Tampere University of Technology

Department of Communication Engineering

Pasa Maharjan

Comparing and Measuring Network Event Dispatch Mechanisms in Virtual Hosts

Master of Science Thesis

Subject and examiners approved by the
Faculty of Computing and Electrical
Engineering Council on 08.06.2011
Examiners: MSc Bilhanan Silverajan
Prof. Jarmo Harju

Abstract

Tampere University of Technology
Degree Program in Information Technology, Department of Communication Engineering
Pasa Maharjan: Comparing and Measuring Network Event Dispatch Mechanisms in Virtual Hosts
Master of Science Thesis, 50 pages.
Examiners: Prof. Jarmo Harju, MSc Bilhanan Silverajan
Keywords: Virtualization, Event Dispatch Mechanism

Virtualization technology provides the foundation for building and managing reliable virtualized IT infrastructure by abstracting processors, memory, storage and networking resources into multiple virtual machines. As a result, modern servers are rarely deployed over hardware dedicated to a single software environment. The single physical server is split into number of virtual web servers, where each virtual server is completely isolated from each other and has its own operating system. Event dispatch mechanisms such as select and poll are common approaches that are often implemented on servers to retrieve events from file descriptors. However, there are various other high performance event mechanisms, such as epoll for Linux, kqueue of FreeBSD and event ports for Solaris 10, available. The research conducted in this thesis focuses on measuring and comparing the performance of network event dispatch mechanisms, for TCP and UDP traffic, deployed by different virtual hosts under the same physical hardware specification.

From the web server benchmark result, we observed that in the absence of idle connections, the event mechanisms select and poll performed comparatively well as regards to the high performance event mechanisms (epoll, kqueue,/dev/poll) in all the platforms. This is due to the fact that in this experiment the number of socket descriptors tracked by each event mechanism is not very high. However, the performance of select and poll degrades rapidly in all the system as the number of idle connections is increased. The results obtained from the UDP server benchmark show the similar pattern, as the number of ports opened increased, the response time for select and poll increased rapidly in all the platforms.

Preface

This Master of Science Thesis 'Comparing and Measuring Network Event Dispatch Mechanism in Virtual Hosts.' was carried out in the Institute of Communication Engineering at Tampere University of Technology, Finland.

I am sincerely grateful to my supervisors Bilhanan Silverajan and Prof. Jarmo Harju for their valuable guidance, excellent support and tremendous encouragement throughout my research. Their insight and motivation were invaluable.

I would also like to thank Matti Tiainen for his help during my studies, research and thesis. I am grateful towards my parents for their love and inspiration. Finally I would like to express my greatest thanks to my friend Jenny Maria Molin for her love and tireless encouragement during this work.

Tampere, May 22 , 2012

Pasa Maharjan

Tampere, 33720
Finland
pasa.maharjan@tut.fi

Contents

List of abbreviations

VMM	Virtual Machine Monitor
VM	Virtual Machine
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
CPU	Central Processing Unit
NIC	Network Interface Card
VMM	Virtual Machine Monitor
OS	Operating system
API	Application Programming Interface
HAV	Hardware Assisted Virtualization
KVM	Kernel-based Virtual Machine
BIOS	Basic Input-Output System
OS	Operating system
HTML	Hypertext Markup Language

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing size of the Internet and complexity of modern computing systems, today there is a necessity of large scale systems with significant system management effort that includes maintenance, reconfiguration, fault tolerance and administration. To fulfill this effort, recent virtualization technology emphasizes ease of system management and administration.

Virtualization technology addresses the needs of massive data center environment by providing efficient, secure and flexible system infrastructure to meet the demanding resources requirement of modern computing systems. This technology enables consolidation of multiple virtual systems on a single physical server to improve the scalability and resource utilization. The introduction of Virtual Machine Monitor (VMM) or hypervisor, which runs directly on server hardware, allows running multiple guest virtual machines. Each guest virtual machine is capable of running a different operating system instance.

Hypervisors like VMware ESX [2] and VMware ESXi [2] provide the foundation for building and managing reliable virtualized IT infrastructure by abstracting processors, memory, storage and networking resources into multiple virtual machines. These systems not only reduce the hardware cost by running multiple operating systems on single server but also lower the management overhead, providing high levels of performance for the most resource intensive applications [2]. With the available products for virtualization in the market, modern servers are rarely deployed over hardware dedicated to a single software environment. The single physical server is split into a number of virtual web servers, where each virtual server is completely isolated from each other and has its own operating system.

Virtual Web servers are a very popular way of providing low cost web hosting services

these days. However, the increasing size of Internet and number of connected users has made these servers process and handle a large volume of network traffic. Today, network traffic is predominantly based on web and video traffic. For web traffic, data is carried via HTTP over TCP, while most realtime video traffic is carried via RTP and RTSP over UDP. As services increase in popularity, the servers which host the services must scale upwards to meet the increased demand. To achieve the scalability, an event driven approach is often implemented in the servers to multiplex a large number of concurrent connections. Network event dispatch behavior is a crucial component which directly affects the response and latency times of network level requests and responses from a server.

Event dispatch mechanisms such as select and poll are common approaches that are often implemented on servers to retrieve events from file descriptors. However, these mechanisms are not scalable enough when the server is overloaded with a large set of descriptors. The scalability issues and limitations of select and poll have been pointed out by Banga et al. [3]. Various other scalable event dispatch mechanism such as epoll, kqueue and /dev/poll/ are developed and implemented in different platforms like Linux, FreeBSD and Solaris. All these event mechanism are designed to overcome the limitation of traditional select and poll and improve scalability of the network servers.

1.1 Thesis objectives

This thesis will investigate this new trend: How virtualised instances of hosts can scale to meet traffic demands for popular Internet services. Several of the most popular server platforms, such as Linux, FreeBSD, Solaris and Windows would be deployed atop a commercial enterpriselevel virtualisation server. These servers would then be subjected to a variety of workloads to study the performance of the various network event dispatch mechanisms which have been implemented in these operating systems. By using a common hardware baseline for all the systems under measurement, their relative performances for UDP and TCP traffic can be evaluated to understand which platforms are suitable for what kinds of traffic patterns.

1.2 Thesis contributions

The contributions of this thesis are summarized as follows:

- introduction of virtualization technology and different network event dispatch mechanism,

- measurement setup for event dispatch mechanism in different server platforms for TCP and UDP traffic,
- comparison and performance analysis with TCP and UDP traffic.

1.3 Thesis outline

The thesis consists of six Chapters. The structure of thesis follows with brief overview of virtualization and virtualization techniques in Chapter 2. Chapter 3 familiarizes with I/O multiplexing and network event dispatch mechanism designed for different platforms. The discussion of research methodology and measurement setup is detailed in Chapter 4. Chapter 5 presents the comparison result of event dispatch mechanisms for different virtual hosts. Finally the conclusions from this research are presented in Chapter 6.

Chapter 2

Theoretical Background

2.1 Virtualization overview

New concepts and technologies continue to emerge, develop, and mature at a rapid pace. At the same time, the widespread use of computer servers and Internet has given inevitable rise to resource management complexities and security hazards. Virtualization technology emerges to address these issues. At its simplest level, the term virtualization may refer as to partitioning of resources of single system into multiple execution environments or ability of a computer to run multiple guest operating systems and applications on one piece of hardware in such a way that these environments are completely isolated from one another. For example, a single server can be partitioned into multiple operating systems so that each instance can be dedicated to one customer.

In slightly more technical terms, virtualization essentially introduces a level of indirection to a system that decouples users, operating systems, and applications from the specific hardware characteristics of the underlying host system [4; 5]. The technology promises important properties such as isolation and mobility, providing numerous useful benefits [5]. Some of the significant benefits are listed below:

- consolidate workloads to reduce hardware, power and space requirements,
- run multiple operating systems simultaneously,
- run legacy software on newer, more reliable, and more power efficient hardware,
- dynamically migrate workloads to provide fault tolerance,
- provide redundancy to support disaster recovery,
- ease of Testing and Development, as designers can compare application performance across different operating environments.

Virtualization can be performed at a number of levels of abstraction. However the main two methods of providing virtualization are hardware virtualization and operating system (OS) virtualization. Hardware virtualization method use the hardware abstraction layer called Virtual Machine Monitor (VMM) [6] which resides on virtualization layer, on top of the actual hardware. This layer decouples the OS from the hardware so that an entire OS environment and associated applications can be executed in a virtualized environment [4]. Example of hardware level virtualization technologies are VMware, Virtual PC and Xen. OS-level virtualization partitions the operating system to create multiple isolated virtual machines (VM). These virtual machines provide a virtual execution environment that can be instantly forked from the base operating environment [4]. Figure 2.1 shows the virtualization layers of hardware-level virtualization and OS-level virtualization.

Different levels of virtualization can differ in isolation strength, resource requirement, scalability and flexibility. In general, when the virtualization layer is closer to hardware, isolation of created VMs are much better from one another and better separated from the host machine. However, it offers less flexibility with more resources requirement.

2.2 Virtualization techniques

In the following, some of the central virtualization techniques and concepts are presented.

2.2.1 Virtual machine

Virtual Machine (VM) is a software implementation of an execution environment that enables multiple virtual environments on single physical machine. Each VM is isolated from each other and capable of running regular operating systems and applications as if it were a physical computer. A virtual machine behaves exactly like a physical computer and contains its own virtual CPU, hard disk and network interface card (NIC), providing users the illusion of accessing a real machine directly. VMware Server is an example of virtualization where the virtual machine approach is used to virtualize the whole underlying hardware layer. The concepts of virtual machine and virtual machine monitor are introduced by Popek and Goldberg in their article Formal requirements of Virtualizable Third Generation Architectures, published in 1974. The article [7] presents the three general requirements that a virtual machine should fulfill:

1. The efficiency property, which means that the majority of the guest instruction must be executed directly by hardware.

2. The resource control property, which means that the VMM acts as an intermediary, and it is therefore not possible for the guest system to affect the system resources without the VMM.

3. The equivalency property, which refers to that the software is executed in the same manner regardless of the execution environment.

The VMM is the software component that hosts guest virtual machines and abstracts the physical resources for use by each virtual machine. VMM can be implemented in different ways to achieve the virtualization. VMM can be run directly on real hardware, independent of any host operating system and could also run as an application on top of a host operating system. VMM running on top of host OS use the underlying host API to perform necessary task. Figure 2.1 shows an example where VMM is implemented on top of host OS. The VMM controls the individual virtual machines, shares and assigns the resources of the system. VMware server is an example of this kind of virtualization technique. The drawback of this approach may be loss of performance. Figure 2.2 shows an example where virtualization is implemented without a host OS. In this case, virtualization layer functions as OS and performance can be improved.

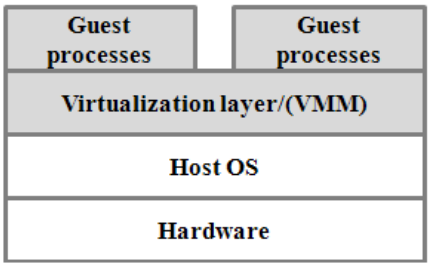


Figure 2.1: VMM is implemented on top of host OS

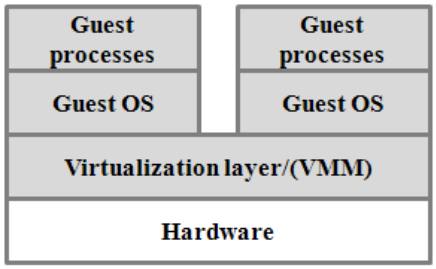


Figure 2.2: Implementation of virtualization without a host OS

2.2.2 Full virtualization

Full virtualization is one of the most common server virtualization technique that was designed to provide total abstraction of the underlying physical hardware and creates a complete virtual system in which unmodified guest operating systems and its applications can execute [8]. Each of the virtual systems are isolated from each other and managed by virtualization layer or hypervisor, which controls the flow of instructions between guest OS and physical hardware such as CPU, disk storage and memory. [9]

The unmodified guest operating system refers to capability of hypervisor that provides most of the same hardware interface as those provided by the actual hardwares physical platform. As a result, the guest OS or application is not aware of the virtualized environments and have capability to execute on VM just as they would on a physical system [8; 9]. This provides complete isolation of different applications and high security for virtual machines. Full virtualization supports dissimilar operating systems like Windows and Linux. Microsoft virtual server and VMware ESX Server are examples of full virtualization.

However one of the potential drawbacks of full virtualization lie on its performance as application often run somewhat slower on virtualized system as the computing power of a physical server and related resources is reserved for hypervisor that need data processing. [8; 9]

2.2.3 Paravirtualization

The paravirtualization refers to a virtualization technique where each VM is provided with an abstraction of the hardware that is similar but not identical to underlying physical hardware. Thus, paravirtualization technique requires modifications to the guest operating systems in order to work with the commands of the virtual machine [8]. As a result, the guest operating systems are aware that they are executing on VMs. This provides a number of benefits such as less complex virtualization layer and more opportunities for optimizing as OS is aware of its environment. However, one potential downside of this technology is that the modified guest operating system cannot be migrated back to run on physical hardware [10]. Xen [8] is the best known virtualization system implementing paravirtualization.

