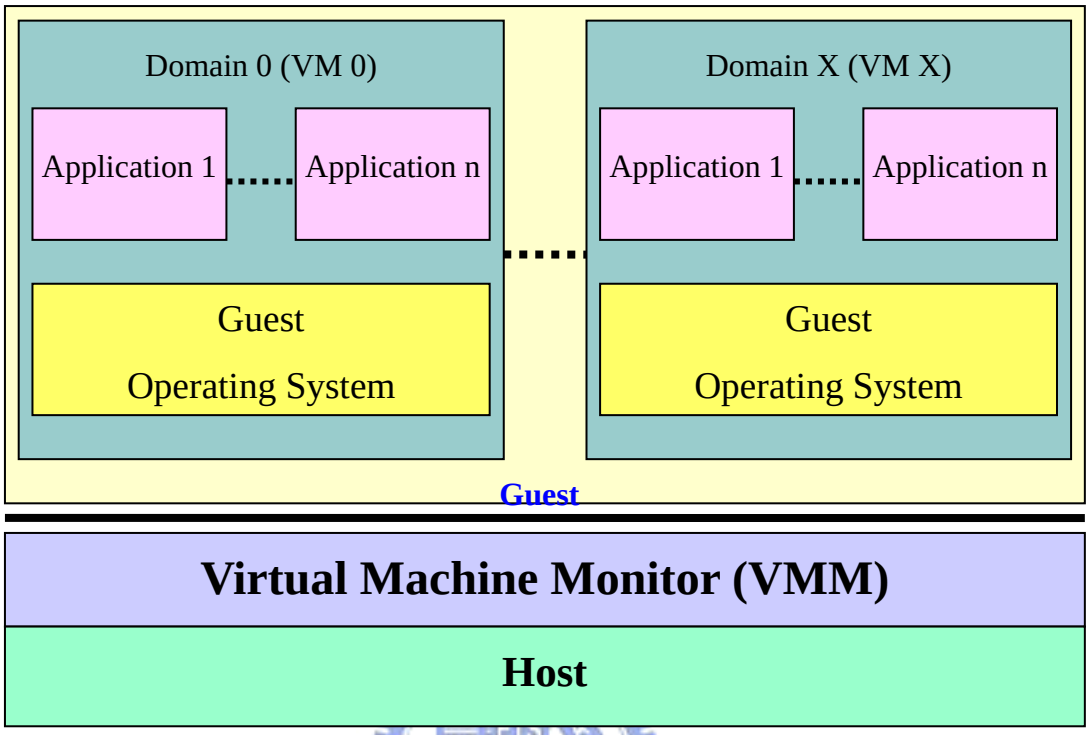


Figure 2 A Typical Virtual Machine Architecture

Traditional virtual machines [4][5][33][34] *fully* virtualize the underlying hardware so that any programs, including operating systems, can run on top of these virtual machines without modifications. However, some processor architectures such as x86 do not support virtualization very well [15][18], and hence fully virtualize these processors is not easy and requires high performance overhead.

To reduce the performance overhead, para-virtualization was proposed [24][26][25], which requires cooperation (i.e., some modifications) of the guest operating systems. In other word, these guest operating systems know that they work on the virtual machine. Our system, fast-IVC, is based on the para-virtualization VMM. A pair of sockets communicates with each other via a simple and shared-memory based protocol when the VMM detects that the ends of sockets are both on the top of the VMM.

2.3 Optimizations on Local Inter-Virtual Machine Communi -cation

Local inter-virtual machine communication (local IVC) is defined as the communication between different domains on the same VMM. Under this situation, the communication environment is much simper than the real network (i.e., no packet loss, no packet reordering, etc.) and thus performance improvement is possible by simplifying the protocol processing job.

IBM proposed Hipersocket [27][28][29], which is used to construct virtual LAN in z/VM [36], a virtual machine system for the IBM zSeries servers. It eliminates the MAC header and CRC processing, and directly copies data from the source data queue to the destination data queue. Sun also introduced a virtual LAN technique called InterDomain Networks (IDN) [20] on its StarFire Enterprise 10000 servers in the late 1990s. It allows domains to connect by using shared memory regions, which eliminates the memory copying of the Hipersocket. The main drawback of these techniques is that they only reduce the overhead of the data link layer, which is quite small in the whole TCP protocol stack [7].

IBM also proposed a technique called Inter-User Communication Vehicle (IUCV) [10][11], which provides a socket interface for communication between guest operating systems. It does simplify the protocol stacks. However, it is not application transparent. Specifically, the application should explicitly use the IUCV connections.

In summary, IUCV is not transparent to users while the others do not elimiate the overhead of the TCP/IP protocol processing. By contrast, fast IVC can eliminate the TCP/IP network protocol processing in a transparent way.

CHAPTER 3

DESIGN AND IMPLEMENTATION

In this chapter, we describe the design and implementation of fast-IVC, which is a mechanism to improve the performance of communication among different domains on the same physical machine. Section 3.1 describes the design goal, which is followed by the design of the fast-IVC mentioned in Section 3.2. The implementation details are shown in Section 3.3.

3.1 Design Goal

In our design, we expect to meet the following requirements:

- Performance Improvement

  Fast-IVC should improve the performance of communication among different domains on the same physical machine or VMM (i.e., local IVC).

- Transparency

  Applications should not aware of the existence of the fast-IVC. They use network sockets to communicate with each other as before. Fast-IVC should automatically transmit the data from the source socket to the destination socket if they are on the same VMM.

- Little Overhead

  More system resources are required while the sockets are on the same VMM. This is due to create communication tunnel between a socket pair. Fast-IVC should

occupy resources as little as possible.

3.2 System Design

As mentioned above, the communication environment of local IVC is much simpler than a real network since we do not need to care about the problems such as packet loss, packet reordering, etc. Therefore, we can use a much simpler protocol instead of the traditional communication protocols.

We designed a fast communication mechanism between domains on top of the same VMM called fast-IVC, which is illustrated in Figure 3. As shown in the figure, fast-IVC builds up a tunnel between the communicating domains and automatically switches the traditional network socket channel to the memory-based tunnel when the source and destination ends of the connection are on the same VMM. The switch is transparent to user applications and the performance of the local IVC is improved due to the skip of the network protocol processing.

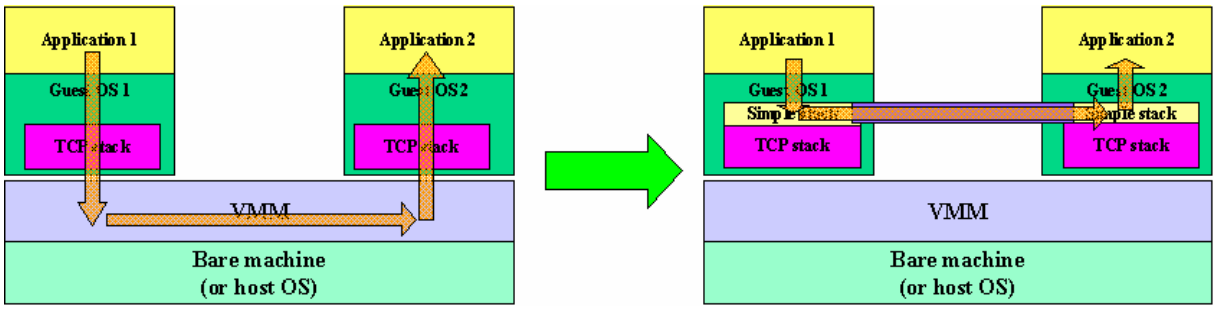


Figure 3 Comparison of the Original (left) Protocol Processing and fast-IVC (right)

Figure 4 illustrates the architecture of fast-IVC, which consists of four components: tunnel manager, protocol monitor, tunnel protocol and event manager. We will describe these components in the following sections.

