High performance kernel mode web server for Windows
Degree Project D, 10 credits

Bo Brantén

Department for Applied Physics and Electronics at Umeå University

1st June 2005

Abstract

This degree project describes a method to increase the performance of a web server by splitting it in two parts, one part executes in kernel mode and process requests for static web pages and picture files while requests for dynamic web pages that is generated by executing scripts or contains information from an database is handed over to a web server that executes in user mode. The web server that executes in kernel mode can be optimized for its purpose and minimizes process scheduling and copying of data which means that a larger number of requests per second can be processed. The web server that executes in user mode can be an unmodified standard web server. In the project a prototype is implemented for the Windows family of operating systems, specific problems that is solved includes network communication from kernel mode and how to hand over a request to user mode. The prototype implementation is used to measure performance and evaluate how well the method works. The project is also related to earlier work in Linux.

Sammanfattning

Det här examensarbetet beskriver en metod att höja prestanda för en webbserver genom att dela upp den i två delar, den ena delen exekverar i system-mode och behandlar anrop till statiska webbsidor och bildfiler medan anrop till dynamiska webbsidor, dvs sådana som genereras genom exekvering av skript eller innehåller information som hämtas från en databas, skickas vidare till en webbserver som exekverar i användar-mode. Den webbserver som exekverar i system-mode kan optimeras för sin tillämpning och innebär att man minimerar processchedulering och kopiering av data varför ett större antal anrop per sekund kan behandlas. Den webbserver som exekverar i användar-mode kan vara en omodifierad webbserver av standardtyp. I examensarbetet utvecklas en prototyp för Windows-familjen av operativsystem, speciella problem som löses är nätverkskommunikation från system-mode och hur överlämning av anrop till användar-mode kan utföras. Prototyp-implementationen används för att göra prestanda mätningar och utvärdera hur väl metoden fungerar. Arbetet relateras också till tidigare arbeten i Linux.

Contents

1 Introduction

This section contains information about the report that is not directly related to the topic itself. Some background to the degree project is given. Information on the author and supervisors is presented. Also some details on the hardware and software used during the project are given. Finally the acknowledgments are presented.

1.1 About this report

This is a report on a degree project at the department for Applied Physics and Electronics[1] at Umeå University. Some of the areas the department focus on is data communication and embedded systems, they have given courses on development of device drivers for both Windows and Linux in their four year education to a Degree of Master of Science with a major in Applied Electronics.

The intention of the degree project is to further develop the student's ability to gain deeper knowledge in a subject and apply it to a bigger project. It also gives insights in research and development work and is an exercise in documenting and presenting a project.

As a degree project I have chosen to investigate a method to improve the performance of a web server and develop a prototype for Windows using this method. The project is called "High performance kernel mode web server for Windows". First this report gives some background on the method and earlier work in Linux, this is followed by a description of the Windows networking architecture. Then the prototype implementation is described. This prototype is used to evaluate how the method works in the Windows environment and performance measurements is done. Finally the results are analyzed and conclusions are drawn.

The selection of this degree project is based on personal interest and work experience from device drivers in Windows, particular network communication from kernel mode and file system drivers.

1.2 About the author

The author of this report Bo Brantén is studying to a Degree of Master of Science with a major in Applied Electronics at the department for Applied Physics and Electronics at Umeå University. In addition to the studies he has worked with Windows device drivers both as a teaching assistant and in the industry, he is also internationally recognized in the open source community

for providing information on Windows file system drivers. Table 1 contains information on how to contact him.

Name	Bo Brantén
E-mail	`bosse@acc.umu.se`

Table 1: Author contact information

1.3 Supervisors

The supervisor for this degree project is Ulf Brydsten, Fil lic and the examinator is Dan Weinehall, MSc. Both are working at the department for Applied Physics and Electronics at Umeå University. Their contact information is provided in table 2 and the address to the department is provided in table 3 .

Name	Ulf Brydsten	Dan Weinehall
E-mail	`ulf.brydsten@tfe.umu.se`	`dan.weinehall@tfe.umu.se`

Table 2: Supervisor contact information

Name	Department for Applied Physics and Electronics Umeå University
Address	901 87 Umeå Sweden
Phone	+46-(0)90-786 50 00
Fax	+46-(0)90-786 64 69

Table 3: Department contact information

1.4 Typographical conventions

In the report the following typographical conventions is used.

Roman regular — Used for normal text.

Roman bold — Used for description and table headings.

`Constant width` — Used for source code, email addresses and URLs.

1.5 Hardware and software used

The prototype was implemented on a standard personal computer running Windows. Table 4 lists the software used.

Windows® 2000 Professional with Service Pack 4
Visual Studio® 6.0 with Service Pack 5
Windows® 2000 Driver Development Kit (DDK)
Debugging Tools for Windows® 6.4.7.2
SoftICE™ 2.5
A collection of tools from OSR[2] and Sysinternals[3].

Table 4: Software used

For the evaluation of the prototype a test network was built, the equipment used is described in section 5 on page 32.

The report was typeset by the author in LaTeX.

1.6 Acknowledgments

I would like to thank the Department for Applied Physics and Electronics for creating a friendly environment for studies. Ulf Brydsten has been of great support both during the degree project and in the earlier studies. Also I want to thank Anders Myrberg and the others at GTC Security AB[4] for given me the opportunity to learn more about the internals of Windows file system drivers and network communication from kernel mode.

Umeå 1st June 2005 Bo Brantén

2 Background

This section gives some background to the project, it describes earlier work in Linux and some methods to improve the network API.

2.1 The Mindcraft benchmark

In April 1999 a company called Mindcraft published a benchmark[5] comparing Windows NT Server 4.0 and IIS with Red Hat Linux and Apache. The conclusion of the benchmark was:

> Microsoft Windows NT Server 4.0 is 2.5 times faster than Linux as a File Server and 3.7 times faster as a Web Server.

Mindcraft is a company which specializes in benchmarking systems for paying customers, in this case the customer was Microsoft.

This benchmark became heavily debated, it was criticized by the Linux community for using an improper configuration of Linux and Apache for a big server under high load and exploiting weaknesses in them with the paying customer in mind. Even though Mindcraft has admitted that the benchmark had problems and has since published new results it showed that there was a need to implement new system calls and new communication architectures in the Linux kernel to support high performance web servers, it became the starting point of a number of activities to respond to that.

2.2 Improvements to the network API

This section describes a number of improvements to the network API that is related to the performance of web servers.

2.2.1 sendfile()

The traditional way for an application to send a file is to read parts of it into a buffer and then write the buffer contents to a socket, this means that the data will be copied several times to and from the user address space and that several context switches is needed. To optimize this a new system call `sendfile()` was developed. It takes two file descriptors as parameters, one referring to a file to read data from and one referring to a socket that the data is written to, since this is done in the kernel with one system call the amount of data copying and context switches will be reduced. The `sendfile()` system call was developed to improve the performance of user

mode web servers but it can also be used in kernel mode. Following is the function prototype for `sendfile()` with a description of its parameters and return value.

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

out_fd File descriptor opened for writing.

in_fd File descriptor opened for reading.

offset Pointer to a variable holding the input file pointer position from which `sendfile()` will start reading data. When `sendfile()` returns, this variable will be set to the offset of the byte following the last byte that was read.

count The number of bytes to copy between the file descriptors.

The return value is the number of bytes copied.

The `sendfile()` system call copies data from the file system cache to the socket buffer in kernel mode but even this copy can be eliminated by using something called "zero-copy networking" witch means that the TCP/IP-stack can read the data directly from the file system cache, it is described in one of the following sections.

On Windows there is a corresponding system call `TransmitFile()` available for user mode applications.

2.2.2 TCP_CORK

A new socket option called `TCP_CORK` was introduced together with the `sendfile()` system call. If this option is set on a socket the TCP/IP-stack will not send a small packet over the network as soon as something has been written to the socket but instead wait for more data, the send will take place either when enough data has been written to the socket or when the `TCP_CORK` socket option is cleared again. This is especially useful together with `sendfile()` since it is common to send a header before the file. Using the `TCP_CORK` socket option the header will be sent together with the file data witch decreases the number of packets.

Also without using `TCP_CORK` the example above would suffer from the effect of "Nagle's algorithm" that was invented to avoid that a packet is sent over the network for every keystroke in a terminal application. If the application writes characters one by one to the TCP/IP-stack Nagle's algorithm will send the first character right away but the next one is buffered awaiting

acknowledge for the first packet, during this wait more characters could be entered by the user and sent together as one packet or a packet could arrive from the server, the buffered data can then be sent together with the acknowledge for that packet. This algorithm will decrease the network bandwidth used by a terminal application but for a web server that typically sends a header followed by a file it would split the header and file data in two packets and introduce a delay for the second packet.

Using the `TCP_CORK` socket option is an important optimization for both user mode and kernel mode web servers.

On Windows there is no socket option that directly corresponds to `TCP_CORK` however this is not a problem since the user mode function `TransmitFile()` has parameters for an optional header and trailer to the file and the kernel mode TDI interface is using chained MDLs, therefore it is possible to put together the data that should be sent as one packet before handing it over to the TCP/IP-stack.

2.2.3 Zero-copy networking

As was described in an earlier section the `sendfile()` system call copies data from the file system cache to the socket buffer. New hardware features makes it possible to eliminate even this final coping of data by implementing something called "zero-copy networking". To utilize zero-copy networking the network card must support two features in hardware:

- Scatter/gather I/O.
- Calculation of TCP and IP checksums.