The potential drawback of paravirtualization is eliminated with the new virtualization technology from Intel and AMD allowing guest operating system to run without modification. The special privilege level for virtualization layer is introduced which is called root mode, makes possible to install unmodified guest OS.

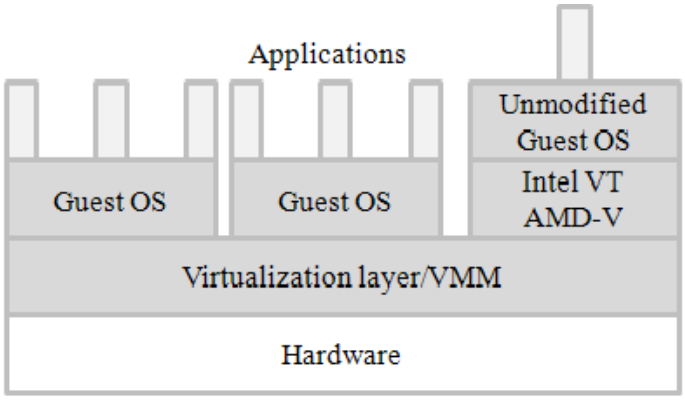


Figure 2.3: Example of paravirtualization

2.2.4 OS-level virtualization

OS-level virtualization, also referred as shared OS virtualization, refers to a virtualization technique that partitions the physical machines resources at the operating system level. Such partition at the operating system creates multiple isolated virtual machines (VM), also called containers, that are isolated from each other but sharing a common OS kernel. These virtual machines provide a virtual execution environment that can be instantly forked from the base operating environment [4].

The operating system level virtualization architecture where the entire containers are deployed on top of a single OS, introduces a whole new set of advantages. Such advantages include low overhead that helps to maximize efficient use of server resources that are available to the applications running in the containers. It is cost effective and convenient as patches or modifications can be made easily to the host server that could be instantly applied to all the containers. However, this approach typically limits the operating system choice as every guest operating system must be identical or similar to the host in terms of version number and patch level. Example of implementation of virtualization on OS level includes Linux VServer [11], OpenVZ [12], FreeBSD Jails [13].

2.3 Virtualization software

There are a great number of virtualization systems available today. Many of them are proptitery . Some of the popular virtualization software are discussed below.

2.3.1 Xen

Xen [14] is an external hypervisor, a layer of software running on computer hardware replacing the operating system. It supports x86, x86_64 and can run Linux, Windows, Solaris, and some of the BSDs as guests.There are three components required while using Xen for virtualization: Xen hypervisor, Domain 0(Dom0 or the privileged domain), and Domain Us (Dom U or an unprivileged domain). Dom0 runs on the hypervisor with direct access to hardware and system administrator manages the whole system by logging into Dom0. Dom Us runs the guest operating systems and have no direct access to hardware.

Xen also can be run in two modes: paravirtualization and full virtualization. Paravirtualization allows guests to run without special calls to the processor. Full virtualization, also known as Hardware Assisted Virtualization (HVM), which depends on processor virtualization technology (Intel-VT or AMD-V) offers even better performance and extended features.

2.3.2 KVM

KVM [15] (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). KVM consist of a loadable Linux kernel module that provides the core virtualization infrastructure and another processor specific hardware virtualization extension module.

A user space application called QEMU [16] is needed to create a virtual machine using KVM. QEMU, is a generic open source machine emulator and virtualizer, emulates a whole computer including various processors and devices. This emulation process allows running unmodified guest operating systems.

2.3.3 VMware ESX and VMwareESXi

VMware ESX and VMware ESXi [2]are the latest commercial hypervisor, based on bare-metal architecture, from VMware that runs directly on hardware. Both the hypervisors insert a robust virtualization layer between hardware and operating system offering partition of a physical server into multiple secure and portable virtual machines. These virtual machines, represents a complete system with BIOS, processors, memory and networking, are completely isolated from each other and run side by side on the same physical server. For each virtual machine the operating system and software applications can be installed without any modification.

The major difference between these two systems lies in the architecture and the operational management of VMware ESXi. VMware ESX does not run on top of another operating system, but rather sets up its own Linux kernel. This kernel is used to load specialized virtualization components (vmkernel), and becomes the first running virtual machine called service console. The service console also performs some management functions including executing scripts and installing third party agents for hardware monitoring. Whereas in VMware ESXi, this service console has been removed and provided with remote scripting environments such as vCLI and PowerCLI to allow the remote execution of scripts and commands. [2]

2.3.4 VirtualBox

VirtualBox is a free, open source, powerful cross-platform virtualization software for x86 based systems. It runs on Windows, Linux, and Mac OS and supports a large number of guest operating systems. VirtualBox requires an existing operating system installed and can run alongside with an existing application on that host. VirtualBox provides wizard-based interface that simplifies the creation and management of virtual machines.

One of the significant benefits of VirtualBox is that it doesnt require the processor features built into newer hardware like Intel VT-x or AMD V. Thus it also supports an older hardware where these feature are not present. The VirtualBox uses industry-standard Open Virtualization Format (OVF) to import and export of the virtual machines. OVF is a cross-platform standard supported by many virtualization products which allows for creating readymade virtual machines that can be imported into a virtualizer such as VirtualBox. [17]

Chapter 3

Network event dispatching

The daily advancement of the technologies, applications and substantial use of Internet has compelled us to think about the power of computation and scalability of network server. Scalability of network server generally refers to the ability to process large amount of network events simultaneously without degrading the performance. To achieve this scalability, an event driven approach is often implemented in the servers to multiplex a large number of concurrent connections over a few server processes without blocking its main process [18].

3.1 I/O models

The network I/O models describe the technique to accomplish I/O task such as connecting to a server, writing data to disk and receiving data from network. The scalable I/O operation can be accomplished in either synchronous or asynchronous way.

The synchronous I/O operation causes the requesting process to be blocked until I/O operating completes. In other words, the operation sender must wait until the hardware has completed the physical I/O, so that it can be informed of the success or failure of the operation. [19] Examples of such I/O operation are blocking I/O, non-blocking I/O.

Asynchronous I/O event handling basically refers to the ability of a process to perform input/output on multiple sources at one time and permits other processing to continue before the transmission has finished. The five I/O models available under Unix system are presented here.

3.1.1 Blocking I/O model

Blocking I/O is the simple model and used in first socket implementations. In this model the entire API calls return only when the requested operation completed. Figure 3.1 shows the example of blocking I/O model where process calls read and system call doesn't return until the data is ready. The process is set to sleep mode until the data is ready to copy, leaving system resource idle.

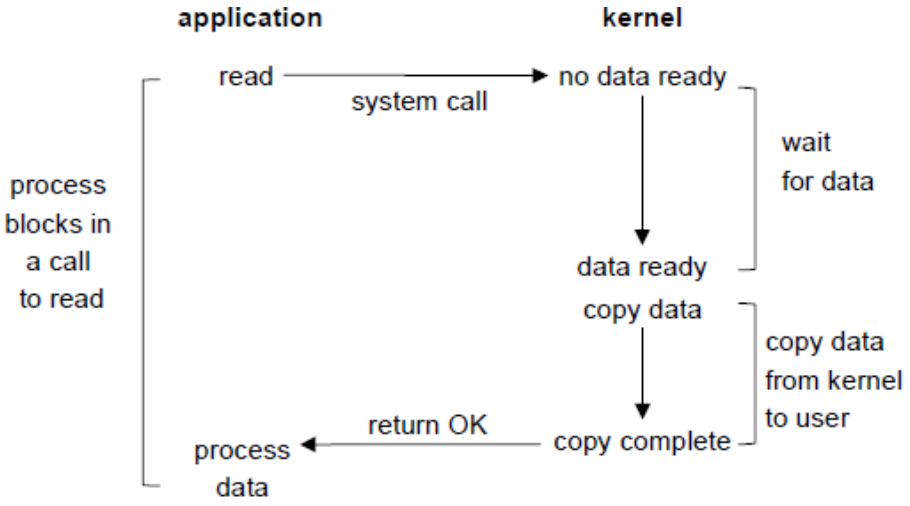


Figure 3.1: Blocking I/O model [1].

By default, a socket is in blocking mode and behaves as shown above.

3.1.2 Non-blocking I/O model

In non-blocking I/O mode the process gets never blocked or in sleep mode instead kernel sends the error message when a requested I/O operation cannot be completed. Figure 3.2 shows and example of non-blocking I/O model where the kernel immediately returns an error message EWOULDBLOCK application when there is no data ready. The model uses polling method to repeatedly call read and receives error message until the data is ready without putting any process to sleep.

3.1.3 I/O multiplexing model

In I/O multiplexing model, instead of blocking in the actual I/O system call, it uses UNIX system calls such as select () or poll () and block in one of these system calls. Figure 3.3 shows example of I/O multiplexing Model using select system call. The select () function

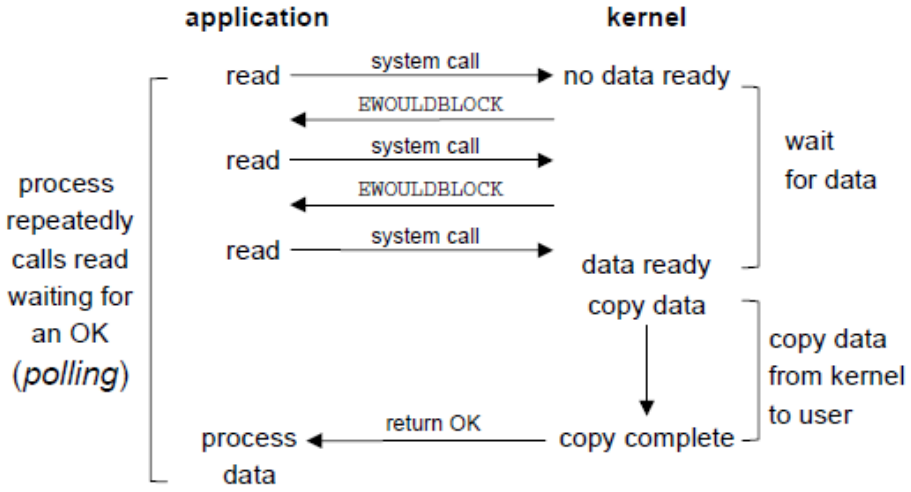


Figure 3.2: Non-blocking I/O model [1].

blocks and waits until the data is readable. Then the read function is called to copy the data

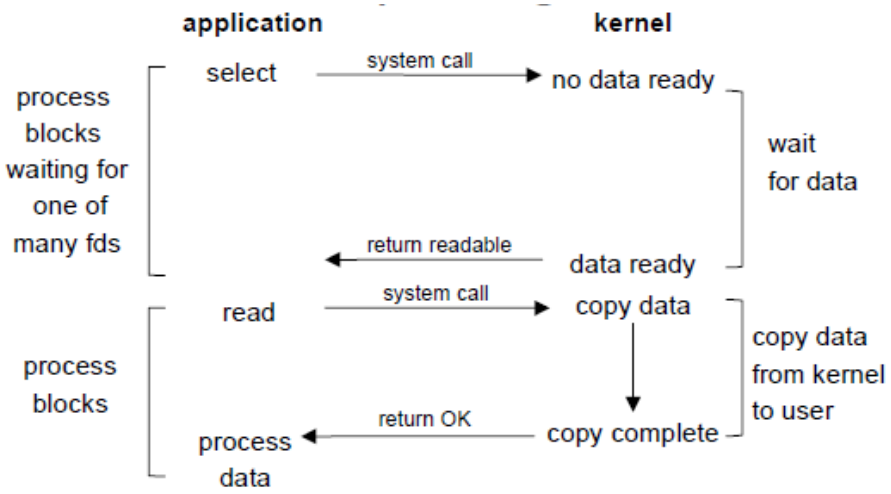


Figure 3.3: I/O Multiplexing model [1].

In this case select blocks and acts as blocking socket, however select is more powerful than blocking sockets because it can wait on multiple events.

3.1.4 Signal I/O model

Signal-driven I/O mode enables the kernel to notify the process with a (SIGIO) signal when a descriptor is ready. To read the data from the socket the process needs to establish a

signal handler for the (SIGIO) signal which is done by sigaction system call. The process is not blocked and kernel interrupts or notifies the process when data is ready to be processed. Figure 3.4 shows the example of signal-driven I/O model.

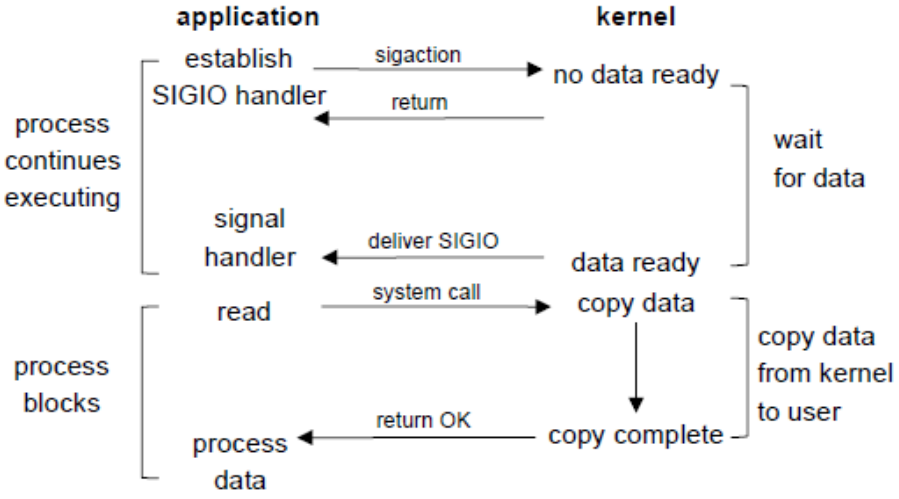


Figure 3.4: Signal-driven I/O model [1].