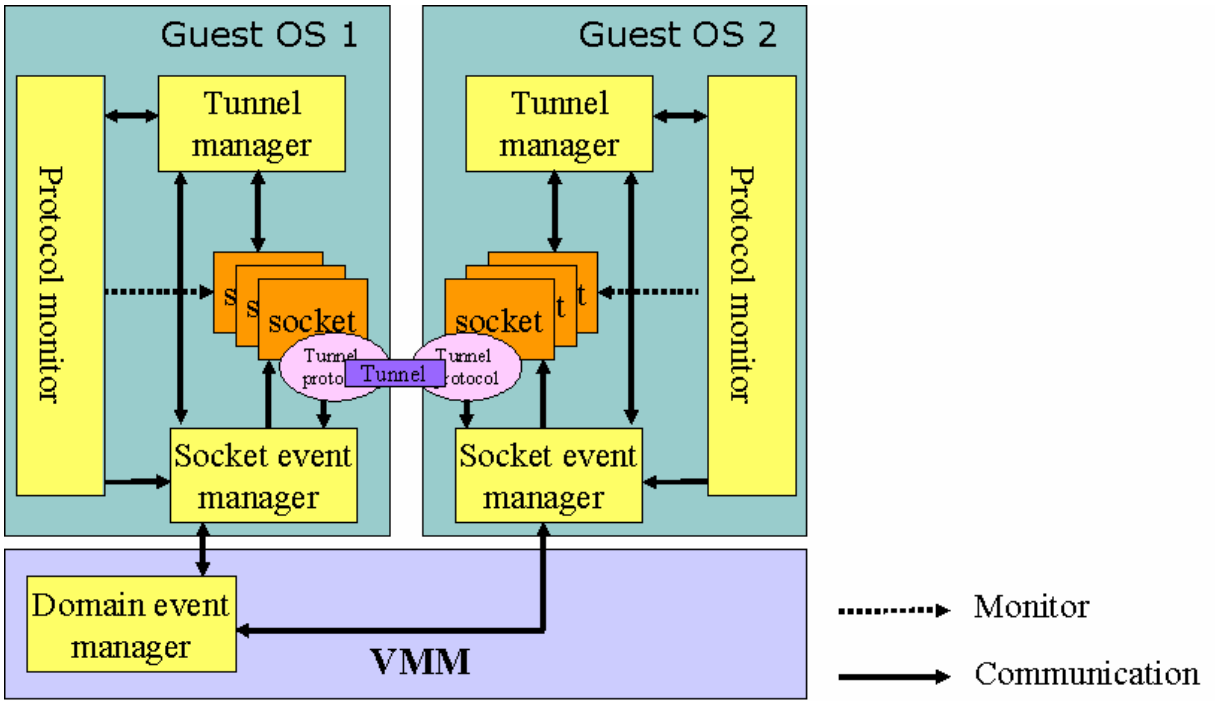


Figure 4 The Architecture of Fast-IVC

3.2.1 Tunnel Manager

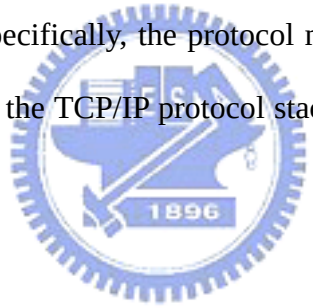
The tunnel manager is a component that provides functions to create or release a tunnel in the guest OS. A tunnel is actually a shared memory block, which will be created if the source and destination of a socket connection are on the same VMM. After a tunnel is created, the tunnel should be mapped into both sender's and receiver's address spaces. Tunnel creation will fail if one or both of the communication ends can not map the tunnel into their address spaces.

The memory block of a tunnel can be allocated from the memory pool of the VMM, the sender, or the receiver. We do not allocate tunnel memory from the VMM since its address space limitation. Moreover, allocating memory from sender or receiver makes no differences so we allocate the memory from the sender's memory pool.

3.2.2 Protocol Monitor

The protocol monitor detects whether or not a tunnel should be created by checking the destination IP addresses, and asks the tunnel manager to create a tunnel when necessary. As usual, a user creates a connection by using TCP sockets. If the source and destination are on the same VMM, the protocol manager will detect the fact and switch the communication protocol to the tunnel protocol. Otherwise, the communication goes through the traditional network protocol.

We intercept network protocol operations to perform the destination address checking. In TCP, the destination can be known after a socket executes *accept()* or *connect()*. In UDP, the destination is known when the socket executes sendmsg()/rcvmsg(). Therefore, we intercept the send/receive operations. Specifically, the protocol monitor determines whether to change to the tunnel protocol or to use the TCP/IP protocol stack when a send or receive operation is issued at the first time.



3.2.3 Tunnel Protocol

The tunnel protocol is a simple communication protocol used for local IVC. After a tunnel is mapped into both the sender's and the receiver's address spaces, the data will be transferred by the tunnel protocol. The sender pushes data into the tunnel, and the receiver is notified when the current memory chunk[2] is full or no more data needs to sent. When the receiver is notified, the memory chunk should be locked from the sender until the receiver reads all the data of the chunk. After all the data is read, the chunk can be used again by the sender.

[2] A tunnel consists of multiple memory chunks which are called *channels*, and we will describe the details in Section 3.4.5

3.2.4 Event Manager

In order to support sender-receiver cooperation during tunnel creation/release and data transmission, an event notification mechanism is needed. However, due to isolation maintained by the VMM, a domain can not send messages or events directly to the other domains. Therefore, we design an event manager to allow a guest OS to send events to VMM or another guest OS. Moreover, since the destination of an event is a socket, which is not aware by the VMM, we divide the event manager into two parts: Domain Event Manager (DEM) and Socket Event Manager (SEM). The former resides in VMM and is responsible for dispatching events to the corresponding domains, or getting events from domains. The latter resides in each guest OS and is responsible for dispatching events to the corresponding sockets and providing an event interface to the other components (e.g., protocol monitor) in the guest OS.

Table 1 shows the events that are provided by the event manager. When a tunnel is created in the sender side, the sender will notify the receiver by *EVENT_CREATE_ TUNNEL*, and receiver will map the tunnel into its address space so that the data can be transferred by the tunnel. The receiver notifies sender by *EVENT_REJECT_TUNNEL* if the tunnel mapping fails and the sender switches back to TCP/IP protocol when it gets this event. When a sender wants to close a tunnel, it sends the *EVENT_SEND_CLOSE_TUNNEL* event. Similarly, the *EVENT_ RECV_CLOSE_TUNNEL* event is sent when a receiver wants to close a tunnel. Finally, *EVENT_SEND_DATA* is used to notify the receiver to get data from the tunnel, and *EVENT_ RECV_GET_DATA* is used to notify the sender that the channel is free so the sender can use the channel again.