When `sendfile()` is used on a system with this kind of network card is it possible for the TCP/IP-stack to read data directly from the file system cache, calculate the checksum in hardware and send the packet on the network without any extra copying being done.

An application or kernel mode code doesn't have to do anything special to utilize zero-copy networking, it is implemented by the TCP/IP-stack and the driver for the network card.

On Windows the requirements to utilize zero-copy networking is the same, it will be available if the network card and driver supports it. The kernel mode TDI interface is using chained MDLs and the file system interface supports requesting such chained MDLs representing data in the file system cache. In user mode is the function `TransmitFile()` available, it uses the just described features in its implementation.

2.2.4 Delayed accept

A basic design of a web server is to call `accept()` on the main socket in a loop, when a new connection is established does `accept()` return a socket representing it and the application creates a new thread or process to handle the request where after the loop calls `accept()` again waiting for more connections. The new thread or process will begin by calling `recv()` that blocks waiting for data to arrive on the socket.

This process can be optimized by implementing something called "delayed accept". When a client connects to the server is the connection implicitly accepted by the TCP/IP-stack but the application isn't notified yet, instead the TCP/IP-stack waits until some amount of data has arrived. The application will then be given both a socket representing the new connection and a buffer containing the first data, for example a HTTP-request, with the same system call. This will decrease the amount of scheduling needed. On Linux can delayed accept be requested by setting the socket option `TCP_DEFER_ACCEPT`, Windows has instead implemented a special function `AcceptEx()` that both do accept and returns the first data received while FreeBSD even lets the user install an accept filter that can specify what data to receive before accepting an connection.

2.2.5 Send and disconnect

After the web server has sent all the data of a file it notifies the client of this by doing disconnect, the standard way to do this is calling `shutdown()` with the how parameter set to `SD_SEND` followed by `close()` on the socket. However it is possible that the TCP/IP-stack would transfer the disconnect information in its own network packet, therefore it would be an advantage if one could flag the data sent as the "last data to go" and have the TCP/IP-stack send the disconnect information together with it. Neither on Windows nor on Linux does the TCP/IP-stack has an interface to do send and disconnect as one integrated operation, however the function `TransmitFile()` on Windows takes a flag `TF_DISCONNECT` that will disconnect the socket after sending the file.

Something to think of when using the TDI interface on Windows is to not wait for the last (and for small files possible only) send to complete before doing disconnect because that means waiting for an acknowledge to arrive from the remote node on the data packet, this unnecessarily delays the disconnect.

This may not sound like the most important optimization but in fact it helps getting good results in certain benchmarks namely the number of

requests per second for small files using a serialized client. In this case the largest amount of time is taken by connecting and disconnecting the TCP protocol, after receiving the file data the client waits for more data to arrive or disconnect to happen before it sends a new request, a fast disconnect will improve the benchmark result in this situation.

2.3 Previous work

By tradition software is divided into what is executed in user mode respective what is executed in kernel mode and as much as possible is placed in user mode while kernel mode is only chosen when direct access to hardware is needed or performance motivates it, for example by device drivers.

The idea that the Linux developers got was to increase the performance of a web server by splitting it in two parts, one part executes in kernel mode and process requests for static web pages and picture files while requests for dynamic web pages that is generated by executing scripts or contains information from an database is handed over to a web server that executes in user mode. The advantages of executing a web server in kernel mode is that it decreases process scheduling, context switching and copying of data, also it makes it possible to integrate the web server with the TCP/IP-stack and not be limited to the standard socket interface. This makes it possible to process a larger number of requests per second and get higher throughput. The web server that executes in user mode can be an unmodified standard web server.

In the Linux community two projects has been created based on this method, kHTTPd by Arjan van de Ven and TUX by Ingo Molnar, they are described in the following sections.

2.3.1 kHTTPd

kHTTPd[6] was the first kernel mode web server project for Linux, it was started in 1999 by Arjan van de Ven as a response to the MindCraft benchmark. kHTTPd was developed on the 2.2 series of the Linux kernel and was integrated in the main kernel distribution from version 2.3.14.

To begin with the kHTTPd server used a simple architecture, it started a new thread for every request and the communication that should be handed over to the user mode web server was copied from socket to socket. However kHTTPd was soon developed to use an architecture that allows for higher performance, it is shortly described in the following paragraph.

When the kHTTPd kernel module is loaded it creates one thread for each

CPU on the system. The processing of requests is divided in a number of states with a corresponding queue to each state. The requests is transferred from one queue to the next when the state changes. The states are:

- Wait for accept.
- Wait for HTTP-header.
- Send data or hand over to user mode web server.
- Cleanup.

The first state "wait for accept" is queued by the operating system while the other queues is handled by kHTTPd. After parsing the HTTP-header the request is either put in the send data queue or handed over to the user mode web server, kHTTPd selects between them depending on the URL and the filename extension. If the request is to be handed over to the user mode web server kHTTPd directly enters it in the accept queue for that socket, this can be done since a kernel module in Linux has access to data structures and functions internal to the kernel and the source code is available. The normal setup is that kHTTPd is listening on port 80 while the user mode web server is listening on some other port for example 8080. In the cleanup queue the socket is closed and information about the request is logged. The requests waiting in the queues is regularly checked using non-blocking system calls to see if any input or output can be done.

The measurements published on the kHTTPd project homepage claims that it was four to five times faster than Apache on the same system at that time. Later measurements[9] has showed that kHTTPd is faster or at least as fast as the best user mode web servers but that it is outperformed by newer kernel mode web servers using better architectures.

In the 2.5 series of development kernels kHTTPd was again removed from the main kernel distribution for a number of reasons among them the debate if a web server belongs in the kernel and the availability of TUX, a new project described in the next section. Also kHTTPd had been un-maintained for a while. Therefore it was decided that this kind of project is best maintained separately outside the main kernel distribution.

2.3.2 TUX

TUX[7][8] stands for "Threaded linUX http layer" and is designed and implemented by Ingo Molnar at Red Hat. Like kHTTPd it is also a kernel mode web server but it differs in design and features. TUX is available for both the 2.4 and the 2.6 series of the Linux kernel and it is ported to several

of the CPU architectures that Linux supports. In marketing the name "Red Hat Content Accelerator" is used in addition to TUX.

Architecture: TUX takes a different approach than kHTTPd to respond to network events, instead of using the normal socket interface from kernel mode and using non-blocking calls to examine the connections for activity TUX hooks in to the TCP/IP-stack by changing function pointers in the internal structure representing an socket to point to functions provided by TUX so that when a network event occur will TUX be called directly and can respond to it. The two most important functions that TUX hooks in to is `data_ready` that is called when data is received on the socket and `write_space` that is called when the TCP/IP-stack is ready to send more data. The functions `state_change` and `error_report` is also hooked by TUX. If the hardware supports it is zero-copy networking used by TUX to send data direct from the file system cache.

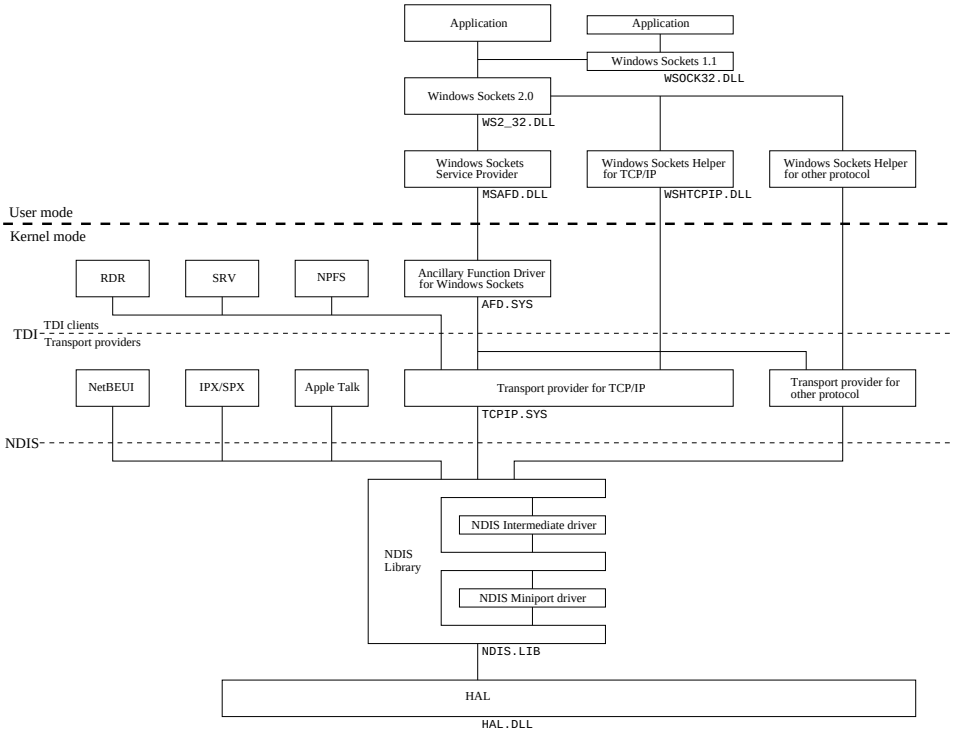
In addition to the different architecture to get higher performance TUX also differs from kHTTPd in that it has several advanced features. For example it supports caching of data from the user mode web server, it has an interface to be extended with modules, both in kernel mode and user mode, a module could interpret CGI scripts or serve a specific MIME type. Also it has full support for logging, support for virtual hosting, and can act as an FTP server.