3.1.5 Asynchronous I/O model

In asynchronous model, the process uses asynchronous aio_read() or aio_write () system calls for I/O request and returns immediately once the I/O request has been passed down to the hardware or queued in operating system. The process is not blocked and continues executing. The results of I/O operation can be received later when they are available. Figure 3.5 shows the example of asynchronous I/O model.

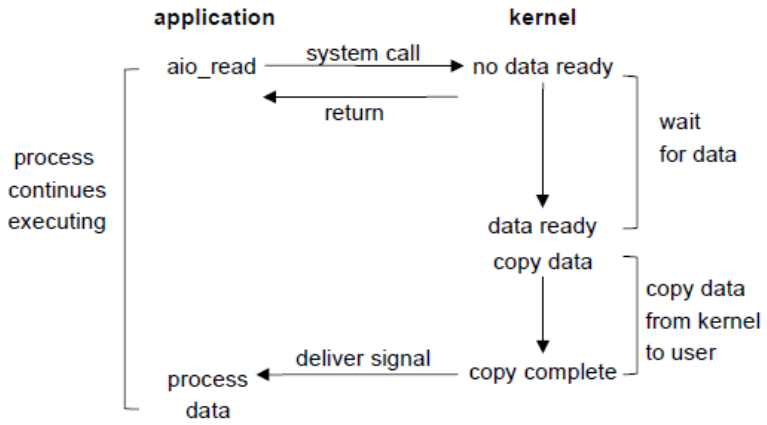


Figure 3.5: Asynchronous I/O model [1].

3.2 Network I/O management architecture

3.2.1 Multi-Process/Multi-Thread

A multi-process/multi-thread is a simple architecture where server allocates a process to each connection. Each connection in the server is treated by single service thread and I/O multiplexing is performed by process/thread switching. These service threads can be implemented in two ways:

- On-demand: Each service thread is forked whenever a new connection is accepted and handles the request for the connection [20]. Under the high load when there are large numbers of new connections being established, this technique can lead to large forking overhead.
- Pre-forked: The server could have a pool of pre-forked service threads. When the master thread receives a new connection, it can select one thread from the pool and hand over the connection. This method prevents the forking overhead however; require high memory usage even under low loads. [20]

3.2.2 Event-based

In event base, the server employs event dispatch mechanism provided by the underlying operating system to perform network I/O [20]. A single thread of execution uses non blocking I/O to multiplex its service across multiple connections. The OS uses some form of event notification to inform application when one or more connections require service. These mechanisms are highly scalable as the processing cost of each event does not depend on the number of the concurrent connection. [20]

There are several system interfaces implemented for event driven servers and moreover, many operating systems also have their own interfaces for event driven communication processing such as, epoll in Linux, kqueue in BSD variants, event ports, /dev/poll/ or poll device files in Solaris.

3.3 Edge triggered and level triggered notification schemes

In edge triggered, a file descriptor is returned as being available for I/O after a change has happened on that file descriptor. In other words, we get a single notification when the state changes and then nothing more until another state change. System call such as epoll and kqueue use edge triggered I/O notification scheme.

Unlike edge triggered notification scheme, in level triggered a file descriptor is returned if it is possible to read or write now which means that we get notification whenever the event is present (which will be true over a period of time). System call such as select () and poll() use level triggered I/O notification scheme.

3.4 Network event multiplexing

3.4.1 select () system call:

The select () is a powerful function that enables an application to multiplex I/O. The I/O multiplexing is achieved by allowing a single thread or process to multiplex its time between a numbers of concurrently open socket connections [20]. The application calls the select() system call by providing the set of interested file descriptors to watch and the kernel reports back to the program which file descriptor's state have changed. The file descriptor's state is identified as readable without blocking, writeable without blocking or exception pending [20]. If any file descriptor in any of the sets is ready for its given condition, select () returns the number of ready file descriptors and modifies the sets to indicate which file descriptors are available. If none are available, select() blocks (sleep) for a specified period of time waiting for any of the file descriptors to become ready. The system call is declared as:

```
int select(
                int nfds,
                fd_set *readfds,
                fd_set *writefds,
                fd_set *exceptfds,
                struct timeval *timeout);
```

The parameter readfds, writefds and exceptfs identifies the sockets that are to be checked for readability, writeability and any exceptional error conditions.

The parameter timeout controls the behavior, how long the select () can wait for an event. If the timeout is set NULL, selects wait indefinitely for an event. If the value of timetable structure are set to 0 the function returns immediately without waiting for any events.

3.4.2 poll () system call:

A poll is also powerful tool for network I/O multiplexing, which attempts to consolidate the arguments of select () and provides notification of a wide range of events. A poll system

call has basically similar functionality as select but it uses a slightly different interface. Poll uses an array of pollfd structures to describe its interest set. The kernel then returns the set of ready descriptors also as a list of pollfd structures. [20]

```
struct pollfd {
                        int fd;
                        short events;
                        short revents;
                    }
```

```
int poll ( struct pollfd *ufds,
  unsigned int nfds;
int timeout ) ;
```

The member of the structure include *fd* indicates which fd to monitor for an event. *events* and *revents* represents which events will be monitored and which events were detected in a call to poll().

The parameter ufds and nfds in a function holds array of interested pollfd and the number of file descriptors set in fds respectively. Similarly, the parameter timeout controls the behavior, how long the poll () can wait for an event and the return value indicates how many fds had an event occur.

3.4.3 epoll () system call:

Epoll is a efficient and highly scalable I/O event notification mechanism for Linux that was introduced in Linux 2.5.44. It is designed to overcome the inefficient scalability of select and poll system calls over large number file descriptors. Unlike traditional system calls epoll supports both edge-triggered and level triggered fashion. One of the significant drawback of select and poll is that they are dependent on the size of the interest set than number of events returned. For example select () monitor up to FD_SETSIZE number of descriptor, poll () doesn't have a fixed limit of descriptors; however they perform a linear scan of all the passed descriptors every time to check readiness notification causing O(n) performance. The epoll avoids such fixed limits and linear scans but uses callbacks in the kernel file structure improving O(1) performance. [18]

The epoll instances are created and managed by following system calls.

epoll_create(2) creates and return epoll instance.

```
\#include <sys/epoll.h>

int epoll_create(int size)
```

epoll_ctl(2) control interface for adding or removing the interested file descriptors that needs to be monitored my epoll instance.

```
       int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

epoll_wait(2) is used to wait for events on the watched set,

```
int epoll\_wait(int epfd, struct epoll_event *events,
                int maxevents, int timeout);
```

The epoll event structure is defined as

```
typedef union epoll_data {
                void        *ptr;
                int          fd;
               uint32_t     u32;
               uint64_t     u64;
           } epoll_data_t;


 struct epoll_event {
                uint32_t     events;      /* Epoll events */
                epoll_data_t data;        /* User data variable */
           };
```

3.4.4 /dev/poll/ system call:

/dev/poll is a state-based event dispatching mechanism that was first introduced in Solaris 7. /dev/poll was designed as improved on polls performance by removing the need to specify interesting set on every poll(). The idea behind /dev/poll is that the application can open the device file to build a set of descriptors of interest inside kernel. This device file allows a process to monitor multiple sets of polled file descriptors which can be accessed

through open(), write() and ioct1() system call [21]. This technique reduces the amount of interesting set information copied between user space and kernel space.

Another main benefit is the ability to handle dead connection by avoiding the linear scan of function for events. it uses callbacks for notifying the driver code. /dev/poll mechanism further reduced data copying by using a shared memory region to return events to the application [18].

3.4.5 kqueue API

kqueue is a event notification API introduced in FreeBSD 4.1. Kqueue provides a generic method of notifying the user when an event happens or a condition holds, based on the result of small pieces of kernel code termed filters [22]. Kqueue supports both edge-triggered and level triggered notification scheme providing efficient scalability over large number of descriptors.

The kqueue API uses two system calls kqueue() and kevent(). The system call kqueue() creates a new notification channel or queue where application registers the number of interested events. The returned value from kqueue() is treated as an ordinary descriptor and can be passed to other system calls such as poll() and select() [23]. The system call kevent() is used by applications to register events with the queue and also retrieve any pending events to the user. Any changes that should be applied to the kqueue are mentioned in changelist, which is a pointer to an array of kevent structures. Any events returned are places in the eventlist and the maximum size of the eventlist is determined by nevents. The timeout specifies a maximum interval to wait for an event.
The application registers the events to the system with struct kevent where event is uniquely identified by a tuple kq, ident, filter . The detailed description of fields of struct kevent and flag value can be found in [22; 23], however some are presented in Table 3.1.

3.4.6 Event completion framework

Solaris 10 introduced the Event Completion Framework (ECF), similar to FreeBSD (kqueue), to solve the problems of traditional select and poll not being able to scale efficiently on large number of file descriptors. Event ports or ECF is a powerful concept to deal with events from various sources, including file descriptors, asynchronous I/O, other user processes in a scalable and efficient manner. The fundamental concept of event completion framework is creating a port. An application uses these ports to register the events of interest and reap events on the object of interest using a single interface. The general use example of event completion framework is presented below.

Table 3.1: Flag values for kevent

ident	Value used to identify this event. The exact interpretation is determined by the attached filter, but often is a file descriptor.
filter	Identifies the kernel filter used to process this event. The pre-defined system filters are described below.
flags	Actions to perform on the event.
fflags	Filter-specific flags.
data	Filter-specific data value.
udata	Opaque user-defined value passed through the kernel unchanged.