Name	Source/Dest.	Description
EVENT_CREATE_TUNNEL	Sender/Receiver	Sender creates a tunnel, and receiver can use the tunnel.
EVENT_SEND_CLOSE_TUNNEL	Sender/Receiver	Sender closes a tunnel, and receivers can close it.
EVENT_REJECT_TUNNEL	Receiver/Sender	Receiver can not map the tunnel into its address space.
EVENT_ RECV_CLOSE_TUNNEL	Sender/Receiver	Receiver closes the tunnel, and sender can close it.
EVENT_SEND_DATA	Sender/Receiver	Sender has put the data into the tunnel, and receiver can get data from the tunnel.
EVENT_ RECV_GET_DATA	Receiver/Sender	The channel (i.e., memory chunk) can be used again by the sender.

Table 1 Events Supported by the Event Manager

3.3 Discussions

TCP periodically send KEEPALIVE message if the connection is idle, e.g. no data, so that the other end of connection can know whether the connection is alive or not. In the fast-IVC, the TCP connection is not removed when the tunnel protocol is used. Because fast-IVC never sends data by TCP, like Figure 5, TCP will detect the connection state by the same mechanism. In other word, we can detect connection state by original TCP.

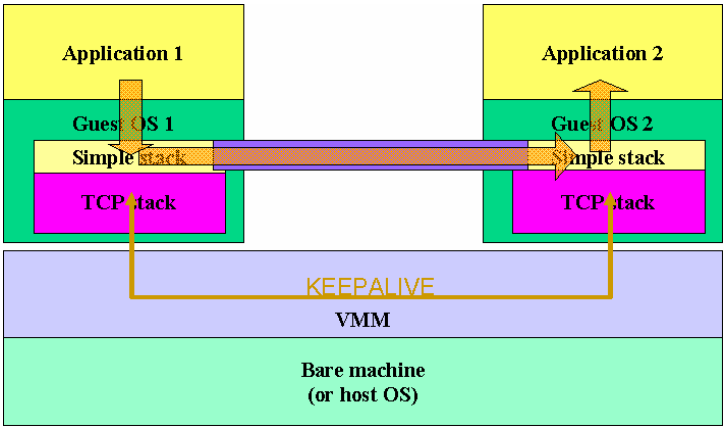


Figure 5 KEEPALIVE message

3.4 Implementation

3.4.1 Platform

We have implemented fast-IVC in Xen 1.2 [1] with Xenolinux 2.4.26. Xen is an open source and para-virtualized VMM, which was introduced by the Computer Laboratory in University of Cambridge. The Xenolinux is a modified Linux kernel running on Xen.

3.4.2 Socket Operation Interception

As mentioned above, we intercept the socket send/receive operations to perform the destination address checking. Each socket has a socket operation structure, *ops*, which records the pointers of the socket operation functions. The structure is initialized according to the type of the socket when the socket is created. For example, the structure refers to the TCP operation set (i.e., ops_tcp) when a TCP socket is created. To perform the interception, we modified the TCP socket creation code so that the structure refers to an initial operation set (rather the TCP operation set) when a TCP socket is created. As shown in Figure 6, the initial operation set checks whether the source and the destination are on the same VMM when the first time the send or receive operation is invoked. If they do, the structure will be set to refer to the tunnel operation set and the TCP/IP processing can totally be skipped. Otherwise, the structure will be set to refer to the TCP operation set and the socket communication is via TCP/IP.

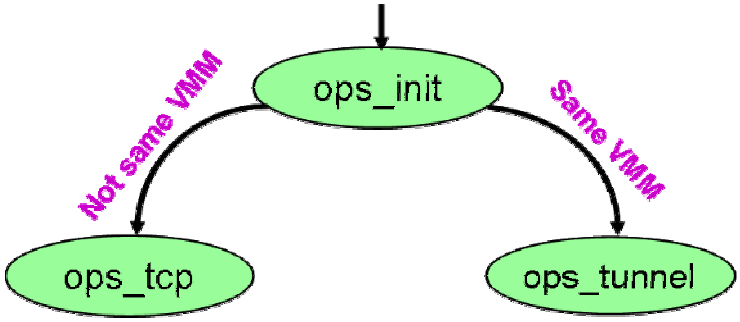


Figure 6 changing of *ops* filed

3.4.3 IP-to-Domain Mappings

When the socket send/receive operation is intercepted, the protocol monitor will check the destination address of the socket to see if the sender and the receiver are on the same VMM. The checking is straightforward. If the IP address of the receiver corresponds to a domain managed by the VMM, the sender and the receiver are on the same VMM. Otherwise, they are not. However, only the VMM (i.e., Xen) knows the mapping between the IP and domain. Therefore, the protocol manager has to invoke the VMM to perform the checking.

Invoking the VMM is high since it involves processor mode changing. One way to reduce the cost is to cache the IP-to-Domain mappings in the protocol monitor in each guest OS. However, we do not implement it in fast-IVC because it incurs the consistency problem; when an IP-to-Domain mapping is updated, DEM has to notify each domains.

3.4.4 Tunnel Creation and Release

Figure 7 shows the flow of a successful tunnel creation. First, a tunnel is created by the tunnel manager of the sender's guest OS when the IVC is identified as local. Each tunnel contains a number of memory pages and is initially mapped into the virtual address space of

the sender's guest OS. Then, the protocol monitor asks the socket event manager to send an *EVENT_CREATE_TUNNEL* event to the receiver(s). When the receiver gets the event, it obtains the tunnel information from the VMM and tries to map the tunnel into its address space. Note that the mapping may fail, as shown in Figure 8. Under this situation, the receiver sends back an *EVENT_REJECT_TUNNEL* event to the sender.