Performance: An performance analyzes done by a research group at IBM[9] showed that the SPECWeb96 result for TUX was approximately twice that of kHTTPd witch in turn received twice the result of Apache. Looking at connection rate and throughput separately shows some interesting differences. Measuring connection rate TUX was only 10 to 20 percent faster than kHTTPd while both were approximately four times faster than Apache for small files, for big files there was less difference in connection rate. Instead measuring throughput for big files TUX performed three times better than kHHTPd that performed approximately equally to Apache. For small files there was less difference in throughput. The line between small and big files lies somewhere in the range of 32kB to 64kB. This shows that it is important to specify what to measure when comparing web server performance.

3 The Windows Networking Architecture

The Windows Networking Architecture is made of a number of layers with well defined interfaces between them. This makes it possible for modules

from different vendors to work together and lets the user install new modules that provide added functionality. The following figure shows the most important parts and how they are related.



The Windows Networking Architecture

Starting from bottom is the hardware with one or more "Network Interface Cards" (NICs). The software closest to the hardware on the Windows NT family of operating systems is the "Hardware Abstraction Layer" (HAL), it contains for example functions to access I/O-ports and handle interrupts. The purpose of HAL is to isolate the rest of the operating system from differences in the hardware among different computers within the same CPU architecture. The next layer above HAL is NDIS.

3.1 NDIS

The "Network Driver Interface Specification" (NDIS) was created to separate drivers for "Network Interface Cards" (NICs) from protocol implementations. The driver for a NIC is called an "NDIS miniport driver". The NDIS miniport driver implements the part that is specific to a NIC and uses the NDIS support library for the processing that is common to all NIC drivers.

The NDIS support library envelopes an NDIS miniport driver both above and below, it contains functions to access the hardware, on the Windows

NT family of operating systems will the NDIS support library use HAL to access the hardware. The interface to the transport providers above NDIS also goes through the NDIS support library, it is designed as a call back interface where the NDIS miniport driver registers functions that the NDIS support library will call when different events occur.

An NDIS miniport driver implements the interface specified by NDIS and is restricted to call a limited number of functions provided by the NDIS support library, this makes NDIS miniport drivers binary compatible among operating systems that implement the NDIS support library. For example Microsoft supports the use of the same NDIS miniport driver binary on Windows 9x and the Windows NT families of operating systems. It has even been created modules[10][11] for Linux that allows loading an NDIS miniport driver compiled for Windows in the Linux kernel and use it as a network driver for Linux.

There is also something called "NDIS intermediate drivers" that can be put above NDIS miniport drivers but below the transport providers, they filter the network traffic at a low level and can be used for example to implement a firewall, however it is also possible to install filter drivers above the transport provider, so called TDI filter drivers, to filter the network traffic at that level.

3.2 Transport providers

The implementation of a network protocol is called a "transport provider", the transport providers implement an interface called "Transport Driver Interface" (TDI) and uses the NDIS interface below them to talk to the network hardware. A transport provider can dynamically be connected to different network cards or other communication channels like IrDA or a SAN, this is called "binding" and is done by the system administrator. Example of transport providers included with Windows is TCP/IP, NetBEUI, IPX/SPX and AppleTalk.

3.3 TDI clients

The kernel mode interface to network protocols is TDI and the applications that uses it is called "TDI clients". TDI is the only network interface that is available to kernel mode code since Windows Sockets is only implemented for user mode. TDI is an asynchronous event driven interface that is very general so that it can be used for a wide variety of network types and protocols. Since the TDI interface is so general it has become abstract and is considered difficult to work with. One of the most important TDI clients is the Windows Sockets implementation that is described in more detail in the next section.

Examples of other TDI clients are RDR and SRV the client respective the server for the Windows network file system using the SMB protocol.

3.4 Windows Sockets

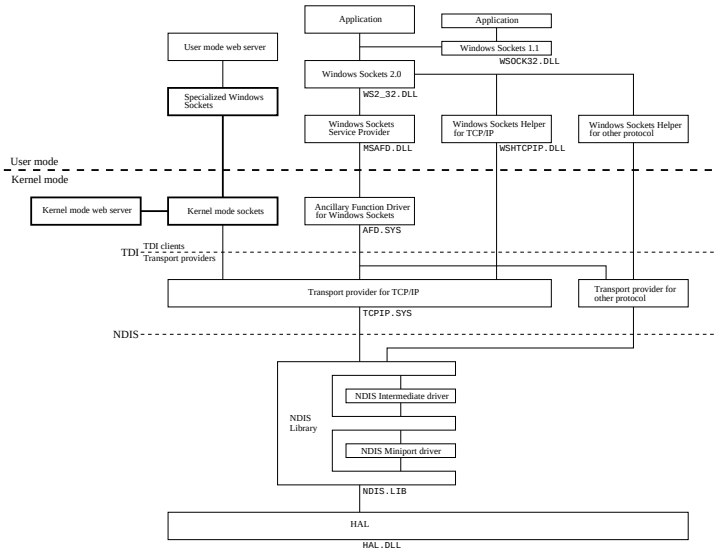
The most widely used network interface is called "BSD sockets" since it was first implemented on BSD UNIX. The BSD sockets interface contains functions like `socket()`, `connect()`, `send()` and `recv()`. Microsoft has defined a network interface for Windows that is compatible with BSD sockets, it is called "Windows Sockets". On older versions of Windows was Windows Sockets 1.1 used, it is a close implementation of BSD sockets with some extensions. Windows Sockets 2 is however a more general interface that can be extended to support other network protocols in addition to TCP/IP.

To provide a new protocol one implement something called a "Windows Sockets Helper" (WSH) DLL. When the WSH DLL has been registered in the system can applications request the new protocol when they create a socket handle and then use the ordinary Windows Sockets function to communicate using this handle. Windows Sockets will provide the common functionality and call functions in the WSH DLL for tasks that is specific to the new protocol. It is common that the new protocol also has its own transport provider in kernel mode and that the WSH DLL communicates with this. For example the TCP/IP transport provider is TCPIP.SYS and its WSH DLL is WSHTCPIP.DLL.

The kernel mode part of the Windows sockets implementation AFD.SYS is also a file system driver, the reason for this is that it makes it possible to let file objects and normal file handles represent sockets. Suppose that an application creates a socket and makes a connection using the functions `socket()` and `connect()`. Now if the application instead of using `send()` and `recv()` to transmit and receive data decides to use normal file I/O functions like `read()` and `write()` or hand over the handle to an application that don't know that the handle represents a socket. When the application calls `read()` or `write()` will the I/O manager know what file system driver the file object belongs to and call the read or write function that the file system driver has exported, since the file system driver for this file object is the driver implementing sockets can it let its read and write functions translate to recv and send, the socket handle can thus be used exactly as a normal file handle.

More information on the Windows networking architecture can be found in the MSDN Library[12] and the Windows DDK[13].

4 Implementation



Implementation

First this section describes the methods that where used to implement a prototype for Windows, following that are the components of the prototype described in more detail.

4.1 Methods

This section describes the methods that where used to implement a prototype for Windows.

4.1.1 Avoiding splitting sends

When a web server answers a HTTP GET request it first sends a header to the client followed by the file data. If one simply calls the TCP/IP-stack to send the header followed by one or more calls to send the file data it is likely that the header will be sent in its own network packet before the next data is given to the TCP/IP-stack. For small files this will double the number of network packets. An important optimization for web servers is therefore to send the header and the first file data in one packet.

On Linux this is accomplished with the `TCP_CORK` socket option, and in user mode on Windows can the `TransmitFile()` function be used, it takes both a file handle and a header and trailer to send together with the file data. However in kernel mode on Windows is non of these possibilities available,

instead is the TDI interface used. The TDI interface is not using normal pointers to memory areas to describe buffers but is instead using something called "Memory Descriptor Lists" (MDLs). An MDL is an internal structure used by Windows to describe the physical pages that back a virtual memory range. After allocating MDLs that represent the buffers containing the header and file data they can be linked together using the `Next` field, an example from the source code is showed here:

```
headerMdl->Next = fileMdl;
```

This is called an "MDL-chain". When the TCP/IP-stack is given the MDL-chain it can program the network card to do DMA directly from the different buffers that the MDL-chain describes.

4.1.2 Zero-copy networking

Eliminating the copy of data from the file system cache to the TCP/IP-stack when sending is called zero-copy networking. To truly get zero-copy networking the network card and its driver must support scatter/gather DMA and calculation of TCP and IP checksums in hardware. The TDI interface is using MDLs to describe the buffers to read data from and therefore must one get an MDL that represents the data in the file system cache. Simply creating a buffer and allocating an MDL for it would mean a copy operation, however the file system interface as defined by Windows includes the possibility to request an MDL that directly represents the physical pages in the file system cache.

This is done by setting the minor function code in the I/O-stack location to `IRP_MN_MDL` and the major function code to `IRP_MJ_READ`. When the read IRP completes is an MDL returned in `Irp->MdlAddress`. After use is this MDL returned to the file system driver using the minor function code `IRP_MN_COMPLETE_MDL`. The file system driver typically implements support for `IRP_MN_MDL` and `IRP_MN_COMPLETE_MDL` with help of the cache manager functions `CcMdlRead()` and `CcMdlReadComplete()`. The fast-I/O interface has the corresponding functions `MdlRead()` and `MdlReadComplete()`.

Note that when not setting the minor function code to `IRP_MN_MDL` it is possible to specify a pre-allocated MDL in `Irp->MdlAddress` that represents a buffer that the data will be read into, this is the "normal" way to do I/O using MDLs, however when specifying `IRP_MN_MDL` the `Irp->MdlAddress` is not initiated before calling the file system but will instead on complete of the call be set to an MDL allocated by the file system driver and directly representing the data in the file system cache.