```
/* Create port to use for event completion */
int portfd = port_create();
...
/* Register, or associate, the objects and events you are
   interested in */
port_associate(portfd,  ... );
...
/* Block until a single event appears on the port */
port_get(portfd,  ... );
```

The framework provides port_create() function to create port for event completion, returning a non-negative integer representing the ports identifier. The port_associate() function associates an object such as file, socket and timer with previously created port. The first parameter is port identifier returned by port_create() and the second parameter associates a list of objects that will be monitored by the port. The function port_get() reaps the completed events from the port.

Chapter 4

Experiment methodology

In this chapter the test environment will be introduced. The test network, hardware and software configuration used for conducting measurement is introduced in Section 4.1. Section 4.2 describes the environment for implementing web server and UDP server. The implementation of web server and UDP server is discussed in Section 4.3 . The workload generators for web server and UDP server are detailed in Section 4.4. Finally the procedure for benchmarking a web server along with some benchmarking tools are discussed in Section 4.5.

4.1 Experimental environment

4.1.1 Test Network

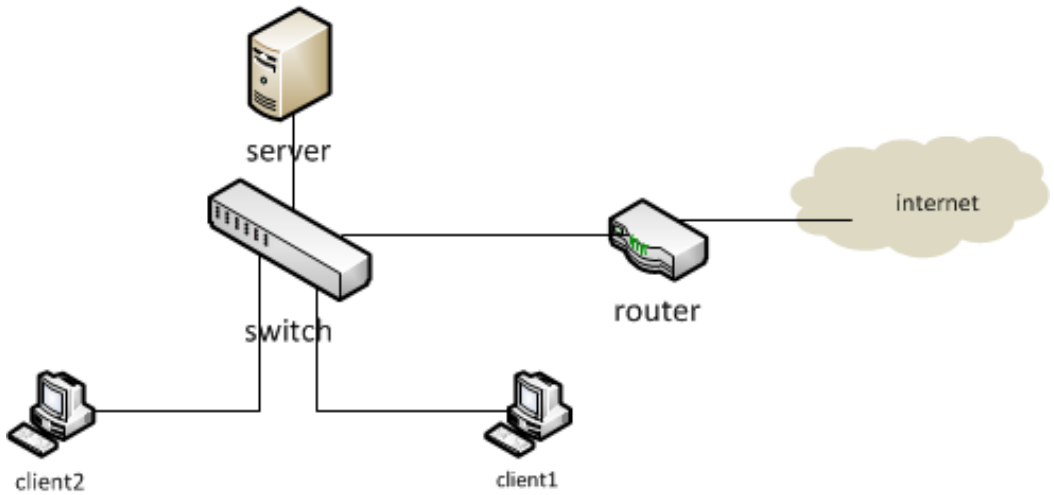


Figure 4.1: Physical diagram of test network.

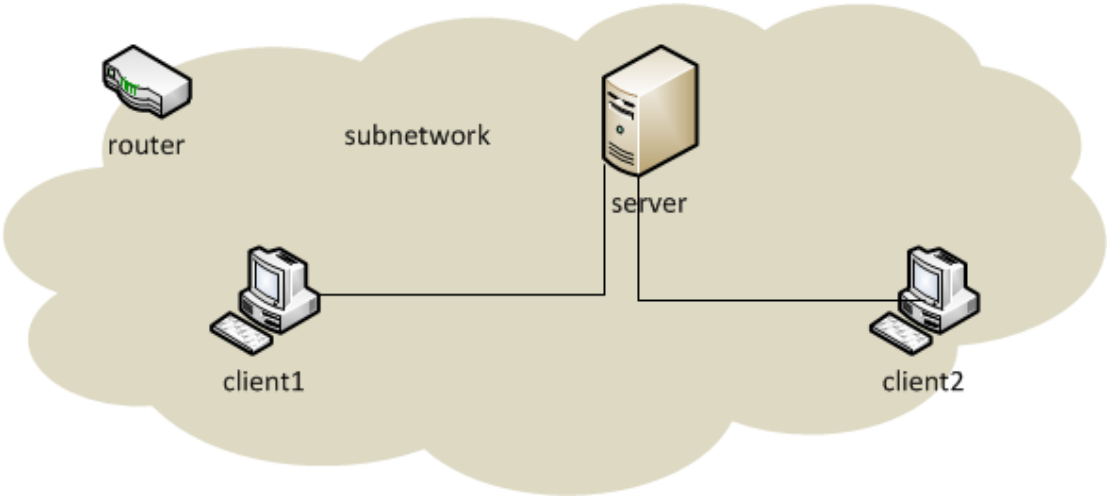


Figure 4.2: Logical diagram of test network.

4.1.2 Hardware Configuration

The test server consists of four different virtual servers installed on VMware ESX server. The physical hardware and the server configurations are presented in Tables 4.1 and 4.2. We used two client machines for event based web server and one client to communicate with UDP server. The clients contain Intel Pentium(R) 4 CPU 2.00 GHz processor, 1 GB of RAM and use 32-bit Linux operating system (Ubuntu 10.10). The operating system was updated whenever new updates were available.

Table 4.1: Physical Hardware setup.

General Manufacturer	IBM
Mode	BladeCenter HS22 -[7870L2G]-
Processors	8 CPU x 2,266 GHz
Processor Type	Intel(R) Xeon(R) CPU L5520 @ 2.27GHz
Hyperthreading	Active
Total Memory	47,99 GB
Number of NICs	7
State	Connected
Virtual Machines	8
vMotion Enabled	Yes
Active Tasks	No

Table 4.2: Server specifications for all the given virtual hosts.

Servers	Specification
Linux(bill-linux.rd.tut.fi)	- one Intel Xeon 2,27 GHz CPU L5520 - 1024 MB of physical memory (RAM) -one Gigabit network interface full-duplex - Ubuntu 10.04.1 LTS 64-bit -kernel 2.6.32-28
FreeBsd(bill-freebsd.rd.tut.fi)	- one Intel Xeon 2,27 GHz CPU L5520 - 1024 MB of physical memory (RAM) - one Gigabit network interface full-duplex - FreeBSD 8.1-RELEASE (64-bit)
Solaris(bill-osol.rd.tut.fi)	- one Intel Xeon 2,27 GHz CPU L5520 - 1024 MB of physical memory (RAM) - one Gigabit network interface full-duplex - OpenSolaris 2009.06 / SunOs 5.11
Windows7(bill-win7.rd.tut.fi)	- one Intel Xeon 2,27 GHz CPU L5520 - 2048 MB of physical memory (RAM) - one Gigabit network interface full-duplex - Microsoft Windows 7 (64-bit)

4.2 Development environment

4.2.1 Socket

A fundamental technology for programming software to communicate in IP networks are sockets, which provides a bidirectional communication endpoint for sending and receiving data with another socket. Sockets are the Application Programming Interface for Transmis-

sion Control Protocol (TCP) and User Datagram Protocol (UDP). The importance of sockets can be realized as many of popular network software packages including web browsers, instant, messaging application and P2P file sharing system rely on sockets.

In general, socket represents a single connection between client and server which is identified by a socket descriptor that can be used like a regular file descriptor on a UNIX system. A socket descriptor is basically an integer that identifies a socket when making a system call. A network programmer accesses these sockets using the socket API that already exist such as Berkeley Socket Library for UNIX system and Windows Socket(Winsock) for Microsoft operating systems.

The most commonly used socket types are stream socket and datagram socket. One of the significant differences between these two types is that stream socket implements connection oriented semantics where as datagram socket implements connectionless semantics. Connection oriented sockets means that the communicating parties first need to carry out a TCP handshake before any data transfer takes place. However, connectionless socket doesnt need any prior connections. Either party can simply send datagrams as needed and wait for the responses.

TCP uses stream socket type and UDP uses datagram socket types to communicate over the internet. To identify specific computers, IP socket libraries use the IP address and the ports. The need for ports is obvious as IP address only identifies the network interface of a computer. The port number distinguishes multiple applications from each other. For example web browser uses port 80 as a default for socket communications with web servers. Another powerful socket type is UNIX domain sockets (AF_UNIX), which are used to communicate between processes on the same machine efficiently. UNIX domain sockets are named with UNIX paths. For example, a socket might be named /tmp/test.

4.2.2 HTTP/TCP

HTTP (Hypertext Transfer Protocol) is a communications protocol that facilitates the transfer of information on the Internet. It is a request-response protocol between clients and servers. A HTTP transaction consists of three steps: TCP connection setup, HTTP layer processing and network processing. The TCP connection setup is performed through a three way handshake, where the client and the server exchange TCP SYN, TCP SYN/ACK and TCP ACK messages. Once the connection has been established, the client sends a request for an object which can be for example static Hypertext Markup Language (HTML) files, image files or various script files. The server handles the request and

returns the object or the results of these queries. Finally, the TCP connection is closed by sending TCP FIN and TCP ACK messages in both directions. Figure 4.3 shows TCP Three-Way Handshake Connection Establishment Procedure.

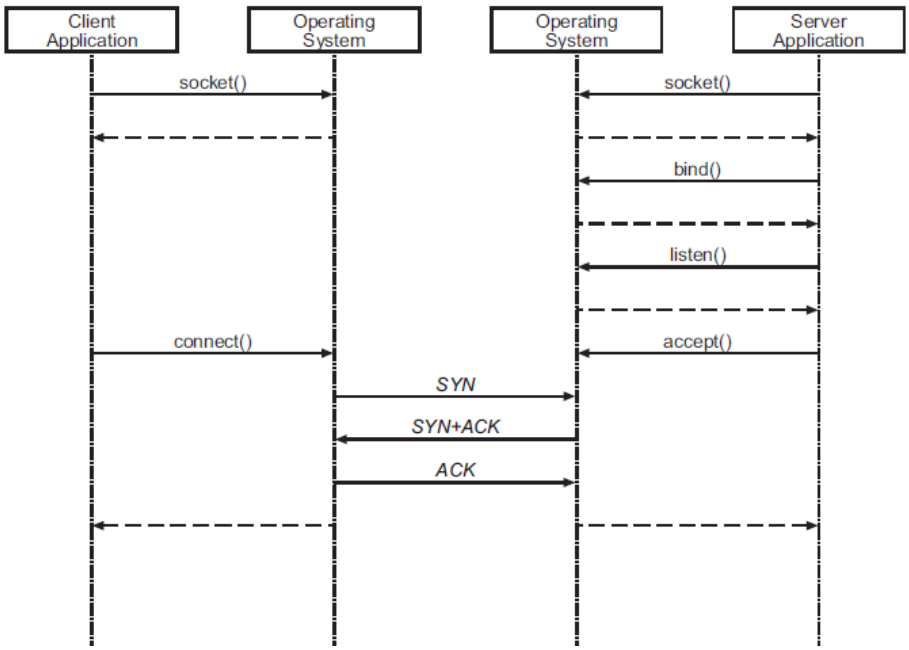


Figure 4.3: TCP 3 way handshake

Transmission Control Protocol (TCP) [24] is the standard protocol that is used for most of the data transmissions over the Internet. It use connection oriented approach providing a reliable data transmission between two processes running on computers connected by an IP network. Beside reliable connections, TCP provides stream data transfer, efficient flow control and acknowledgment to ensure stations are not flooded with data.

TCP is full duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction with stream data transfer, TCP delivers an unstructured stream of bytes identified by sequence number. For each data sent, the sequence number is set to indicate the last byte sent. Bytes are acknowledged on each TCP packet by setting the acknowledge sequence number to indicate that all lower numbered sequence number have been received. The reliability mechanism of TCP allows devices to deal with lost, delayed, duplicate packets.

4.2.3 UDP

User Datagram Protocol (UDP) [25] is another standard protocol used for data transmission over the internet. Unlike TCP, UDP use connectionless approach meaning that that do not establish end-to-end connections between communicating end systems. Therefore, the service provided by UDP is unreliable service that provides no guarantees for delivery of contents and does not promise to avoid duplication. However, the simplicity of UDP reduces the overhead form using the protocol and its services may be adequate for many cases.

One of the major advantages compared to TCP is that it supports broadcasting and multicasting messages. The broadcast messages are received by every host in a network and multicast messages are only received by the hosts that have subscribed to that particular multicast group. The length of the messages in UDP packet is limited by the maximum size of Internet Protocol packet. If the messages are longer that the maximum length, that application must break in different fragments and recombine it later.

Another significant benefit of UDP over TCP is that it reduces the amount of overhead of the network as it doesn't need to open a connection which saves a few packets. Sending of acknowledgements is carried out by the particular application or can be avoid completely.

UDP is popularly used for video streaming and also used as tunneling protocol, where a tunnel endpoint encapsulates the packets of another protocol inside UDP datagrams and transmits them to another tunnel endpoint, which decapsulates the UDP datagrams and forwards the original packets contained in the payload. UMTP (UDP Multicast Tunneling Protocol) is an example of such tunneling protocol.

4.2.4 Event notification tool: Libevent

Libevent [26] is an asynchronous event notification API that provides a mechanism to execute a callback function if any specific event occurs on file descriptor or after a timeout has been reached. The API provides a high performance event loop found in event driven networks and easy managing I/O events portably. It supports a number of backend such as select, poll, epoll, kqueue and /dev/poll. The core of libevent system acts as a wrapper around the underlying network backend and provides platform-specific APIs for monitoring large numbers of connections for I/O events. A program written in one platform based on these platform-specific APIs can be easily ported across all other platform such as Linux, FreeBSD and Solaris. The system makes it easy to add handlers for the connections while simplifying the underlying I/O complexities. The additional components including a buffer

event system for buffering data to/from clients and core implementations for HTTP and DNS system, add further functionality to the approach. The basic method for creating a libevent application is to register events and their call back functions that need to be executed if a particular operation or event occurs. Such operations could be as accepting a connection form a client. The event_base structure is implemented that holds a set of events and can poll to determine which event are active and finally the main event _dispatch () loop is called and the control of the execution process is handled by libevent system. Moreover, the events can be added and deleted from the event queue the enables to build flexible network handling systems.

4.3 Server software

4.3.1 HTTP Web server

A web server is software responsible for accepting HTTP [27] requests from clients, and serving them with HTTP responses. We used libevent framework to create a simple event driven HTTP server. The basic method for creating a event driven HTTP server is to register functions to be executed when a particular event occurs (such as accepting a connection for from client) and then call the main event loop. The entire process for running a HTTP server basically goes through four function calls: initialize, start HTTP server, set HTTP callback function, and enter event loop. The flow design of HTTP server using libevent API is shown in Figure 4.4.

A libevent event_base is a structure that holds a set of events and can poll to determing which events are active. The event_base also controls the backend method, such as epoll, kqueue, select and poll, that determines which events are ready. The libevent uses the high scable event mechanism as backend method by default. For example, in Linux it uses epoll and in FreeBSD it uses kqueue. The following code segment shows initializing of an event_base and changing the backend methods.

```
#include <event.h>
#include <evhttp.h>

struct event_base *base = event_init();
if (base == NULL) return -1;

struct event_config *config;
config=event_config_new();
```

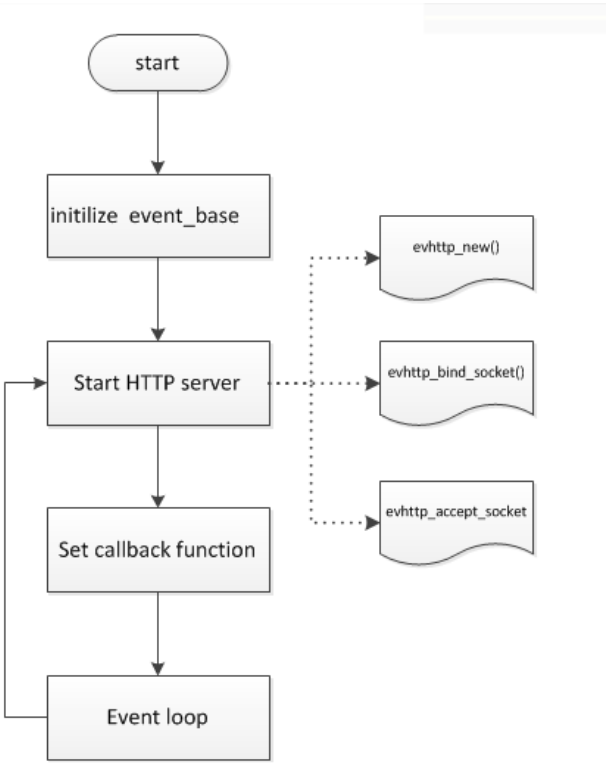


Figure 4.4: Design of HTTP server using libevent API

```
// changes the backend method to poll in case of Linux
event_config_avoid_method(config,"epoll");
// set the base with new backend method.
base=event_base_new_with_config(config);
```

An event_config is a structure that holds information about the preference for an event_base. A server is started by opening a listening socket and registering a callback function each time the server accept () a new connection. A server can be created by simply calling evhttp_new(). The following example code shows binding HTTP server on the specified address and port.