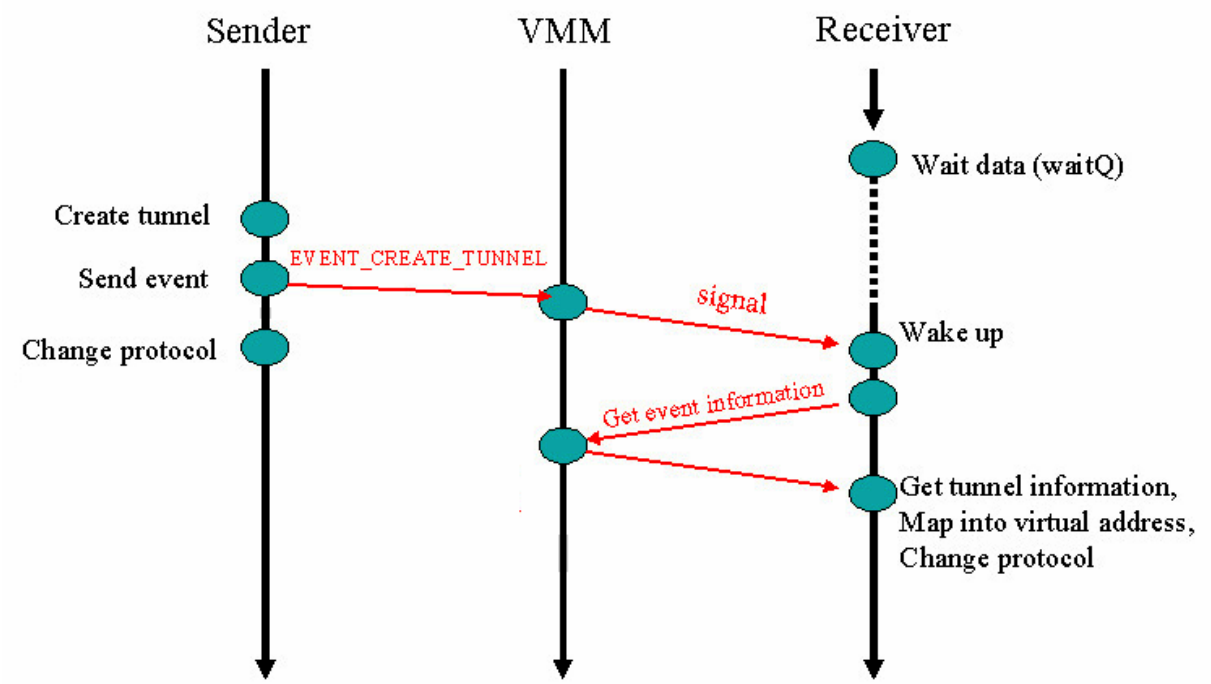


Figure 7 Message Sequence Chart for a Successful Tunnel Creation

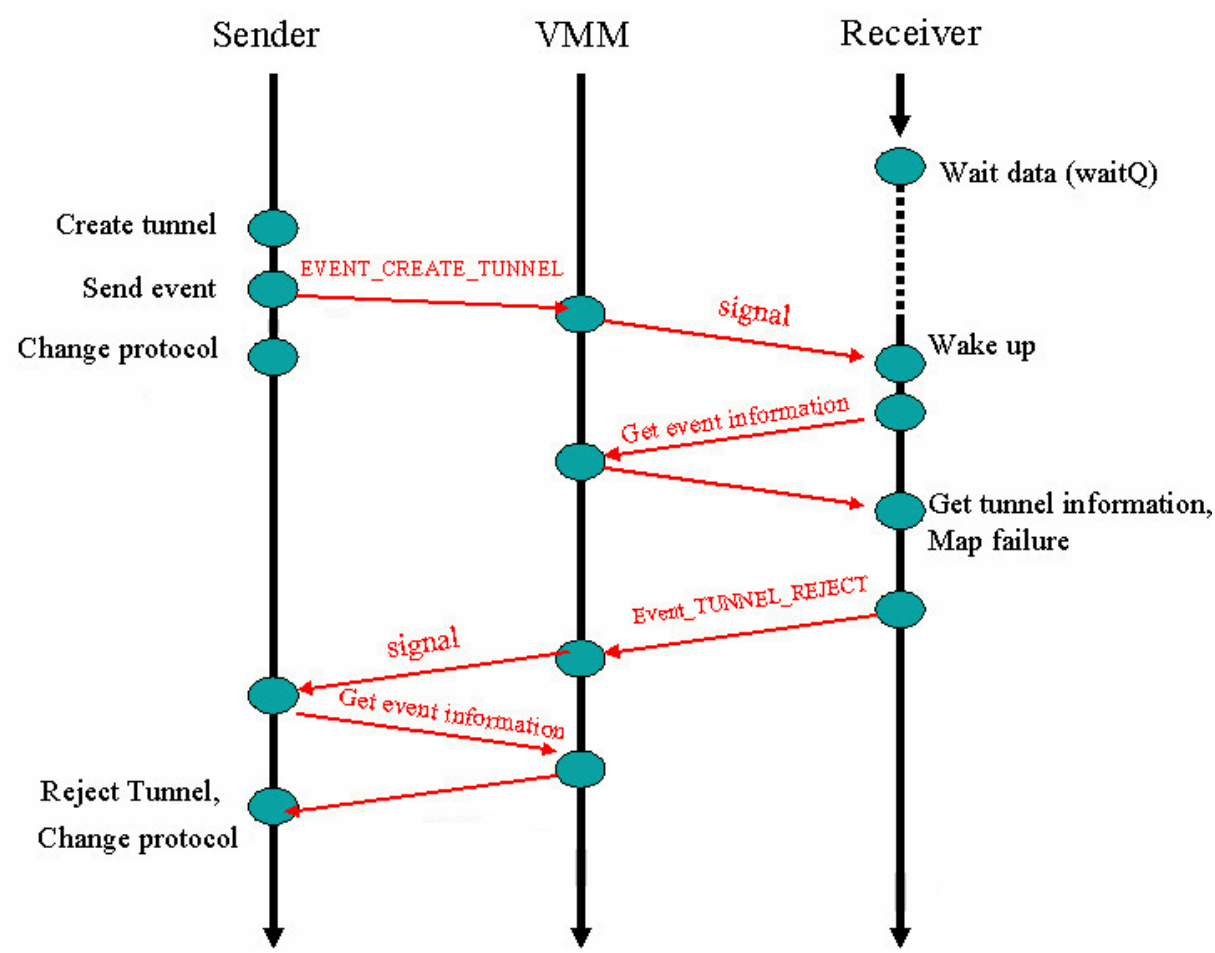


Figure 8 Message Sequence Chart for a Failed Tunnel Creation

The tunnel manager releases a tunnel when the sender or all the receivers close the tunnel. When a sender closes a tunnel, the tunnel manager will send an *EVENT_SEND_CLOSE _TUNNEL* event to all the receivers. Once a receiver is notified by the event, it should get the remaining data from the tunnel and then unmap the tunnel. The tunnel manager of the last receiver is responsible for sending an event *EVENT_ALL_RECV_GET_DATA* back to the tunnel manager of the sender when it unmaps the tunnel. Once the event is received, the tunnel manager of the sender will actually release the tunnel.

When a receiver actively closes a tunnel, the tunnel manager of the receiver will unmap the tunnel and send an event *EVENT_RECV_ CLOSE_TUNNEL* to the sender, which will modify the reference count of the tunnel. When the reference count reaches zero, the tunnel

manager of the sender will release the tunnel.

3.4.5 Tunnel Protocol

After both the sender and the receiver map a tunnel into their address spaces, they can follow the tunnel protocol to perform data communication. Before the description of the protocol, we describe the structure of a tunnel first, which is shown in Figure 9. In order to reduce the synchronization time between the sender and the receivers, we implement multiple channels in a tunnel. Therefore, the sender can write data to one channel while the receivers can read data from the other channel. Currently, a tunnel has four channels, each of which is a memory page.