As a curiosity it could be mentioned that there is minor function codes and

fast-I/O functions to read compressed files in raw format, that is without decompressing them, the HTTP protocol allows a web browser to notify the web server that it supports different compression algorithms, if a web browser where to support the compression algorithm used by the NTFS file system this could be used to decrease the amount of data transferred on the network without increasing the server load or storage requirement. A common way to implement support for compressed files in a web server is otherwise to store the files both un-compressed and compressed with one or more compression algorithms using different file name extensions, the web server can then choose the version of a file to send depending on what compression algorithms the client supports.

4.1.3 Handing over a request to user mode

Since the basic idea is to let a kernel mode web server handle some requests while others is handed over to a user mode web server an important question is how the hand over is done, this section describes different methods to do this.

HTTP redirect: The HTTP protocol allows a web server to send a redirect response to a request from a web browser, the browser then does a new request from another location, the redirect response could be used for the requests that should be handed over to the user mode web server, for example by changing the port number. However this would double the number of network connections needed and increase the load on the server. Because of the low performance this method is not used in any existing project as far as known but an advantage is that it is easy to implement and it could be used in a prototype during experimentation.

Socket to socket copy: When the kernel mode web server finds that a request should be handed over to the user mode web server it could simply copy the data between the socket belonging to the remote client and a socket belonging to the user mode web server, possible creating a new thread for each connection or using non-blocking socket calls. This method was used in the first versions of kHTTPd but to increase performance it was soon replaced with the method described in the next paragraph. An advantage of this method is that it is rather simple to implement, it could be used for the prototype on Windows but one would need to investigate the affects on performance.

The Linux way: Newer versions of kHTTPd and TUX use a particular efficient method to hand over a request to the user mode web server, they directly enters it in the accept queue for the socket it is listening on, this can be done since a kernel module in Linux has access to data structures and functions internal to the operating system and the source code is available. When reading the HTTP request the kernel mode web server uses the flag `MSG_PEEK` that returns the arrived data without removing it from the buffer in the TCP/IP-stack, if the request is to be handed over to the user mode web server the data is still there to be read again. After the request is moved to the new socket is the original socket empty and the process waiting for a connection on the new socket will be waken up as if the connection had been done to that socket from the beginning. Of the methods described this one has the highest performance since it avoids any extra copying of data and is transparent to the system, however it can only be done if the source code for the operating system is available like it is for Linux. It is not possible to modify the Windows Sockets implementation to use this method.

Replacing the original Windows Sockets implementation: The method described in the last paragraph to hand over a request from the kernel mode web server to the user mode web server is the most efficient one, the reason it can not be used on Windows is that the source code to the Windows Sockets implementation is not available, however it is still possible to solve the problem based on this method by replacing the Windows Sockets implementation with one that is aware of the relation between the kernel mode web server and the user mode web server. This specialized Windows Sockets implementation would hand over a request to the user mode web server if it can not be processed by the kernel mode web server. The user mode web server is like other network applications dynamically linked to a DLL that implements the Windows Sockets API, as is described in one of the following sections it is possible to replace this DLL for only the user mode web server while other network applications will continue to use the ordinary implementation. Advantages with this method is that it will probably be as efficient on Windows as on Linux and that the user mode web server can be unmodified, an disadvantage is that it is a lot of work to implement it.

A new protocol family with a Windows Sockets Helper DLL: Windows Sockets version 2 is a general interface that can be extended to support new protocols. A new protocol is implemented in two parts, one kernel mode transport provider and one Windows Sockets Helper DLL. To use the new protocol an application specify it as the parameter called “protocol family” or “address family” to the `socket()` function. The common processing will be done by the ordinary Windows Sockets implementation while it will call

the Windows Sockets Helper DLL to perform functions that is specific to the new protocol. The advantage of this design is that it makes it easy to add a new protocol and that the applications can use it with the standard Windows Sockets API. It would be possible to use this method to implement a way for the kernel mode web server and the user mode web server to communicate, if one let the new protocol only adds this functionality to the existing TCP/IP protocol this method might be somewhat easier than the method described in the last paragraph, however one drawback is that the user mode web server would need to be modified so that it requests the new protocol when it creates its socket.

4.1.4 Replacing a system DLL

The user mode web server and other network applications use the Windows Sockets interface for network communication, this section describes how to let the user mode web server use a specialized Windows Sockets implementation that communicates with the kernel mode web server. An application that is using Windows Sockets is linked to a DLL that implements the Windows Sockets API, for Windows Sockets version 2 is the DLL named `WS2_32.DLL` while Windows Sockets version 1.1 is implemented by a DLL named `WSOCK32.DLL` that to the most part forwards the calls to the corresponding function in `WS2_32.DLL`.

When an application that is linked to a DLL is loaded will the system search for the corresponding DLL in certain places in order and loads the first instance that is found. The search order is:

1. The directory from which the application is loaded.
2. The current directory.
3. The system directory. (typical `c:\windows\system32\`)
4. The 16-bit system directory. (typical `c:\windows\system\`)
5. The Windows directory. (typical `c:\windows\`)
6. The directories that are listed in the PATH environment variable.

This makes it possible to replace a system DLL for a specific application by placing a new DLL with the same name in the applications directory. Then that application will use the new DLL while other applications will continue to use the old DLL in the system directory.

Note that on Windows XP and Windows Server 2003 is there a new registry entry called `HKLM\System\CurrentControlSet\Control\Session Manager\`

`SafeDllSearchMode` to change the search order, if set to "1" the current directory is searched after the system and Windows directories, but before the directories in the PATH environment variable. On Windows XP SP1 and Windows Server 2003 is this registry entry set to 1 as default.

If this specialized Windows Sockets only re-implement some of the functions, or only pre- or post-process them, it is convenient to be able to call the original DLL, this can be done with help of the functions `LoadLibrary()` and `GetProcAddress()`. An example shows how this can be done for the function `gethostname()`, first is a function pointer and a variable to store a pointer to the original function declared:

```
typedef int (__stdcall *pf_gethostname)(char *name, int namelen);
static pf_gethostname win_gethostname;
```

At the initialization of the DLL is a pointer to the original function requested using the functions `LoadLibrary()` and `GetProcAddress()`:

```
win_gethostname = (pf_gethostname)
GetProcAddress(LoadLibrary("c:\path\filename.dll"), "gethostname");
```

Then can a call to `gethostname()` easy be redirected to the same function in the original DLL:

```
int __stdcall gethostname(char *name, int namelen)
{
    return win_gethostname(name, namelen);
}
```

An important point to keep in mind when replacing a DLL is that an application can not only link to functions by name but also by number, this number is called "ordinal". To let a function have the same ordinal in the new DLL as in the one being replaced one must create a "Linker Definition File", a part of it defining the ordinal for the function `gethostname()` is shown here:

```
EXPORTS
    ...
    gethostname @57
```

4.1.5 Socket handles as file handles

The socket handles used by Windows Sockets is normal file system handles, this means that an application can call file I/O functions like `read()` and

`write()` with the socket handle and the I/O manager will redirect the call to the driver that the corresponding file object belongs to. Since this is the driver implementing the kernel mode part of Windows Sockets (AFD.SYS) it can translate the read or write call to recv and send respectively.

Windows provides functions to help the implementer of a new protocol to request and manage this kind of handles, these functions are `WPUCreateSocketHandle()`, `WPUQuerySocketHandleContext()` and `WPUCloseSocketHandle()`.

It is also possible for a new protocol to implement this feature by acting as a full file system driver, however that is a large and difficult implementation to do.

4.2 Prototype

This section describes the components of the prototype.

4.2.1 Simple kernel mode sockets

A major part of the project was to implement the kernel mode sockets library. This was difficult for two reasons, first it is more difficult to develop and debug drivers than applications, especially on Windows and second the interface to network communication from kernel mode is in large part undocumented. The TDI interface in itself is documented[13] but this documentation is directed to implementers of new network protocols, there is almost no documentation for developers of TDI clients and most notably is the existing protocols like TCP/IP undocumented. During the work I had good help from postings on the Internet by Gary Nebbett and Maxim S. Shatskih.

The TDI interface is a standard driver interface based on IRP's. To use the TDI interface one opens a named device object representing the protocol off choice. The open operation will create a file handle and a file object is requested with `ObReferenceObjectByHandle()`. This file object can then be used in further calls to the TDI interface to identify the "socket". To make a call an IRP is created with `TdiBuildInternalDeviceControlIrp()` or `IoAllocateIrp()` and initialized with the help of one of the `TdiBuildXxx()` macros. The IRP is then sent to the transport driver using `IoCallDriver()` and it is possible to either wait on an event for it to complete or specify an completion routine that is called when the request completes.

The TDI interface uses two kinds of file objects; representing transport addresses respective connection endpoints, a transport address is always needed while a connection endpoint is used for TCP sockets, a connection endpoint

is associated with a transport address and a transport address can be associated with more than one connection endpoint.

It is also possible to register different event handlers on a transport address, when a network event occur is the corresponding event handler called, since the event handlers is called at DPC level what they are allowed to do is limited.

Following is comments on my experiences trying to implement a simple socket library for kernel mode on Windows using the TDI interface. It is not complete and focus on what is needed by the web server prototype but it includes some functionality not used by the web server prototype like `connect()` and UDP sockets.

Creating a socket:

```
int __cdecl socket(int af, int type, int protocol);
```

A socket is created with the function `socket()`, it takes parameters to specify the kind of socket, the only kind of interest for a web server is stream sockets using the TCP protocol. The return value of the socket function is a handle that is used as a parameter to the functions operating on a socket.