```
struct evhttp *httpd = evhttp_new(base);
if (httpd == NULL) return -1;
```

```
r = evhttp_accept_socket(httpd, nfd);
 if (r != 0) return -1;
```

Once the server is ready to accept new connection from the client, the callback function

is set that is invoked when an event is occured, that means when server accepts the connection from clients. The callback function is set as follows.

```
Function Interface:

Void evhttp_set_gencb(struct evhttp*, void(*)(struct evhttp_request) *, void*)


evhttp_set_gencb(httpd, generic_request_handler, NULL);

void generic_request_handler(struct evhttp_request *req, void *arg)
{
  struct evbuffer *evb = evbuffer_new();

  evbuffer_add_printf(evb, "%s",filedata);
  evhttp_send_reply(req, HTTP_OK, "Client", evb);
  evbuffer_free(evb);
}
```

If any event occurs the callback function invokes the generic_request_handler function that send HTTP_OK reply to the client. Finally a server is set to run in event loop. The event loop keeps running until there are no more registerd events or any loop break function is called. The interface for event loop is given below.

```
int event_base_dispatch(struct event_base *base);
```

4.3.2 UDP server implementation

For UDP traffic, we implemented a simple UDP server based on libevent API. The basic method used to implement an event driven UDP server is, similar to HTTP server, to register functions to be executed when a particular event occurs (such UDP data arrived on specified ports) and then call the main event loop. However the server use datagram socket for communication as it uses connectionless approach. The server uses multiple ports for listening to the UDP datagrams. The basic flow diagram of UDP server is shown in Figure 4.5.

An event_base structure that holds set of event is initialized. A server creates multiple numbers of socket and binds with different port numbers. The number of sockets opened can be controlled within the program code. The server listens to the UDP datagram in all the specified ports. Each time the new socket is created, a new event is registered in

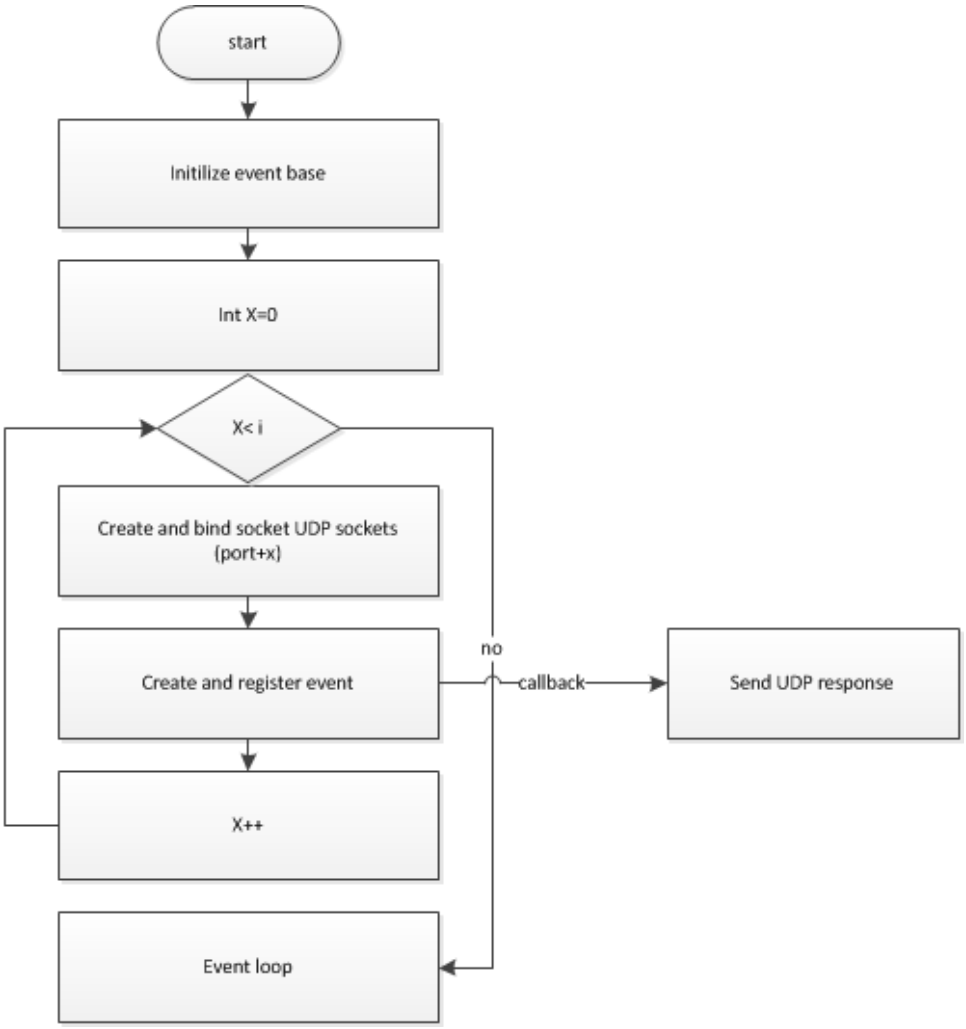


Figure 4.5: Flow diagram of UDP server implementation using Libevent API

event_base including the callback function. The events are invoked when data is ready for read on the sockets. Following segment of code shows the event registration for each socket created.

```
#include <event.h>

struct event_base *base=event_init();

if ( base==NULL)return -1;
int i=0;
for ( i=0;i<40;i++){
    int sockfd=bindsocketport(8000+i);
    if (sockfd<0)return -1;
```

```
struct event *ev=event_new(base, sockfd, EV_READ| EV_PERSIST,
echoback, base);
event_add(ev,NULL);
```

An event_new () creates the new event for the socket sockfd with the callback function echoback, which sends the reply back to the client. This event is triggered when the socket is ready to read the UDP datagrams. The event_add function registers the created event in event_base and the controls of events are handled by libevent. The final step is to run event loop which is done by calling function:

```
event_base_dispatch(base);
```

4.4 Client software

4.4.1 HTTP load generator

For TCP traffic experiment, two client machines are installed with httperf, an open loop work load generator. To automate the process of benchmarking, we also installed Autobench, a Perl scripts that runs httperf a number of time. Both the software can be freely downloaded from the Internet.

Table 4.3: Installation command

httperf	Autobench.
tar xvzf httperf-0.8.tar.gz	tar xvzf autobench-2.1.1.tar.gz
cd httperf-0.8	cd autobench-2.1.1
./configure	./configure
make	make
make install	make install

4.4.2 UDP client

To send the udp traffic, we implemented simple event based UDP client based on Libevent API. The basic idea of UDP client is that, it sends UDP datagram to the random server port numbers that are listening for UDP datagrams and calculates the response time. The simple flow diagram of UDP client is shown in Figure 4.6

The first step of implementation is to create and initialize event_base which holds the set of events. A UDP socket is created and bound with client port number. The client sends

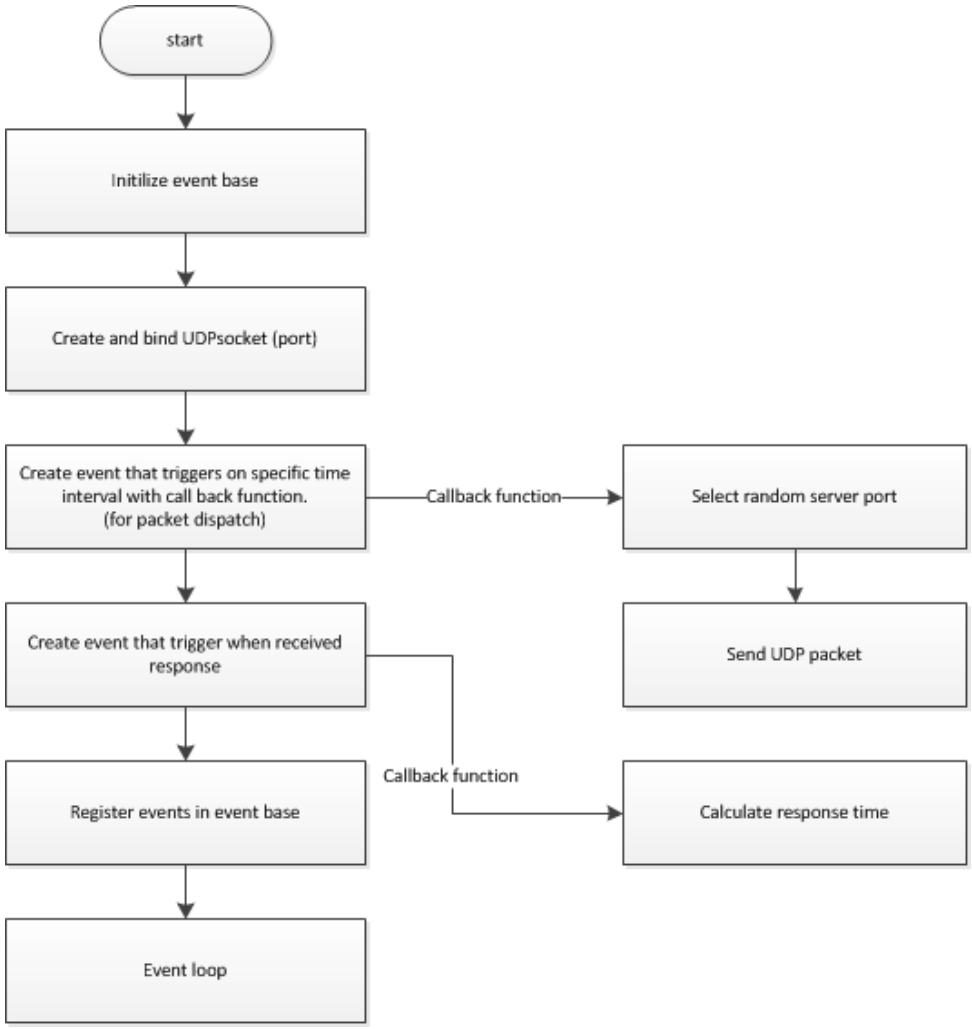


Figure 4.6: Flow diagram of UDP client implementation using Libevent API

and receives UDP packets via this port number. The timer event is set that triggers the event in specified interval of time. For example, to send 100 packets/s an event is triggered after every 100000 microsecond (100 events triggered in 1 sec). The timer event is set as follows.

```
#include <event.h>

struct timeval tv={0,100000};

event_set(&time_ev,-1, EV_TIMEOUT, timer_handler, portinfo);
event_add(&time_ev, &tv);
struct event *ev=event_new(base, portinfo->sockfd, EV_READ| EV_PERSIST,
 calculate_response_time, portinfo);
```

```
event_add(ev,NULL);
```

In the above example code segment, two events (EV_TIMEOUT and EV_READ) are registered in event_base. The first event is triggered after every time interval that is set in struct timeval(tv). Each time the event (EV_TIMEOUT) is triggered the client select one random server port number and sends UDP packets (server is listening on multiple number of ports). The second event (EV_READ)is triggerd when client get response from the server or when data is ready to read from the socket. The callback function calculates the server response time. The server response time is calculated as follows:

```
Server response time = packet arrival time - packet dispatch time
```

```
Average server response time (/sec) = total response time per second /
                                      total no of packet send per second
```

The final step is to run event loop and the execution control is handled by Libevent. The program is terminated when there are no events registered in event_base.

4.5 Benchmarking web server

A benchmark is simply a way to measure system performance. In order to conduct web server benchmarking, a system should at least have following components:

- server running the web server software under test,
- clients running load generating software in sufficient number so as to avoid their saturation,
- a network connecting the clients to the server which is free of other traffic which will not be saturated by the planned test.

The first step in benchmarking a web server is to decide what to measure. Some of the factors measured when benchmarking a web server is presented in Table 4.4.

4.5.1 Benchmarking tools

4.5.1.1 Httperf

Httperf [28] is an open source software tool for measuring web server performance that provides a flexible facility for generating various types of HTTP workloads. It is developed in HP Research labs and supports both HTTP/1.0 [27] and HTTP/1.1 [27].

Table 4.4: Factors measured when benchmanking a webserver

Throughput	data throughput is the rate at which server can process HTTP requests.
Request/sec	the number of requests per second that the web server is able to sustain.
Response time	the time a server spends processing a single request.
Reply rate	the number server responses per second.
Connection rate	the number of new opened connectios per second.
Simultaneous connection	the number of simultaneous connections the web server can handle without errors.
Error rate	the percentage of errors of a given type.

The tool basically measures the throughput of a web server by sending requests to the server at a fixed rate and measuring the rate at which the replies arrive. When test is run several times with monotonically increasing request rate one can see the reply rate level off when the server becomes saturated which means the server is operating in full capacity.

The example of httperf command line is shown below:
httperf –server=bill-linux.rd.tut.fi -port 8081 –rate=10 –num-conns=500 -num-calls=1
where , num-call :number of HTTP requests per connection num-conn: total number of connections to create rate: number of connections to start per second.

The above command connects and issues HTTP get request on the server bill-linux.rd.tut.fi running at port 8081. Httperf attempt total 500 connections, issuing 1 request per connection. The rate specifies that httperf should attempt to create 10 new connections per second.The maximum number of request generated can be calculated by num_call*rate, which is demand request rate =1*10 per second. An example output of httperf is shown in Figure 4.7.

4.5.1.2 Autobench

Autobench [29] is a collection of perl scripts that automate the process of benchmarking a web server or for conducting a comparative test of two different web servers. The scripts works as wrapper around httperf and runs httperf a number of times against the host increasing the number of requested connections per second on each iteration. The tool extracts the significant data for the httperf output and could be saved in CSV or TSV file

```
httperf --client=0/1 --server=www.example.com --port=80 --uri=/ --rate=10 --send-
buffer=4096 --recv-buffer=16384 --num-conns=500 --num-calls=1
Maximum connect burst length: 1

Total: connections 500 requests 500 replies 500 test-duration 50.354 s

Connection rate: 9.9 conn/s (100.7 ms/conn, <=8 concurrent connections)
Connection time [ms]: min 449.7 avg 465.1 max 2856.6 median 451.5 stddev 132.1
Connection time [ms]: connect 74.1
Connection length [replies/conn]: 1.000

Request rate: 9.9 req/s (100.7 ms/req)
Request size [B]: 65.0

Reply rate [replies/s]: min 9.2 avg 9.9 max 10.0 stddev 0.3 (10 samples)
Reply time [ms]: response 88.1 transfer 302.9
Reply size [B]: header 274.0 content 54744.0 footer 2.0 (total 55020.0)
Reply status: 1xx=0 2xx=500 3xx=0 4xx=0 5xx=0

CPU time [s]: user 15.65 system 34.65 (user 31.1% system 68.8% total 99.9%)
Net I/O: 534.1 KB/s (4.4*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Figure 4.7: An output example of Httperf.

format that could be directly imported into spreadsheets for further analysis. The result could also be directly plotted in graphical form using popular tool like gnuplot.