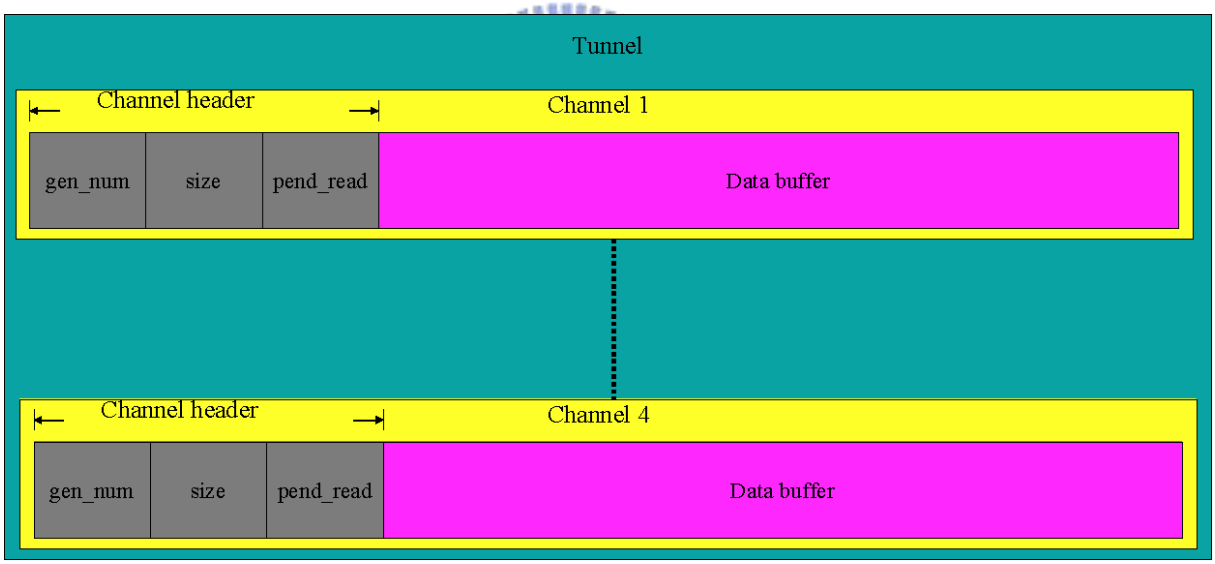


Figure 9 Tunnel Structure

Each channel has a header to keep the information about the tunnel. The header includes three fields: *gen_num*, *size* and *pend_read*. Similar to the concept of TCP sequence number, the sender maintains a sequence number which is increased by 1 whenever it begins to put data into a channel. After the sequence number is increased, the *gen_num* field of the channel

is set to the value of the sequence number to indicate the sequence of the data. Similar to the concept of ACK sequence number used in TCP, each receiver also maintains the next generation number that it expects to see. Before reading data from a channel, the receiver checks if the number equals to the *gen_num* field of the channel. If it does, the receiver can read data from the channel. The *size* field indicates the size of data in the channel. The *pend_read* field represents the pending readers of the channel and the channel should not be reused by the sender until the value of this field becomes zero. The field is set as the number of receivers by the sender when the channel is full or the sender has no more data to put into the channel. While a receiver gets all the data in the channel, it will decrease the value by 1. After all the receivers are done, the field will become zero and the channel can be used again by the sender.

The *gen_num* field of a channel is increased by 1 before the sender writes any data to the channel. Then, the sender copies data into the channel and sets the *size* field. After the data copy completes, the sender increases the *pend_read* field by the number of the receivers and notifies the receivers immediately if the channel is full or no more data needs to be sent. If the channel is full or more data needs to be sent, the sender notifies the receiver by event, then switches to the next channel and repeats the above job until the next channel is still locked from the sender.

Each receiver maintains the *wait_gen*, which is the next generation for waiting. When the *wait_gen* is equal to the *gen_num* and the channel is not empty and, it gets data from the channel. When all data is gotten for a receiver, the *pend_read* will be checked. If it is not zero, it will decrease *pend_read* by one. Once the *pend_read* reaches to zero, and the receiver will send an *EVENT_ RECV_GET_DATA* event to the sender. After decreasing *pend_read*, the *size* field. Since *pend_read* is not equal to zero and size is not equal to MAX_CHANNEL_SIZE only if sender is no more data to send, receiver can know data is sent completely.

3.4.6 Event Manager

The event manager is designed by two levels architecture. When a component wants to send an event to other domain, it will use the event interface which is provided by socket event manger (SEM), and each event is implemented by a hypercall[3], and the event is passed into domain event manger (DEM) which is implemented in the Xen. Then, DEM signals the destination domain by emulating interrupt signals which provided by Xen. However, the interrupt can not carry any information, so the socket event manager (DEM) which is implemented in the guest OS get the event information by hypercall *get_event_information()*

3.4.6.1 Domain Event Manager

The DEM is implemented in Xen, and is responsible for dispatching events to the corresponding domains. The event dispatching is based on IP-to-domain mappings maintained in Xen. Specifically, when a domain sends an event to another domain, the destination IP address will be passed to Xen for looking up the destination domain. Then, the event will be inserted into an event list of the destination domain, waitting for the SEM to get it.

Since events can happen frequently, the Xen-domain mode switches caused by the events will lead to a large overhead. We utilize three mechanisms to reduce the overhead.

First, since those events are not ordered, the DEM can store all the events for a socket in an per-socket *event_info* structure which is allocated in Xen when a tunnel is created. When an event is inserted into the structure, an *event_flag* field of the structure is set to indicate that there are pending events in this structure. When the SEM invokes the *get_event_information()* hypercall, all the pending events are returned to the SEM. Thus, a number of mode switches

[3] Similar to system call interface provided by an OS, hypercall is an interface provided by Xen to allow domains to request Xen to perform privileged operations.

can be eliminated.

Second, all *event_info* are linked in the event_list. Xen must search all list when it wants to find a event_info. To reduce the search overhead, we implement the *event_hint* in DEM. For each domain, DEM stores the pointer of the first corresponded *event_info* and the total number of the corresponded *event_info*, which are in the *event_list*. When a SEM calls *get_event_information()*, the DEM just gets event from the *event_hint*, and searches *event_list* to the next event if there is still a remaining event. Even it has to search *event_list*, it only needs to search from the event, which is store in the *event_list*.

3.4.6.2 Socket Event Manager

The SEM is implemented in the guest OS and is responsible for dispatching events to the corresponding sockets. It provides an interface so that other components can send and receive event. Each event corresponds to a hypercalls. When an event-related interrupt is triggered, ghost OS will invoke the *get_event_information()* hypercall to get the *event_info*, which contains the socket IP addresses, port numbers and other necessary information.

After the SEM gets the *event_info* by *get_event_information()*, it will map the IP addresses and the port numbers to the real memory address of the socket. We do this by the *tcp_v4_lookup()* functions which is provided by Xenolinux. Then do corresponding operations which are provided by other components.

CHAPTER 4

PERFORMANCE EVALUATION

4.1 Experiment Environment

We run Xen 1.2 and Xenolinux 2.4.16 on an Intel Pentium 4 1.6 GHz PC, which is equipped with 1GB DDR RAM, a 100Mbps Ethernet adaptor, and an 80 GB HD (Maxtor DiamondMax Plus 9, 7200 RPM, and 8MB internal buffer).