There is no need to call the TDI interface at the time of creating a socket, instead this function just allocates an internal data structure to keep information about the socket and returns a handle to it. To keep this implementation simple I decided to use minus the address of the internal struct as a "handle" to the socket, using the address of the struct as a handle makes it easily available and the reason for using minus the address is that since a handle is 32-bit and addresses of memory allocated in kernel mode on Windows is above 2GB, minus the address will be a positive number when casted to a signed integer while negative numbers can be used as error codes. Off cause this method should be replaced by using "real" handles, perhaps using the RtlGenericTable functions or even using file system handles.

Binding the socket to a local address and port number:

```
int __cdecl bind(int socket, const struct sockaddr *addr,
int addrlen);
```

For a server the next step is usually to associate the socket with a local port number and possible an address, this is called binding.

The `bind()` function will call the TDI interface to do two things, create a transport address and possible set up one or more event handlers. The

file handle and file object representing the transport address is saved in the internal socket struct for later use. The only event handler needed to be set in my implementation was for the disconnect event.

Making an connection:

```
int __cdecl connect(int socket, const struct sockaddr *addr,
int addrlen);
```

A web server does not need the `connect()` function but I implemented it anyway for completeness and it is shortly described here. If the socket is not bound will the `connect()` function begin by doing an implicit bind, this means that the TCP/IP-stack will assign a free port number to it. Then is something called a connection endpoint created, for information on what this is see the `listen()` function. Finally is a `TDI_CONNECT` IRP sent to the TDI interface.

Listen for connections:

```
int __cdecl listen(int socket, int backlog);
```

The `listen()` function is used by a server to tell that a socket should listen for connection attempts. A connection can then be accepted using the `accept()` function.

To understand how `listen()` and `accept()` relate to the TDI interface and implementing them was the most difficult part of the socket library.

The TCP/IP-stack will listen for connection attempts either if a `TDI_LISTEN` call is made or if a connect event handler is installed on the transport address. The `TDI_LISTEN` call will complete when a connection attempt is made, if the flags are 0 is the connection already accepted at that time so no `TDI_ACCEPT` call is needed, if the flags are `TDI_QUERY_ACCEPT` one can either make a `TDI_ACCEPT` call to accept the connection attempt or a `TDI_DISCONNECT` call to deny it. The connect event handler builds an IRP that will be sent to the TDI interface, it can either be a `TDI_ACCEPT` IRP to accept the connection attempt or a `TDI_DISCONNECT` IRP to deny it.

In my simple implementation I did not use an connect event handler but instead let the `listen()` function only create the connection endpoint, the major part of the work is then done by the `accept()` function. This means an deviation from the BSD socket standard since the socket wont be listening until `accept()` is called but this wont affect the prototype developed in this

project. Another disadvantage with this method is that it corresponds to a fixed backlog of only 1.

In a more elaborate implementation one would instead let `listen()` create a number of connection endpoints and install an connect event handler on the transport address.

Accepting an connection:

```
int __cdecl accept(int socket, struct sockaddr *addr, int
*addrlen);
```

The `accept()` function accepts an connection and returns a new socket that represents it while the old socket can be used again to accept more connections.

In my implementation does `accept()` start by issuing `TDI_LISTEN` with flags set to 0, this means to wait for a connection attempt and accepting it. When the TDI interface is used in this way is `TDI_ACCEPT` never needed. After the `TDI_LISTEN` call completes does my implementation create and initiate a new socket, the connection endpoint is associated with this socket since it is now connected to the remote node. Finally is a new connection endpoint created and associated with the old socket so it can be used to accept more connections.

In the more elaborate implementation suggested in `listen()` above the `accept()` function would instead wait for an event to be signalled by the connect event handler that does the most part of the work. The connect event handler would do two things, build an `TDI_ACCEPT` IRP to accept the connection attempt and create a new connection endpoint to replace the one being used so that the backlog still has the same size. The event that the `accept()` function is waiting on could be signalled by the completion function for the `TDI_ACCEPT` IRP. Note that since the connect event handler runs at DPC level it can not create a new connection endpoint from within it but must instead schedule a work-item to do that.

However I stayed with the simpler implementation in this project.

Sending data:

```
int __cdecl send(int socket, const char *buf, int len, int
flags);
int __cdecl sendto(int socket, const char *buf, int len, int
flags, const struct sockaddr *addr, int addrlen);
```

The functions to send data can be a bit tricky to implement, for example the TCP protocol provides a reliable byte stream and might need to re-transmit a packet if it is lost on the network, there fore doesn't the `TDI_SEND` request completes as soon as the data has been scheduled to be sent, in the case a re-transmit is needed must the buffer remain available to the TCP/IP-stack, instead it will complete after an acknowledge has been received for all packets that were used to send the data in the buffer. However letting `send()` block until then would mean an inefficiency if the user does several small sends after one another, therefore a natural implementation is to let `send()` copy the data to an internal buffer and sending it from there instead so that `send()` can return immediately, only when the internal buffer is full would `send()` block. Since the web server prototype doesn't use `send()` at all but instead a new function specialized to send file data I didn't implement this buffering in the normal `send()` but left it as it is. When sending UDP data this shouldn't be an issue since UDP doesn't use re-transmit.

To send file data as efficiently as possible an MDL representing it in the file system cache is requested and the header and the first data is put together to a chained MDL. To send this I created a send function that takes an MDL instead of buffer and length parameters. The MDL is given to the TDI interface to be sent where after the function returns. A completion routine specified by the caller is run when the call completes to de-allocate the buffer and MDL.

A question that is raised since the send call is non-blocking is how many `TDI_SEND` requests to have outstanding, the documentation says that a send request may return `STATUS_DEVICE_NOT_READY` and that there is an event handler "send possible" to be called when the TDI interface is ready to send again. However for the file sizes used during performance measurements on the web server I didn't had any problems with the send implementation so I left this question unanswered.

Receiving data:

```
int __cdecl recv(int socket, char *buf, int len, int flags);
int __cdecl recvfrom(int socket, char *buf, int len, int flags,
struct sockaddr *addr, int *addrlen);
```

The functions to receive data was more straight forward to implement than for sending, in my implementation I just create an MDL for the users buffer, make a call to the TDI interface and wait for it to complete. I didn't find the need to implement MDL versions of the receive functions for the web server prototype but that could be done to make the socket library more complete.

The TDI interface allows registering an event handler that is called when data is received, doing that wasn't needed in this project but in the conclusions section I suggest that performance could be improved by analyzing and responding to a HTTP request from an receive event handler, however if sending file data from an event handler it must be cached in non-paged memory.

Getting and setting socket options:

```
int __cdecl getsockopt(int socket, int level, int optname, char
*optval, int *optlen);
int __cdecl setsockopt(int socket, int level, int optname, const
char *optval, int optlen);
```

The functions `getsockopt()` and `setsockopt()` is used to get and set options on a socket, the prototype didn't need the use of any particular socket option so I left them unimplemented.

Query the local or remote address and port number:

```
int __cdecl getsockname(int socket, struct sockaddr *addr, int
*addrlen);
int __cdecl getpeername(int socket, struct sockaddr *addr, int
*addrlen);
```

The function `getsockname()` returns the address and port number of the local socket if it is bound. This information can be requested from the TDI interface using the minor function code `TDI_QUERY_INFORMATION` and the query type `TDI_QUERY_ADDRESS_INFO`. When first trying this call I noticed that the address returned was always "0.0.0.0" while the port number was correct. The documentation says:

> `TDI_QUERY_ADDRESS_INFO` specifies that the transport should return the information, formatted as a `TDI_ADDRESS_INFO` structure, in the client-supplied buffer mapped at MdlAddr. FileObj must point to an open file object representing a transport address or a connection endpoint already associated with a transport address.

Note that it doesn't say if it matters if a file object representing a transport address or a connection endpoint is used. After experiencing that the

address returned was always 0.0.0.0 I tried using the file object representing a connection endpoint instead of the transport address and indeed the address was correctly returned! This led to the following implementation of the function `getsockname()` for TCP sockets:

```
else if (s->type == SOCK_STREAM)
{
    *addrlen = sizeof(struct sockaddr_in);

    return tdi_query_address(
        s->streamSocket && s->streamSocket->connectionFileObject ?
        s->streamSocket->connectionFileObject : s->addressFileObject,
        &localAddr->sin_addr.s_addr,
        &localAddr->sin_port
        );
}
```

Note that if the user calls `getsockname()` after `bind()` but before `connect()` or `listen()` the socket has a port number but no connection endpoint, therefore one must query the transport address in this case.

The function `getpeername()` returns the address and port number of the remote node, this is always known and does not require a call to the TDI interface, it is saved in the private socket struct either when connecting to a given address or when accepting an connection attempt at witch time it is provided by the `TDI_LISTEN` call or the connect event handler.

Functions to convert between host and network byte order:

```
u_long __cdecl htonl(u_long hostlong);
u_short __cdecl htons(u_short hostshort);
u_long __cdecl ntohl(u_long netlong);
u_short __cdecl ntohs(u_short netshort);
```

Computers use either big endian or little endian byte order while Internet addresses and port numbers always is transferred in big endian byte order. The functions `htonl()` and `htons()` is used to convert between the hosts byte order and network byte order for long and short integers respectively, the functions `ntohl()` and `ntohs()` is used to transfer between network byte order and the hosts byte order but off cause they do the same thing as the former functions. For a big endian system this functions do nothing but since the x86 CPU is using little endian byte order they should be implemented to swap the bytes, this can be done as a pre-processor macro, anyway the TDI interface is not involved.