4.5.2 Procedure for benchmarking with Autobench

Our goal is to study the performance of event dispatch mechanisms used by the web server. To measure the server performance we installed the HTTP workload generator tool on two clients. In order to automate process of load testing of our web server, we also installed Autobench in both the client. The tool Autobench version 2.1.2 has introduced Distributed Autobench which allows us to conduct automated benchmarks using two or more client machines against the same server.

The distributed Autobench comprise of autobenchd and autobench_admin. The autobenchd listens to the instructions given from autobench_admin. The autobench_admin instructs the clients to benchmark the target web simultaneously and collects the result. The lab configuration of client and server achieved by autobench_admin is illustrated by Figure 4.8. Due to limited resources we used two client machines as autobenchd and in one of the client we open a new terminal which is used for autobench_admin. As autobench_admin simply instructs and collects the information from autobenchd, the performance was not affected while running both child and admin in the same client machine.

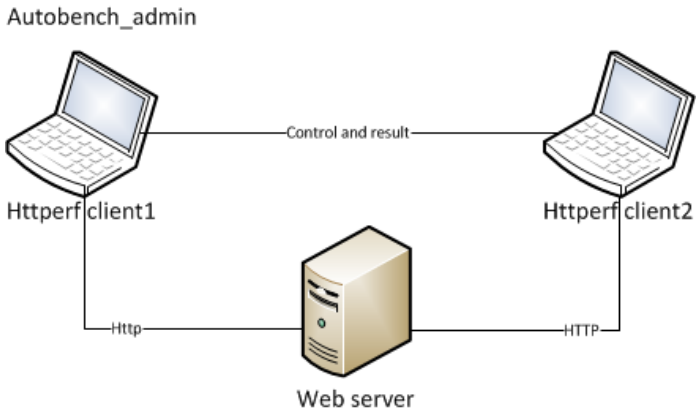


Figure 4.8: The lab configuration of client and server achieved by autobench_admin.

Before the benchmark can be started each participating client is put to the listen mode by starting the autobenchd daemon. The benchmark was conducted using following command:

```
autobench_admin -- single_host --host1 bill-linux.rd.tut.fi --port1 8081
-- clients 130.230.141.56:4600,130.230.141.57:4600 --low_rate 100
-- high_rate 450 -- rate_setp 50 -- num_call 100 -- num_conn 1000
- - timeout 5  file linux_epoll_result.tsv.
```

The command causes autobench to run a series of httperf tests, changing its parameters in such a way that it increases the load in every run and stresses the server in a distributed manner. The example of result obtained from above run is presented in Figure 4.9.

dem_req_rate	req_rate_	con_rate_	min_rep_rate	avg_rep_rate	max_rep_rate	stddev_rep_rate	resp_time	net_io_	errors_
10000	9750.2	97.5	9456.9	9857.7	10043.7	272.6	5.8	21348	0
15000	13271	132.7	12525.9	13567.3	14583.9	1085.3	9.9	29055	0
20000	17219	172.2	15761.5	17865.2	19968.9	2975	18.2	37699	0
25000	19348	193.5	16648.2	17862.2	19076.1	1716.8	22.2	42362	0
30000	20025	200.3	16879	18723.7	20568.4	2608.8	29.05	43843	0
35000	20612	206.1	16927.2	18963.9	21000.7	2880.3	35.25	45128	0
40000	20771	207.7	17054.2	19382.8	21711.4	3293.1	39.1	45478	0
45000	20977	209.7	17404.6	19688	21971.4	3229.2	42.9	45927	0

Figure 4.9: An output example of autobench.

dem_req_rate: is given from 100 to 450 with increasement of 50 requests per second for each round.The maximum number of request generated can be calculated by num_call*rate. req_rate: Request rate gives the rate at which HTTP requests were issued and the period that this rate corresponds to. con_rate: Connection rate shows rate at which the new

connection were made per second. min_rep_rate: minimal statistics for the reply rate. avg_rep_rate: average statistics for the reply rate. max_rep_rate: maximum statistics for the reply rate. stddev_rep_rate: standard deviation statistics for the reply rate. resp_time : Reply Time gives information on how long it took for the server to respond and how long it took to receive the reply net_io: Net I/O gives the average network throughput in kilobytes per second. errors: statistics on the errors that were encountered during a test. Errors includes client timeout, socket-timeout, connection refused, connection reset and file descriptors unavailable.

The changes in client configuration were not necessary once the test started. However we changed event dispatch mechanism parameter such as epoll, select, for each server and run the benchmark test. When the first test run set was completed, we introduce the idle connections to study the scalability of event dispatch mechanisms. The idle connection program maintains a steady number of idle connections to server. The server is preloaded with the specified number of idle connections and same test procedure is repeated. The results of the test run are presented in Chapter 5.

4.5.3 Procedure for UDP server Test

The UDP server discussed on Section 4.3.2 is ported to all the virtual hosts, compiled and ready to run. The client machine is ported with UDP client. The UDP server uses multiple ports to listen to the UDP datagram from the client. The number of ports that server listen to UDP datagram can be specified by user.

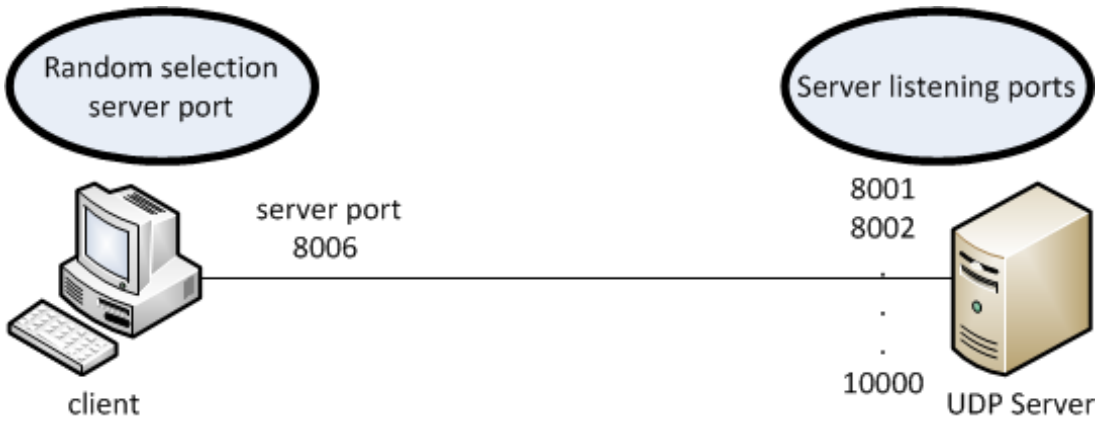


Figure 4.10: Communication between UDP client and server .

The client sends 10 UDP packets per second and for each packet the clients select the server listening random port and send the packet. The client calculates the server response

time and prints the result. We increased the number of ports the server is listening after each run and performed the same test. We took measurement for all the event dispatch mechanism supported by the UDP server on system.

After one set of measurements we increased the number of packets rate to 100per second on client side and followed the same procedure. The result obtained from UDP server tests are also discussed in Chapter 5.

Chapter 5

Results and discussion

In this chapter we present the results obtained from our experiment. These results are compared and the performance of different event dispatch mechaisms in different virtual hosts is analysed.

5.1 Test case: TCP traffic and discussion

At first, the HTTP server is implemented using Libevent and ported into all the virtual hosts. These HTTP servers are used to compare the performance of the event dispatch mechanism in all the hosts. An open-loop workload generator, httperf, is used to generate the TCP traffic and measure the server performance in different platforms.

Figure 5.1 shows the throughput achieved by HTTP servers, with 1000 active connections, implemented on virtual hosts using different event mechanisms. The achieved throughput increases linearly with offered load until the server starts to become saturated. After the saturated point, the server throughput starts to fall off as the increasing amount of time is spent in the kernel to handle the network packets. In absence of idle connections, we can clearly see in the graphs that select and poll perform comparetively well as other high performance event mechanism such as epoll, kqueue, /dev/poll and event ports. This performance is achieved as the number of socket descriptors tracked by each mechanism is low.

The graphs also show that the select and poll have similar characteristics, as they both essentially provide the same functionlity. Both the functions examine a set of file descriptors in a linear way to see if specific events are pending and then optionally wait for a specified time of an event to happen. However the basic difference is that select ()'s fd_set is a bit mask and therefore has some fixed size. With poll (), the user must allocate an array of

pollfd structures and pass the number of entries in this array, so there is no fundamental limit.

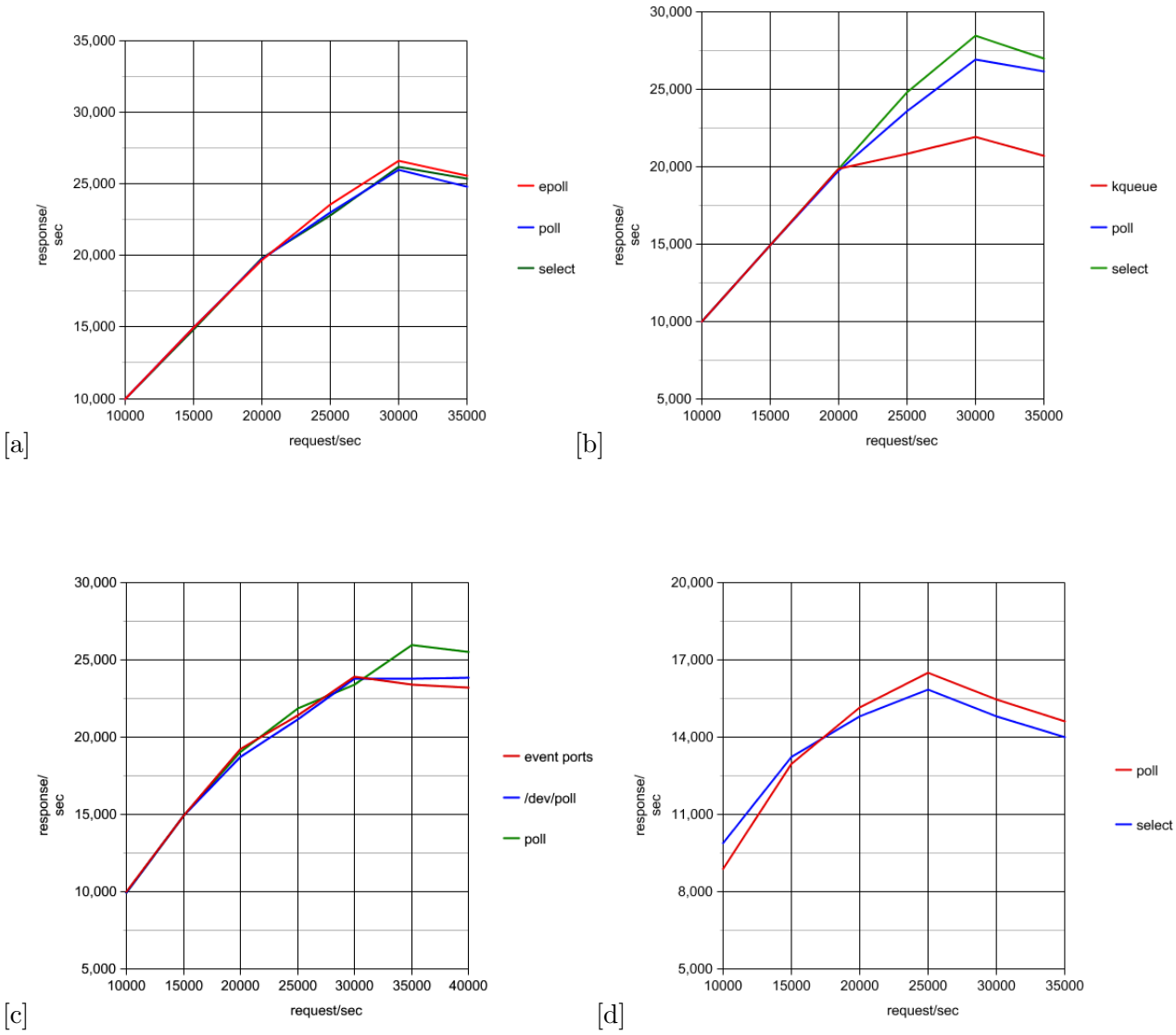


Figure 5.1: Performance of event mechanism in different virtual hosts: (a) Linux performance on TCP response rate using epoll, poll and select; (b) FreeBSD performance on TCP response rate using kqueue, poll and select; (c) Solaris performance on TCP response rate using /dev/poll, event ports, poll ; (d) Windows performance on TCP response rate using poll and select.