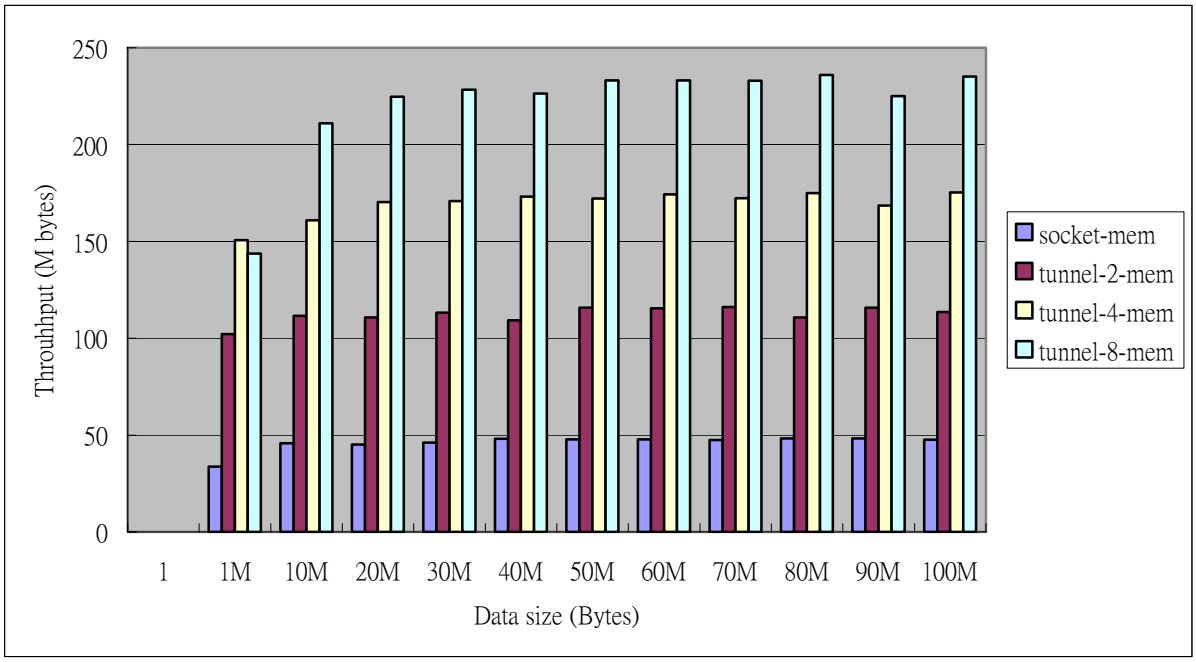
4.2 Maximize throughput when data in the memory

In this evaluation, we run test program on two different domains. The sender sends memory block to the receiver, and receiver does not write data into the disk when it gets the data. We set 128MB RAM and 8 GB virtual disk space for a domain. And we run 100 times for each block size.

Figure 10 shows the throughput comparison of TCP/IP and fast-IVC without disk overhead. We also show the performance of fast-IVC with different channel numbers to evaluate the effect of the channel number. Specifically, 2 (tunnel-2-mem), 4 (tunnel-4-mem), and 8(tunnel-8-mem) channels were evaluated. The throughputs of the 1-byte file are 1353.826, 1606.169, 1445.424 and 1328.372 bytes per second, which are too small to show in the figure.

In this experiment, three pointers are worth mentioning. First, the throughput of the 1-byte of tunnel-8 is small than others since it needs to wait a large continuous memory space when creating the tunnel. Second, excepting the results of tunnel-8 of the 1-byte file, the throughput of the tunnel protocol is always higher than that of TCP/IP, it means the transmission overhead of TCP/IP is much higher than the overhead of tunnel protocol so that

the overhead of tunnel creation can be covered. Third, performance improves as the channel number grows.



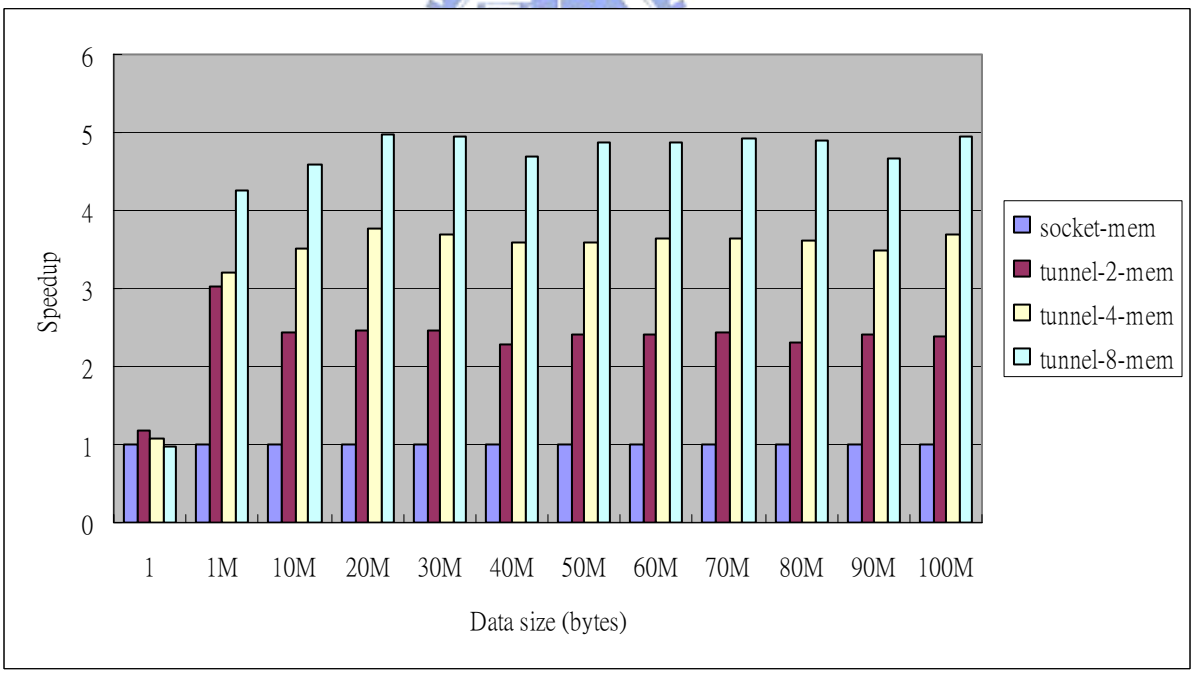


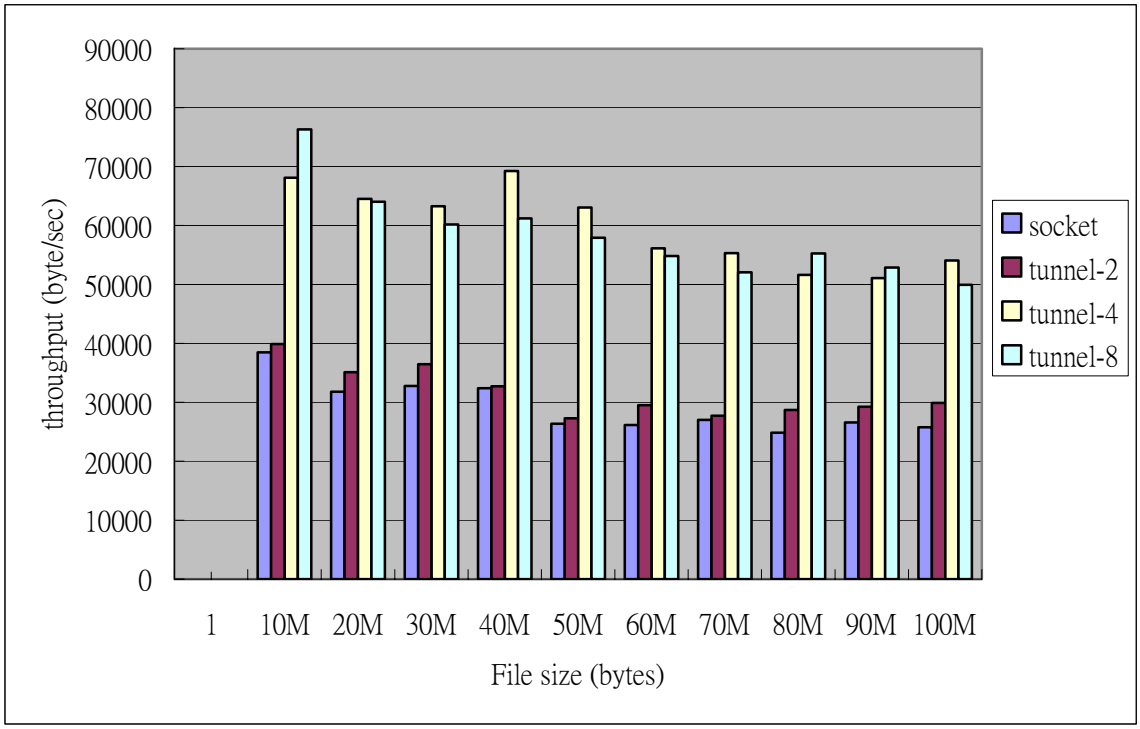
Figure 10 Max throughput (top) and its performance improvement (down) when data in the memory