Functions to convert between ASCII and integer representation of addresses:

```
u_long __cdecl inet_addr(const char *name);
int __cdecl inet_aton(const char *name, struct in_addr *addr);
char * __cdecl inet_ntoa(struct in_addr addr);
```

These functions is used to convert between ASCII and integer representation of addresses, like the byte order functions they are straight forward to implement and does not require the use of the TDI interface, however one issue to mention is that `inet_ntoa()` returns a pointer to an string, if this is a statically allocated string witch is common in user mode will the implementation not be thread safe, on the other hand if it is dynamically allocated like I choose to do it must be de-allocated by the caller after use witch is an incompatibility from the BSD socket standard.

Shutting down a socket:

```
int __cdecl shutdown(int socket, int how);
```

I implemented the function `shutdown()` to call `TDI_DISCONNECT` with the flags set to `TDI_DISCONNECT_RELEASE`, for more information on this request see `close()`.

Closing a socket:

```
int __cdecl close(int socket);
```

It is almost optional to use event handlers but in fact for TCP sockets there is one event handler that must be used and that is the disconnect event. To properly close a TCP connection and be sure that all data sent is transferred to the remote node one must begin by calling `TDI_DISCONNECT` with the flags set to `TDI_DISCONNECT_RELEASE`, however waiting for this IRP to complete and then closing the file objects representing the transport address and the connection endpoint might close them prematurely and data is lost, instead one must wait for the disconnect event to be called witch it is after the TCP protocol has negotiated the closing of the connection.

Note that the flag `TDI_DISCONNECT_RELEASE` does a “graceful close” but it also exists a flag `TDI_DISCONNECT_ABORT` to do a “hard close” of the socket, in this case will the disconnect event handler not be called. However when doing a graceful close but the remote node does a hard close will the disconnect

event handler be called, the flags parameter to the disconnect event handler tells what kind of close the remote node did.

In my implementation I choose to install the disconnect event handler when creating the transport address in `bind()` and send `TDI_DISCONNECT` from either `shutdown()` or `close()` and then finally wait for an event to be signalled from the disconnect event handler. After this wait is the file objects representing the transport address and the connection endpoint de-referenced and their handles closed. The `TDI_DISCONNECT_ABORT` flag was not needed.

DNS lookup functions:

```
struct hostent * __cdecl gethostbyaddr(const char *addr, int
addrlen, int type);
struct hostent * __cdecl gethostbyname(const char *name);
int __cdecl gethostname(char *name, int namelen);
struct protoent * __cdecl getprotobyname(const char *name);
struct protoent * __cdecl getprotobynumber(int number);
struct servent * __cdecl getservbyname(const char *name, const
char *proto);
struct servent * __cdecl getservbyport(int port, const char
*proto);
```

This collection of functions is used to query the DNS, for example to get the Internet address of a named host. Since this is not needed by the web server prototype all of these functions is un-implemented however I will comment on how to implement them.

A straight forward method is to just implement the functions using the socket library, the DNS query protocol is simple and there should be no problems implementing it for kernel mode on Windows. Another possibility is to implement them with the help of a user mode service process that the kernel mode part communicates with. The user mode process can then forward the call to the ordinary Windows Sockets API and finally return the result to the kernel mode sockets library. An advantage of this method is that if the ordinary Windows implementation is changed or re-configured will the kernel mode sockets library automatically make use of this.

The kernel mode socket implementation compiles to a static library that a driver can link to.

4.2.2 pthreads

"pthreads" or the "POSIX Threads API" is a standard that defines how to create multiple threads in a process and how to synchronize and communicate between them. To support developing the prototype I implemented a simple pthread library for Windows drivers limited to the functions used by the prototype. It had off cause been possible to let the prototype use the Windows API directly but an advantage with this method is that it makes it easy to borrow code between Windows and UNIX and user mode and kernel mode while developing the prototype and experimenting with different solutions.

Only the functions to create a thread, terminate a thread and wait for a thread to terminate was needed. Following is comments on their implementation.

Create a thread:

```
int __cdecl pthread_create(pthread_t *thread, const pthread_attr_t
*attr, void *(*start_routine)(void*), void *arg);
```

The function `pthread_create()` creates a new thread using `PsCreateSystem Thread()` and optionally saves a handle to it if the parameter `pthread_t *thread` is non-NULL. If a handle to the thread is saved it must be waited on with the function `pthread_join()` or closed with `ZwClose()`.

Terminate a thread:

```
void __cdecl pthread_exit(void *value_ptr);
```

The function `pthread_exit()` terminates the current thread by calling `PsTerminateSystemThread()`. It is also possible for a thread to terminate by simply returning from its main function.

Wait for a thread to terminate:

```
int __cdecl pthread_join(pthread_t thread, void **value_ptr);
```

The function `pthread_join()` waits for a given thread to terminate using `ZwWaitForSingleObject()`, if the threads exit status is requested it is queried using `ZwQueryInformationThread()`. If one has saved an handle to a thread it must be waited on with this function or closed with `ZwClose()`.

The pthread implementation compiles to a static library that a driver can link to.

4.2.3 Kernel mode web server

The kernel mode web server developed is a prototype intended to be used for performance measurements and evaluation of different methods therefore it only implements the HTTP GET request. It uses the socket and pthread library described above. Following is a description of its implementation.

Web server: The main part of the web server is simple, the `DriverEntry()` function creates a socket and a thread that waits for connections to it. For every connection is separate thread created to handle the request, the thread will read the HTTP request and if it is a HTTP GET request and the requested file exists will it be sent using `send_file_mdl()`. Other HTTP requests is not supported, there is also limited support for the different HTTP headers that can help web browsers and proxy servers to be more efficient.

File-I/O: The most interesting part of the web server prototype is the file-I/O functions since they use the MDL interface to utilize zero-copy networking.

```
HANDLE dopen(PUNICODE_STRING dirName);
```

The function `dopen()` uses `ZwCreateFile()` to open a handle to a directory.

```
HANDLE fopen(char *fileName, HANDLE rootDir);
```

The function `fopen()` uses `ZwCreateFile()` to open a handle to a file. A nice feature of `ZwCreateFile()` that is used here is the possibility to do a so called "relative open", this means that a file handle to an already opened directory is given, the path and file name is then interpreted to be relative to this directory. This is used by the web server to open the requested file relative to the directory that has been specified as the document root.

Another thing worth to mention here is that I specify the flag `FILE_SEQUENTIAL_ONLY` to inform the cache manager that the file will be read sequentially, from the beginning to the end, the cache manager can then use this information to optimize the caching of the file.

```
int fread(HANDLE fileHandle, void *buf, int len);
```

The function `fread()` reads file data with `ZwReadFile()` to a buffer, it is included here only for demonstration and comparison since the web server instead should use `fread_mdl()`.

```
int fread_mdl(PFILE_OBJECT fileObject, PLARGE_INTEGER offset,
ULONG len, PMDL *mdl);
```

The `fread_mdl()` function uses the special MDL interface to the file system to request an MDL representing the data in the file system cache. It allocates at initializes a normal read IRP but sets the `MinorFunction` to `IRP_MN_MDL` and leave the `Irp->MdlAddress` un-initialized. When the call completes has the file system driver set `Irp->MdlAddress` to an MDL that is returned to the caller.

```
int fread_mdl_complete(PFILE_OBJECT fileObject, PMDL mdl);
```

The function `fread_mdl_complete()` is used to return (de-allocate) an MDL requested with `fread_mdl()`, it sets the `MinorFunction` to `IRP_MN_COMPLETE_MDL` and `Irp->MdlAddress` to the MDL that should be returned to the file system.

```
void fclose(HANDLE fileHandle);
```

The function `fclose()` uses `ZwClose()` to close the file handle.

```
int send_file(int sock, char *header, HANDLE fileHandle);
```

The function `send_file()` uses `fread()` and `send()` to send a header followed by file data to a socket. It is only included for demonstration and comparison with `send_file_mdl()`.

```
int send_file_mdl(int sock, char *header, HANDLE fileHandle);
```

The function `send_file_mdl()` uses `fread_mdl()` to request an MDL representing the file data, for the first block it links the file data MDL to an MDL for the header. Then the MDL is sent and an completion routine specified to de-allocate it with `fread_mdl_complete()`. This is repeated in a loop until all of the file data has been sent.

```
static void send_mdl_complete(int status, void *context);
```

The function `send_mdl_complete()` is run at DPC level when the send request has completed and the TDI interface doesn't need the buffer any more, it queues a work item to do the de-allocation since that is not allowed at DPC level.

```
static void work_item(void *context);
```

The function `work_item()` is finally run and uses `fread_mdl_complete()` to return the MDL to the file system and de-allocates other resources used.

5 Performance measurements

This section describes how to measure performance of web servers and the results from measuring the prototype and other web servers for Windows and Linux.

5.1 Method

5.1.1 What to measure

The most common parameters to measure is latency (time to complete a request) and throughput (bytes/second), it is also common to study the web servers performance for different file sizes, for small files is the most time spent on setting up the TCP connection and process the request so the number of requests per second gives good information on the performance, for bigger files is the performance more affected on the work of reading file data from the file system and sending it to the TCP/IP-stack so throughput will better show the difference between web serves here.

When the Standard Performance Evaluation Corporation (SPEC) developed their SpecWeb benchmark they analyzed log files from different web servers to determine the file size distribution of the requests handled by a typical web server. The result is listed in table 5 and shows that rather small files is most common.