The presented graph shows that performance of FreeBSD (kqueue) and Open Solaris (/dev/poll/ and event ports) are much less than expected and not able to compete with the performance of Linux (epoll). Even select and poll scales much better than kqueue and event ports. One of the reason is when most of the descriptors are active the work done by select and poll in not such a waste but in case of kqueue and event ports, requires two

system calls per descriptor event. Another possible reason for this behavior is, as discussed earlier, the benefits of VMs are the penalty of performance degradation. Reduction of the performance overhead could be possible as modern VMware allow guest VMs to access hardware resources directly whenever it is possible. However, optimizations with such direct access are not always possible in case of network I/O operation. Because I/O devices are usually shared among all VMs in a physical machine and the VMM has to make sure accesses to them are valid and consistent. [30]

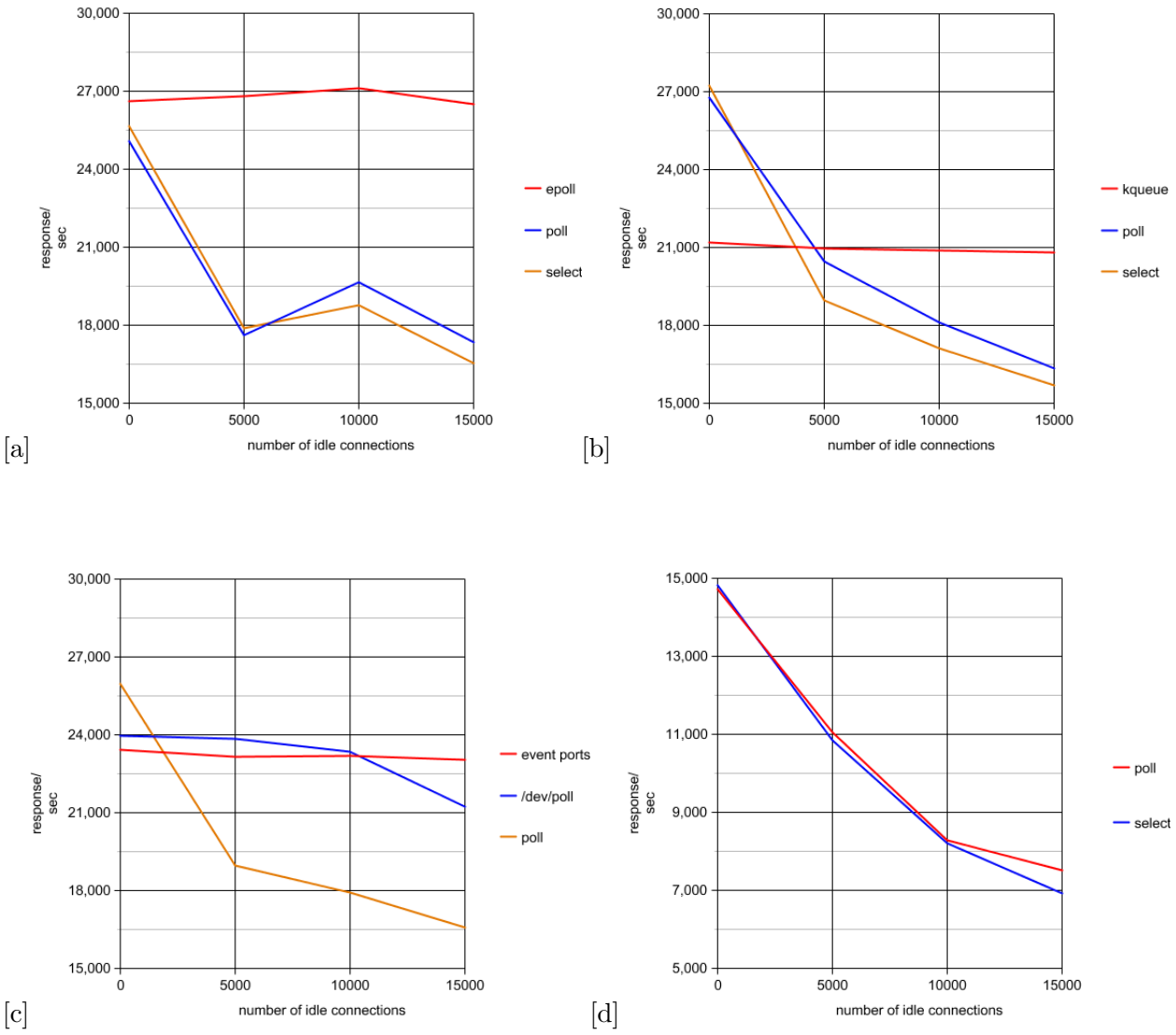


Figure 5.2: Performance of event mechanism with idle connections: (a) Linux performance on TCP response rate in presence of idle connection; (b) FreeBSD performance on TCP response rate in presence of idle connection; (c) Solaris performance on TCP response rate in presence of idle connection; and, (d) Windows performance on TCP response rate in presence of idle connection.

5.2 Test case : TCP traffic with idle connections

The idle connections are introduced to simulate the presence of large numbers of simultaneous connections. As a result the event dispatch mechanism has to keep track of large number of descriptors. However, the active descriptors may be very small portion.The main goal of this study is to look at the behavior of servers under high load as the number of concurrent connection increases.

In each test, the client opens a number of TCP connections to the server and no traffic flows over these connections. These values are presented in X-axis of the graphs for each server on different virtual hosts. The maximum HTTP requests that each server can handle are sent and kept constant, while the number of concurrent connections was varied to see the effect of large number of idle connections on server performance. Y-axis shows the number of request answered per second. Figure 5.2 presents HTTP server performance with 1000 active connections and increasing idle connections.

When we compare the result from Figures 5.1 and 5.2, we can clearly see that response rate of epoll, kqueue, /dev/poll/ and event ports is almost constant. However, the response rate of select and poll in each system shows rapid degradation. The overhead associated with select and poll mechanisms are memory-less property which means that they do not remember what file descriptors the application is interested in. As a result, the kernel must perform linear scan the query parameters of all the interested file descriptors, copying data between kernel and application on each call. In addition, the application does another linear scan on the return values of select () and poll () which makes a network server to scale poorly when it is overloaded with connections.

5.3 Test case: UDP traffic and discussion

To test the performance of event dispatch mechanism of different server hosts for UDP traffic, we implemented a simple Libevent based UDP server and ported it into different virtual hosts. The server opens a number of specified ports to listen and wait for a datagram request from a client. Another, simple UDP client is implemented in client side. The client sends a datagram to the server which then processes the information and returns a response. The client keeps track of the requests time and response received time from the server.

To stress different event dispatch mechanisms, we increased the number of ports opened in the server ranging from 10 to 5000. All these ports are ready to accept UDP datagrams.

However, only few ports receive the request from the client. The test is conducted using 10 and 100 UDP request per second to the server. For each packet or request the client choose the random opened server port and sends the request.

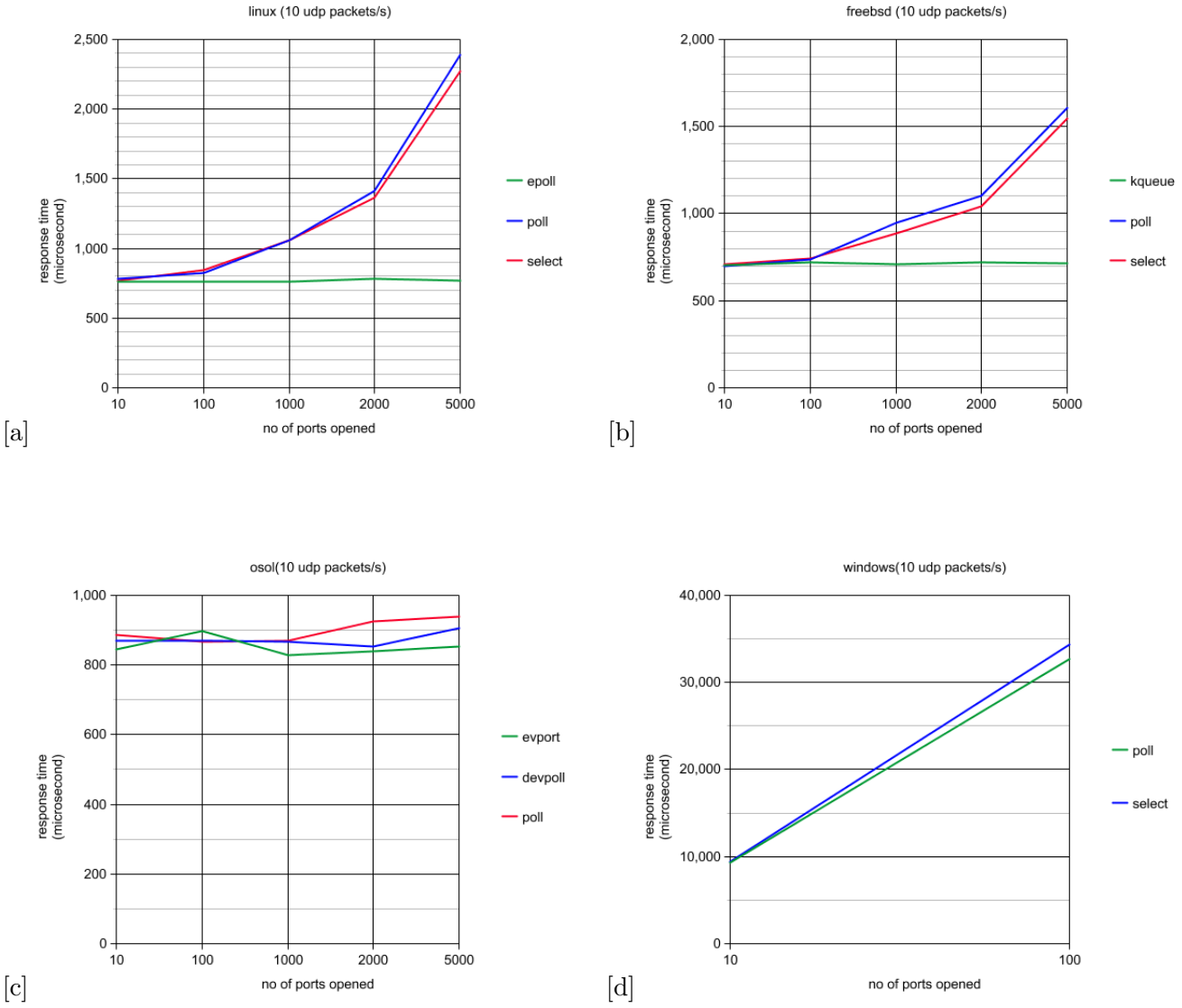


Figure 5.3: Performance of event mechanism for 10 UDP packets per second: (a) server response time for Linux; (b) server response time for FreeBSD; (c) server response time for Solaris ; and, (d) server response time for Windows7.