4.3 Maximize Throughput when Data in the Disk

In this evaluation, we run vsftp server and ftp client on two different domains. Same as 4.2, each domain has 128MB RAM and 8 GB virtual disk space emulated by Xen. The client gets the files from the server, and we run 25 times for each file size.

Figure 11 shows the throughput comparison of TCP/IP and fast-IVC. We also show the performance of fast-IVC with different channel numbers to evaluate the effect of the channel number. Specifically, tunnel-2, tunnel-4, and tunnel-8 are related number of the channels. The throughputs of the 1-byte file are 17.07, 25.56, 26.06 and 15.73 bytes per second, which are too small to show in the figure.

One pointer is interested; the performance does not always improve as the channel number grows. Increasing the channel number from two to four does improve the performance since the memory buffer is enlarged and thus the waiting time of the sender and the receivers is reduced. However, the waiting time is small when the channel number becomes four, and thus increasing the channel number further does not lead to performance improvement.



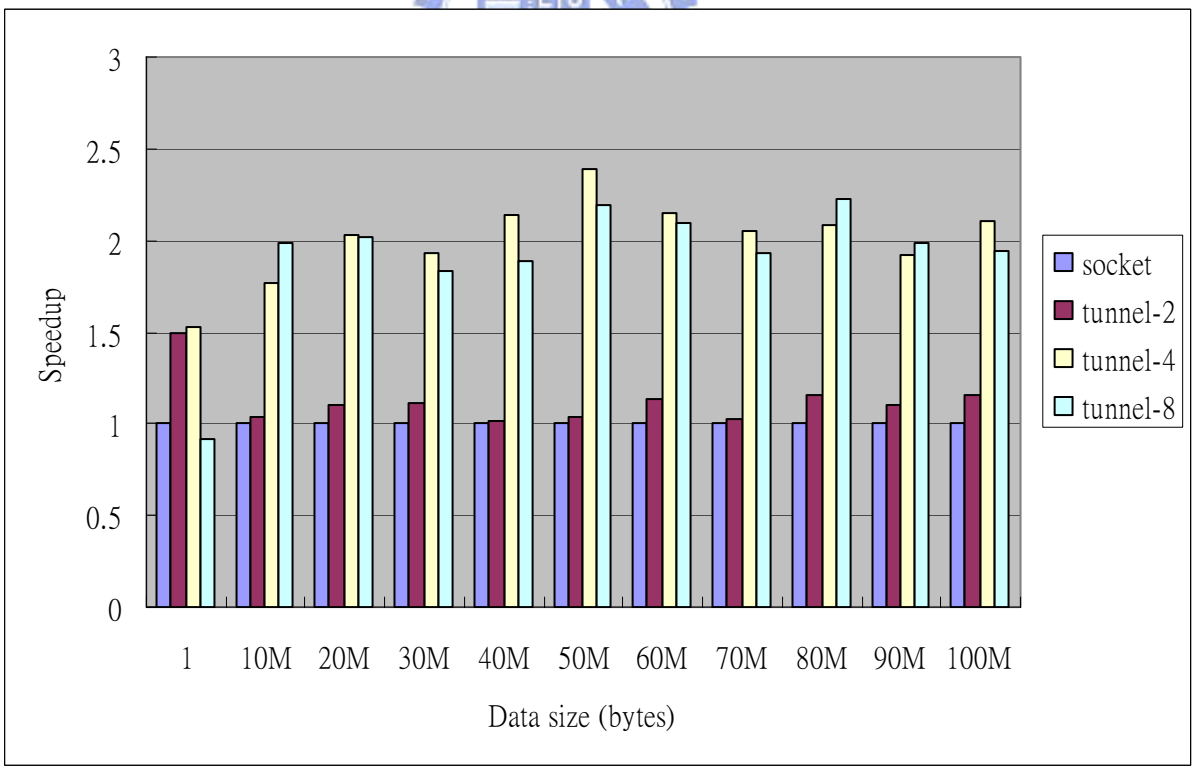


Figure 11 Max throughput (top) and its performance improvement (down) when data in the disk

4.4 Throughput when the Different Channel Sizes

Because *EVENT_SEND_DATA* is only sent when the channel is full, the large channel size can eliminate amount of events. Unfortunately, there is a side effect. Sender and receiver more possibly block for waiting the free channel and data. The result of the effect of different channel sizes is shown in Figure 12. And we keep the channel number is four in this experiment. As expectation, the throughput does not increase when the channel size is increased, in fact, best performance is emergence when channel size is 4K byte.