Figure 5.3 and 5.4 show the response time of different servers under different traffic load. From the graphs we can clearly see that the response time for epoll and kqueue remains almost same throughout the test. On the other hand, as expected, for Linux, FreeBSD and Windows7 servers, the response time for select and poll seems to have increased rapidly as we increase number of ports opened. The mostly likely explanation for this is that

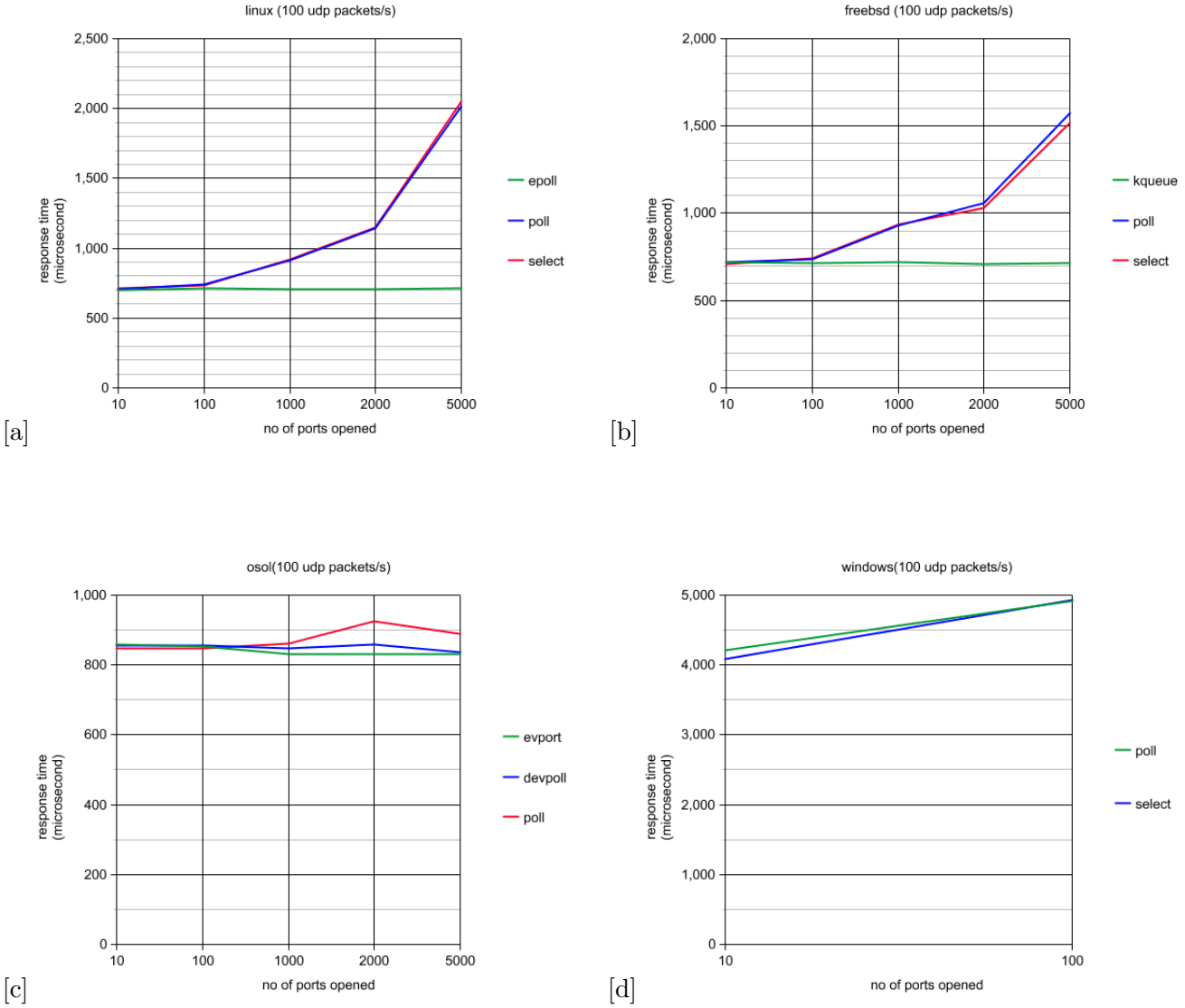


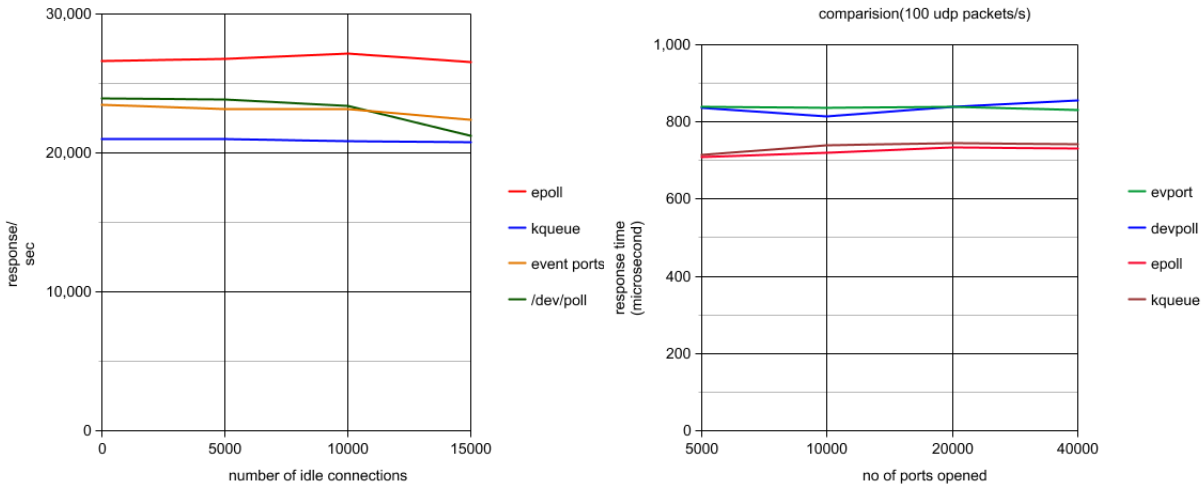
Figure 5.4: Performance of event mechanism for 100 UDP packets per second: (a) server response time for Linux; (b) server response time for FreeBSD; (c) server response time for Solaris ; and, (d) server response time for Windows7.

time spending on event detection is too long to maintain good response time when server listening ports are increased. In the Solaris system, apart from event ports and /dev/poll/, poll has much better response time than Linux and FreeBSD systems. However, response time for Windows7 is much higher than any other hosts.

5.4 Comparison of Result

The performance of Windows7 event dispatch mechanism tested in both the traffic shows poor performance. The TCP response rate is much lower and server response time in UDP

traffic is much higher as compared to other operating system. To compare the scalability of high performance event mechanisms more closely, we excluded the result of select and poll of all the system. The comparative result of event dispatch mechanism of different operating system for both the traffic is show in Figure 5.5.



(a) **(b)**

Figure 5.5: Performance of event mechanisms: (a)TCP response rate with idle connections(excluding select and poll); (b) server response time for UDP traffic (excluding select and poll).

The figure clearly shows that scalability of Linux epoll in both the traffic cases is much better than event dispatch mechanisms in other operating system. The performance of Solaris(/dev/poll and event ports) cannot compete with Linux but scales slightly better than FreeBSD(kqueue). However, the Solaris /dev/poll started degrade slowly when 5000 idle connection is introduced. The /dev/poll curve falls fast when 15000 idle connections are used. This give us a hint that when we increase more idle connections the /dev/poll scalability performance could be less that FreeBSD (kqueue), as the performance of kqueue is much steady. In case of UDP traffic, the server response time of FreeBSD(kqueue) is much better than Solaris event mechanisms. However the response time for Linux (epoll) is better than kqueue.

Chapter 6

Conclusions

Virtualization technology addresses the needs of massive data center environment by providing efficient, secure and flexible system infrastructure to meet the demanding resource requirements of modern computing systems. Hypervisor VMware ESXi provides the foundation for building and managing reliable virtualized IT infrastructure by abstracting resources into multiple virtual machines and reduces the hardware cost by running multiple operating systems on single server.

We conducted sets of tests while measuring performance of the network event dispatch mechanisms for different operating systems installed on virtual hosts. These virtual hosts are installed on VMwareESXi hypervisors. We implemented simple event based HTTP web server based on Libevent API to compare the performance of different event mechanism supported on various operating system. To test the performance of these event mechanism on UDP traffic, we also implemented a simple UDP client and server based on Libevent.

In our web server benchmark test, we used httperf to generate HTTP traffic. The httperf uses connection timeouts to generate the loads that can exceed the capacity of the server. We used autobench to automate our test process. From the web server benchmark result, we observed that in the absence of idle connections, the event mechanisms select and poll performed comparatively well as regards to the high performance event mechanisms (epoll, kqueue,/dev/poll) in all the platforms. This is due to the fact that in this experiment the number of socket descriptors tracked by each event mechanism is not very high. The performance of FreeBSD (kqueue) is highly expected to perform well but Linux epoll performs much better than all the other mechanism measured.

The scalability issues of event dispatch mechanism can be observer by increasing the number of socket descriptors. We preloaded the server with steady number of idle connectio-

ns and run the similar test again. The result obtained from these tests shows that the performance of select and poll degrades rapidly in all the system as the number of idle connection increased. However, for Linux epoll, FreeBSD kqueue, Open Solaris event ports doesn't show much effect. The Solaris /dev/poll started degrade slowly when 5000 idle connection is introduced. The /dev/poll curve falls rapidly when 15000 idle connections are used. From the all the obtained graph we can clearly conclude that Linux epoll scales much better than all the event mechanisms tested and Windows 7 shows very poor scalability.

The graph obtained from the UDP server benchmark show the similar pattern, as the number of ports opened increased the response time for select and poll increased rapidly in all the platforms. However, in Solaris the performance of select and poll is comparatively good than in other platforms. The response time of kqueue is slightly more than epoll showing that Linux performs much better and suitable for UDP traffic.

From all the result obtained we can conclude that Linux is stable in virtual environment and performs comparatively well for both UDP and TCP traffic. FreeBSD and Solaris are stable as well but dispatch mechanism doesn't scale as expected. Windows7 show poor performance in both the traffic cases. The efficiency for these systems could be affected by virtual environment as the benefit of virtualization comes with the penalty of performance degradation.

References

[1] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *"UNIX Network Programming : The Sockets Networking API"*. Addison-Wesley Professional, third edition, November 14, 2003.

[2] VMware ESX and VMware ESXi . The Market Leading Production-Proven Hypervisors . [Online] Available via http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf, referred January 12th, 2011.

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

[4] Oren Laadan and Jason Nieh. Operating system virtualization: Practice and Experience. *ACM 978-1-60558-908-4/10/05*, 2010 May.

[5] Clark Scheffy. Virtualization For Dummies. *AMD Special Edition 2007,Wiley Publishing*, 2007.

[6] Robert Rose. Survey of System Virtualization Techniques . March 2004.

[7] R. P Popek, G. J. Goldberg. Formal requirements for virtualizable third generation architecture. *communications of the ACM archive Volume 17, Issue 7*, page 412 421, july 1974.

[8] Abels, T. Dhawan, and P . Chandrasekaran B. An overview of Xen Virtualization. [Online] Available via http://www.globalsecurity.org, referred December 28th, 2005.

[9] Karen Scarfone, Murugiah Souppaya, and Paul Hoffman. Guide to Security for Full Virtualization Technologies. *National Institute of Standards and Technology, Gaithersburg,*, jan 2001.

[10] Welcome to Linux - VServer.org Linux. [Online] Available via http://linux-vserver.org/Welcome_to_Linux-VServer.org, referred December 28th, 2011.

[11] Virtualization Overview. [Online] Available via http://www.globalsecurity.org, referred December 28th, 2005.

[12] OpenVZ. [Online] Available via http://old.openvz.org, referred December 28th, 2011.

[13] Poul Henning Kamp and Robert N. M. Watson. Jails.Confining the omnipotent root. *In Proceedings of the 2nd International SANE Conference*, 2000.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM ,SOSP'03 , Bolton Landing, New York, USA.1581137575/03/0010* , 2003.

[15] KVM: Kernel-based Virtualization Driver. [Online] Available via http://http://www.linuxinsight.com/files/kvm_whitepaper.pdf, referred january 16th, 2012.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. *USENIX Annual Technical Conference*, 2005.

[17] Oracle VM VirtualBox User Manual. [Online] Available via http://www.virtualbox.org, referred May 20 12th, 2012.

[18] L. Gammo, T. Brech, A. Shukla, and D. Pariag. Comparing and Evaluating epoll, select and poll Event Mechanisms. *Linux Symposium 2004. Volume 1.*

[19] Oracle Internals Notes Asynchronous I/O. . [Online] Available via http://www.ixora.com.au/notes/asynchronous_io.htm, referred january 16th, 2012.

[20] A. Chandra and D. Mosberger. Scalability of linux Event dispatch Mechanisms. *HP Laboratories Palo Alto, HPL-2000-174*, 2000.

[21] Hao-Ran Liu and Tien-Fu Chen. A Scalable Locality-Aware Event Dispatching Mechanism for Network Servers. *Department of Computer Science National Chung Cheng University Chiayi, Taiwan 621, ROC.*

[22] KQUEUE(2) FreeBSD System Calls Manual. . [Online] Available via http://www.freebsd.org/cgi/man.cgi?query=kqueuesektion=2, referred january 16th, 2012.

[23] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. *FreeBSD Project.* Available via http://people.freebsd.org/ jlemon/papers/kqueue.pdf, referred December 13th, 2011.

[24] Transmission Control Protocol. *RFC 793, IETF,January 1980.*

[25] J. Postel. "User Datagram Protocol. *RFC 768,IETF, August 1980.*

[26] Fast portable non-blocking network programming with Libevent. . [Online] Available via http://www.wangafu.net/ nickm/libevent-book/, referred january 16th, 2012.

[27] Hypertext Transfer Protocol – HTTP/1.1 . *RFC 2616, IETF,.*

[28] httperf HTTP performance measurement tool. . [Online] Available via http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.pdf/, referred january 16th, 2012.

[29] Autobench. . [Online] Available via http://http://www.xenoclast.org/autobench/, referred january 16th, 2012.

[30] Performance Best Practices for VMware vSphere 4.1. [Online] Available via http://www.vmware.com, referred january 16th, 2012.