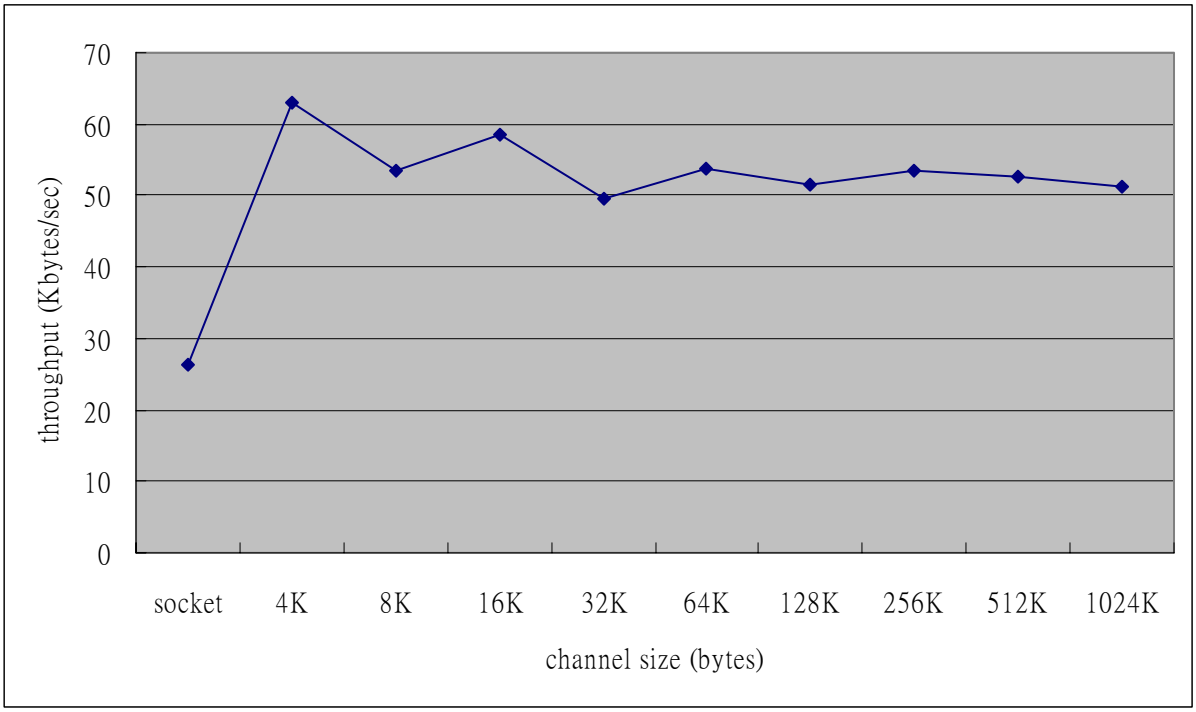


Figure 12 Throughput when different channel size

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we propose a mechanism called fast-IVC to improve the performance of local inter-virtual machine communication. It automatically switches the TCP/IP protocol processing to a shared-memory based protocol when the end points of a connection are on top of the same virtual machine monitor.

We implement fast-IVC on a para-virtualized machine environment, Xen. The experimental results show that the performance improvement ranges from 50% to 150%. And the creation overhead is small enough to neglect.

5.2 Future Work

Currently, fast-IVC only supports TCP. We plan to support UDP in the future. Generally, supporting UDP may lead to a larger overhead. This is because IP addresses and port numbers can be different in each UDP send/receive operation so that the destination address has to be checked every send/receive operation. As mentioned in Section, the checking involves domain-VMM mode switches.

REFERENCE

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt and A. Warfeld., “Xen and the Art of Virtualization”, In Proceedings of the ACM Symposium on Operating Systems Principles, Oct. 2003.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy., “Lightweight Remote Procedure Call”. In Proceedings of the 12th ACM Symposium on Operating System Principles, vol. 23(5), pp. 102-113, Dec. 1989.

[3] A. D. Birrell and B. J. Nelson, “Implementing Remote Procedure Calls”, ACM Transactions on Computer Systems, vol. 2(1), pp. 39-59, Feb. 1984.

[4] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System”, IBM J. Research and Development, vol. 25(5), pp. 483-490, Sep. 1981.

[5] R. P. Goldberg, “Survey of Virtual Machine Research”, IEEE Computer Magazine, vol. 7(6), pp. 34-45, June 1974.

[6] P. H. Gum, “System/370 Extended Architecture: Facilities for Virtual Machines”, IBM Journal of Research and Development, vol. 27(6), pp. 530-544, Nov. 1983.

[7] CX. Guo and SR. Zheng , "Analysis and Evaluation of the TCP/IP Protocol Stack of Linux", International Conference on Communication Technology Proceedings, vol. 1, pp. 444 -453, Aug. 2000.

[8] N. Harris, F. Armingaud, M. Belardi, C. Hunt, M. Lima, W. Malchisky Jr., J. R. Ruibal and J. Taylor, Linux Handbook: A Guide to IBM Linux Solutions and Resources, IBM redbooks, SG24-7000-01, April 2004

[9] Y. Huang, C. Kintala, N. Kolettis and N.D. Fulton, “Software Rejuvenation: Analysis, Module and Applications”, Proceedings of the Symposium on Fault Tolerant Computing FTCS-25, pp. 381-390, June 1995

[10] IBM, z/VM CP Commands and Utilities Reference, IBM redbooks, SC24-6081-01, Dec. 2004.

[11] IBM, z/VM TCP/IP Programmer's Reference, IBM redbooks, SC24-6021-02, Aug. 2003.

[12] D. B. Johnson and W. Zwaenepoel, “The Peregrine High-performance RPC system”, Software - Practice and Experience, vol. 23(2), pp. 201-221, Feb. 1993.

[13] M. B. Jones and R. F. Rashid, “Mach and matchmaker: kernel and language support for object oriented distributed systems”, In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 67-77, Oct. 1986.

[14] PH. Kamp and Robert N. M. Watson. “Jails: Confining the Omnipotent Root”, In Proceedings of the International SANE Conference, 2000.

[15] S. T. King, G. W. Dunlap and P. M. Chen, “Operating System Support for Virtual Machines”, 2003 USENIX Annual Technical Conference, pp. 71-84, June 2003.

[16] R. R. March, “Survey of System Virtualization Techniques”, March 2004.

[17] S. Nagar, H. Franke, J. Choi, C. Seetharaman, S. Kaplan, N. Singhvi, V. Kashyap and M. Kravetz, “CKRM: Class-based Prioritized Resource Control in Linux”, In Proceedings of the Ottawa Linux Symposium, July 2003.

[18] J. S. Robin and C. E. Irvine, “Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor”, Proceedings of the USENIX Security Symposium, Aug. 2000.

[19] J. E. Smith and R. Nair, “An Overview of Virtual Machine Architectures”, Nov. 2004.

[20] Sun, Sun Enterprise. 10000 : InterDomain Networks User Guide, Sun document, Feb. 2000

[21] W. Torres-Pomales, “Software Fault Tolerance: A Tutorial”, Langley Research Center, NASA, Oct. 2000.

[22] S. Tzou, and D. P. Anderson, “A Performance Evaluation of the Dash essage-Passing System”, Tech. Rep. UCB/CSB 88/452, Cumputer Division, University of California,

Berkeley, Oct. 1988.

[23] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server”, Proceedings of the Symposium on Operating Systems Design and Implementation, Dec. 2002.

[24] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble, “Constructing Services with Interposable Virtual Hardware”, In Proceedings of the 1st Symposium on Networked Systems Design and Implementation, pp. 169-182, March 2004.

[25] A. Whitaker, M. Shaw, and S. D. Gribble. “Scale and performance in the Denali isolation kernel”, In Proceedings of Symposium on Operating Systems Design and Implementation, Dec. 2002.

[26] A. Whitaker, M. Shaw and S. D. Gribble, “Denali: Lightweight Virtual Machines for Distributed and Networked Applications”, In Proceedings of the USENIX Annual Technical Conference, June 2002.

[27] B. White, R. Ayyar and V. Uskokovic, zSeries HiperSockets, IBM readbooks, SG24-6816-00, May 2002.

[28] B. White, J Nesbitt, F Packheiser and E. Palacio, IBM eserver zSeries: Connectivity Handbook book, IBM redbooks, SG24-5444-04, Jan. 2005.

[29] S. Williams, Networking Overview for Linux on zSeries, REDP-3901-00, IBM redbooks, Dec. 2004.

[30] T. V. Vleck, “The IBM 360/67 and CP/CMS”, http://www.multicians.org/thvv/360-67.html

[31] Ensim, http://www.ensim.com/index.html

[32] User-Mode Linux, http://user-mode-linux.sourceforge.net/

[33] VirtualPC, http://www.microsoft.com/windows/virtualpc/default.mspx

[34] VMWare, http://www.vmware.com/

[35] VServer, http://linux-vserver.org/

[36] z/VM, http://www.vm.ibm.com/