Files less than 1KB	35%
Files between 1KB and 10KB	50%
Files between 10KB and 100KB	14%
Files between 100KB and 1000KB	1%

Table 5: File size distribution of requests to a typical web server

Another thing to measure is how the web server handles a large number of simultaneous connections, this will show if the connections is handled in an efficient way or if more and more processing time goes to just administrating the connections. One should not forget that even idle connections could impose a load on the server.

Related to this is testing the web servers overload behaviour, this means to connect a larger number of clients making more requests than the web server can handle. Depending on the design of the web server can two things happen, ideally will many of the clients be processed rather fast while others get connection timeout or connection refused, this means that the server will make progress and that the load will eventually decrease again. On the other hand it could also happen that the new requests fill up the servers resources and make the processing slower and slower for all clients until the server appears to have locked up, this condition is called "receive livelock"[14].

5.1.2 Benchmarks

This section describes the most common benchmarks for web servers.

httperf: The httperf benchmark[15] is developed by David Mosberger at Hewlett-Packard Research Labs. The httperf benchmark is a flexible HTTP client that requests a file from a web server a number of times and possible from a number of parallel threads and then prints out detailed statistics. The most important statistics is the number of requests per second and the network throughput but httperf also gives timing information with min and max limits and information on the errors if any. An example of the output from httperf is shown below. Httperf does not use any particular file set, it is left to the tester to choose them. A strength of httperf is that it can measure how the web server can handle a load with a large number of concurrent connections and records any timeouts or other errors that occur. The httperf benchmark is available for free in source code form and can be compiled for most versions of UNIX.

```
$ httperf --timeout=60 --server=192.168.1.1 --port=80 --uri=/www/64.jpg
--num-conns=1000
Maximum connect burst length: 1

Total: connections 1000 requests 1000 replies 1000 test-duration 0.527 s

Connection rate: 1897.3 conn/s (0.5 ms/conn, <=1 concurrent connections)
Connection time [ms]: min 0.5 avg 0.5 max 46.2 median 0.5 stddev 1.4
Connection time [ms]: connect 0.1
```

```
Connection length [replies/conn]: 1.000

Request rate: 1897.3 req/s (0.5 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)
Reply time [ms]: response 0.3 transfer 0.0
Reply size [B]: header 45.0 content 64.0 footer 0.0 (total 109.0)
Reply status: 1xx=0 2xx=1000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 0.18 system 0.34 (user 34.2% system 64.5% total 98.7%)
Net I/O: 335.4 KB/s (2.7*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
$
```

WebStone: The WebStone benchmark[16] was originally developed by Silicon Graphics and later acquired by Mindcraft. The WebStone benchmark measures throughput and latency for a number of files with different sizes. A standard file set is included with WebStone but it is also possible to use a custom set, if using the standard file set can the result be compared with others, MindCraft collects and publish such results on there web site. WebStone is divided in two parts, a "Webmaster" runs a control program that distribute a benchmark program to a number of "Web clients" using `rexec`. After the benchmark is run will the Webmaster combine the performance results from all the Web clients into a single summary report. The WebStone benchmark is available for both Windows and different versions of UNIX, it is free and the source code is included.

SpecWeb: The SpecWeb benchmark[17] is developed by the Standard Performance Evaluation Corporation (SPEC), it is a non-profit corporation that makes a number of benchmarks for different things. SPEC also review and publish submitted results of its benchmarks for comparison on their web site. The SpecWeb benchmark is named after the year it is made, two versions is in use, SpecWeb96 and SpecWeb99, while SpecWeb2005 is currently under development. The evolution of SpecWeb mostly concerns generating new kind of workloads and support of dynamic content. The SpecWeb benchmark is an integrated test that produces a load that is supposed to closely model real-world applications, the result of the SpecWeb benchmark is a single number. The SpecWeb benchmark is available for both Windows and different versions of UNIX, it is not free but source code is included if

one buys a license. The retail version of SpecWeb99 cost $800 and a license for non-profit or educational use cost $200.

5.1.3 Things to consider

This section discusses things to consider when measuring web server performance.

The network speed: To get a valid measurement result it is important to avoid that the network speed is a bottleneck but that instead it is the server's hardware and software that sets the limit. It is mostly when measuring throughput on larger files that this may become an issue, if the network speed is to low the graphs for the different web servers will come closer to each other and approach the theoretical maximum. My measurements showed that even if the pattern was the same the spread among the web servers was bigger on 100Mb/s Ethernet than on 10Mb/s Ethernet.

The presence of an anti-virus program: At the first measurements I received rather low performance on Windows compared to Linux, therefore I investigated the system to see if there was any difference in the setup that could cause problems on Windows. Among other things I checked that the network card was fully supported by the device drivers on both Linux and Windows and that it was run in the same mode; duplex, DMA and calculation of TCP checksums in hardware. After some investigation I noticed that Norton Antivirus was installed and running on the Windows system, disabling it increased the performance with 20% to 25%, a significant change. The reason for this is that Norton like other antivirus programs installs a file system filter driver that hooks into the file system operations, when a read request is done by an application the file system filter will scan the file data for viruses before returning the data to the application, this will introduce an performance degradation. However there have existed antivirus programs that instead scanned the file content on every open request, they reduced the performance a lot more! To do a fair comparison between Linux and Windows it is important to check for these kinds of differences.

The maximum number of open file descriptors: To benchmark the web servers I used httperf and run it on Linux. A limitation that I noticed during testing with a large numbers of simultaneous connections was the number of open file descriptors allowed, a socket handle is a file descriptor. The maximum number of open file descriptors has changed during the versions of the Linux kernel, on the system I used it was 8192, the following

command shows how to tell the limit:

```
$ cat /proc/sys/fs/file-max
8192
```

The following command run as root increases the limit to 65535:

```
# echo "65535" > /proc/sys/fs/file-max
```

Unfortunately the maximum number of open file descriptors is also compiled into programs using the `select()` system call like httperf with a limit of only 1024. To correct this one must edit the line

```
#define __FD_SETSIZE 1024
```

in the files /usr/include/bits/types.h and /usr/include/linux/posix_types.h to the new value and recompile the program.

Depending on the system configuration it might also be needed to increase the per user limit of open file descriptors, for sh this is done with the command:

```
# ulimit -n number
```

and for csh with the command:

```
# limit descriptors number
```

A related problem is that if a program doesn't specify an local address and port number when calling `bind()` is the port number allocated from a limited range, by default the range is 1024 to 4999, allowing 3975 simultaneous outgoing connections. This range can be viewed with the command:

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
1024 4999
```

The following command run as root increases the range up to 65535:

```
# echo "1024 65535" > /proc/sys/net/ipv4/ip_local_port_range
```

The TIME_WAIT state: When a TCP connection is closed does the protocol go into a state called "TIME_WAIT" that lasts for 60 seconds or twice the maximum segment lifetime. One of the reasons for this is to avoid that duplicate packets who has not arrived yet will be mistaken for valid data on a new connection using the same port. However since the port number is a 16-bit integer there can only be at most 65536 connections at the same time and a port will be unavailable for a minute after closing the connection. This may become a bottleneck both on the client side where the benchmark program is run and on the server side when measuring a high connection rate for a long time, one workaround is to only make short measurements for small files that will give the highest connection rate, it is also possible to avoid or shorten the TIME_WAIT state but that means introducing an incompatibility with the TCP/IP-protocol.

The maximum listen backlog: The function `listen()` takes a parameter called "backlog" that specifies how many connections can be pending waiting for `accept()`. This is the number of connections that may arrive while the server processes an accepted request, possible by creating a new thread to handle the client, before it calls `accept()` again. If more clients than the backlog number try to connect during this time they will get connection refused. The web server can specify a backlog number when it calls `listen()` but the operating system usually has some limit and will not use a backlog number above this. The maximum listen backlog for different environments is listed in table 6 .

Windows Sockets on client versions of Windows	5
Windows Sockets on server versions of Windows	200
TDI in kernel mode on Windows	Decided by programmer
Linux	128

Table 6: Maximum listen backlog

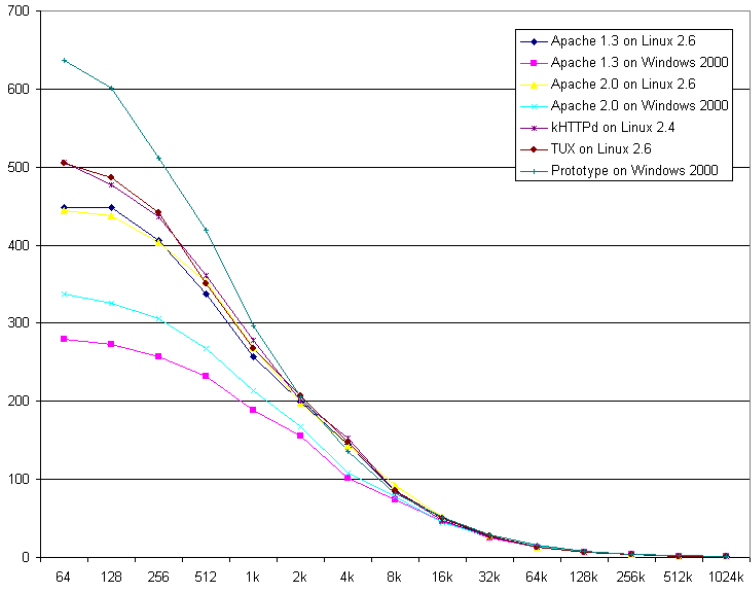
5.1.4 Measurement setup

The measurements were done in a lab at the department, the computers are equipped with AMD Athlon 1133MHz CPU and 128MB RAM. They are configured for dual boot and can either run Windows 2000 Professional or Gentoo Linux[18]. When doing the measurements the computers where connected to a separate Ethernet switch. As benchmark program httperf where used because of its flexibility, it was run on Linux while the server computer either ran Linux or Windows depending on witch web server to

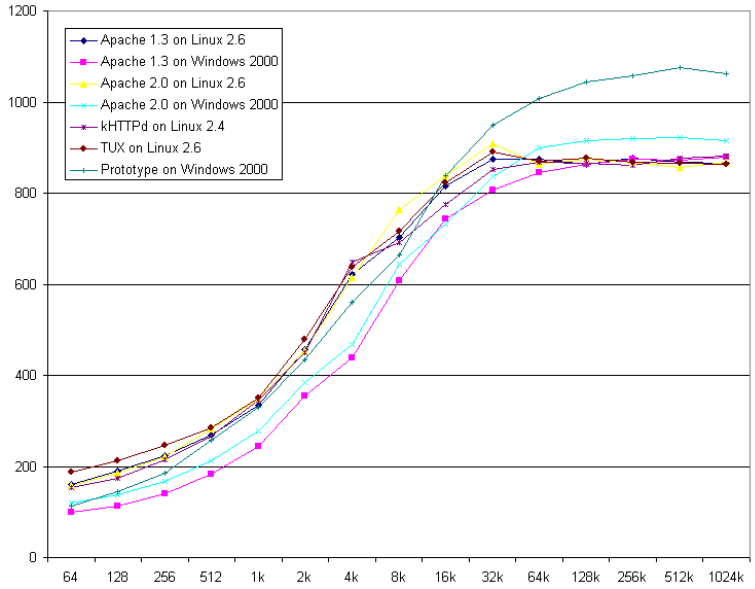
measure.

5.2 Results

This section shows diagrams with the results from the measurements, the comments are in the next section. Measurement on 10Mb/s Ethernet:

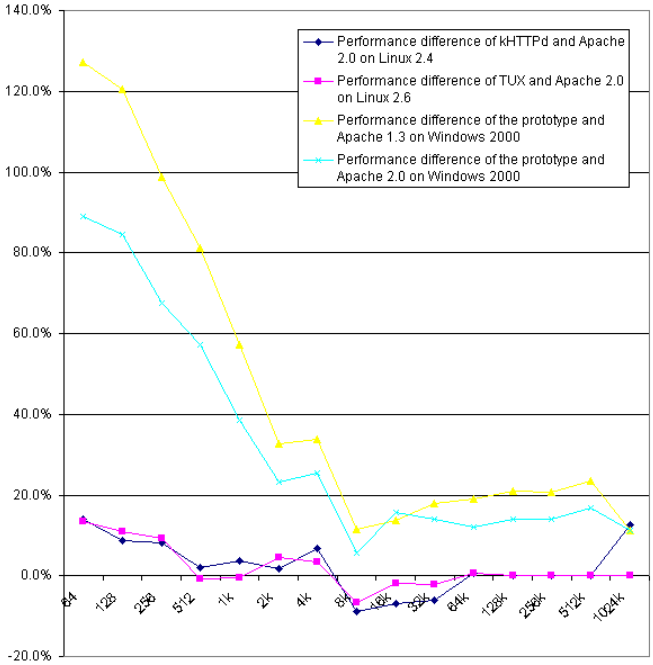


Number of requests per second

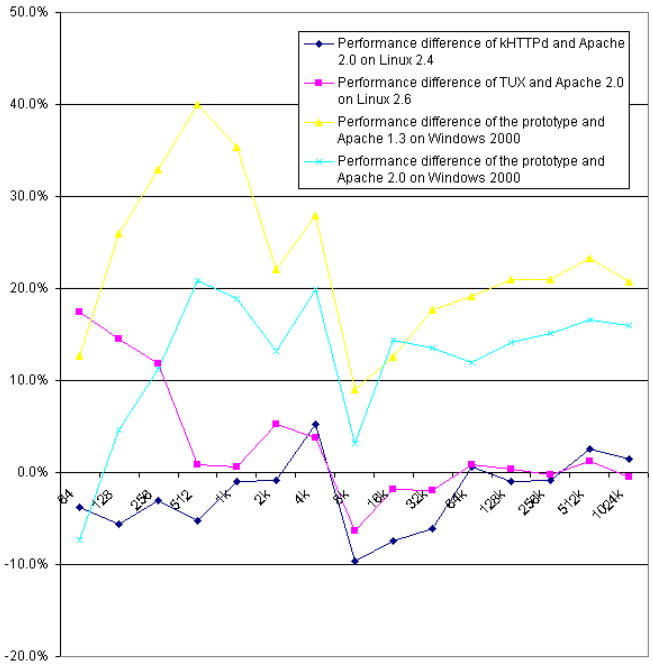


Throughput in kB per second

Measurement on 10Mb/s Ethernet, comparison:

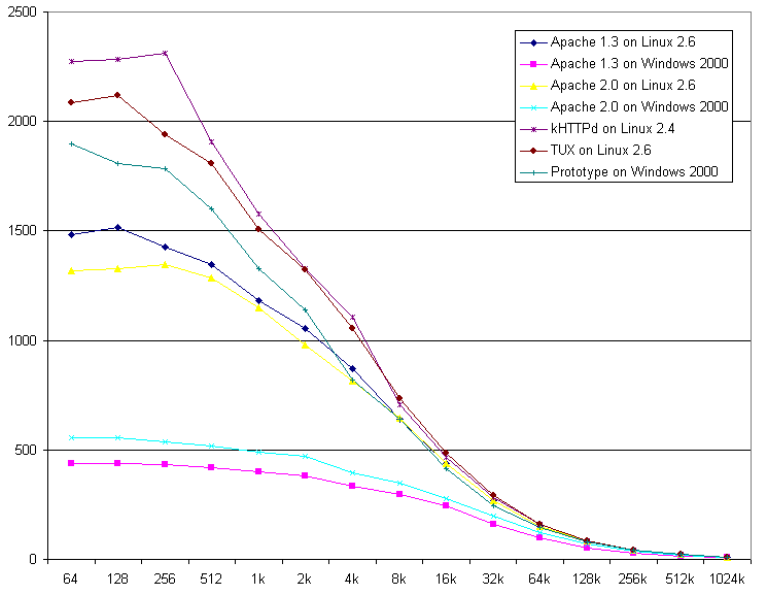


Number of requests per second

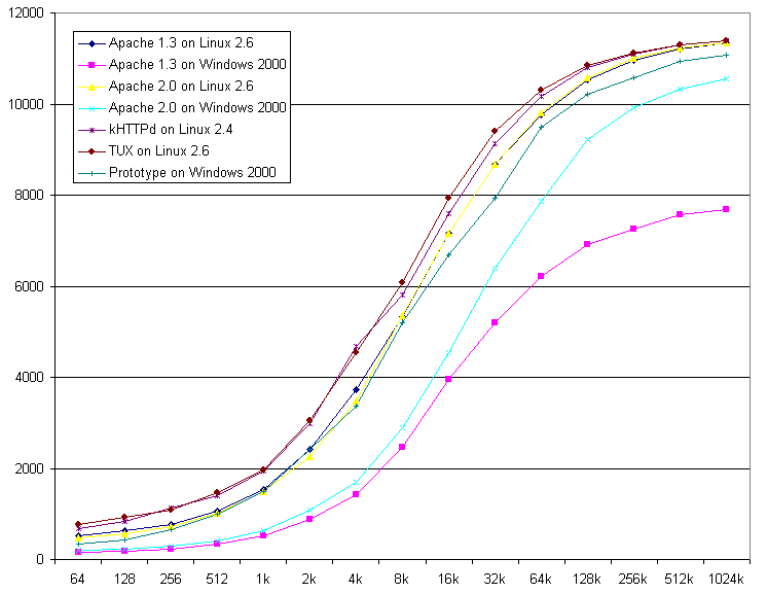


Throughput in kB per second

Measurement on 100Mb/s Ethernet:

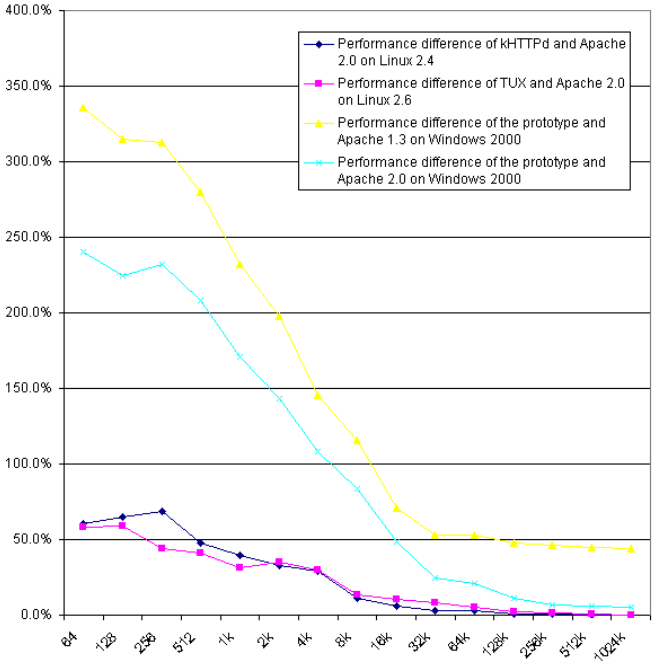


Number of requests per second

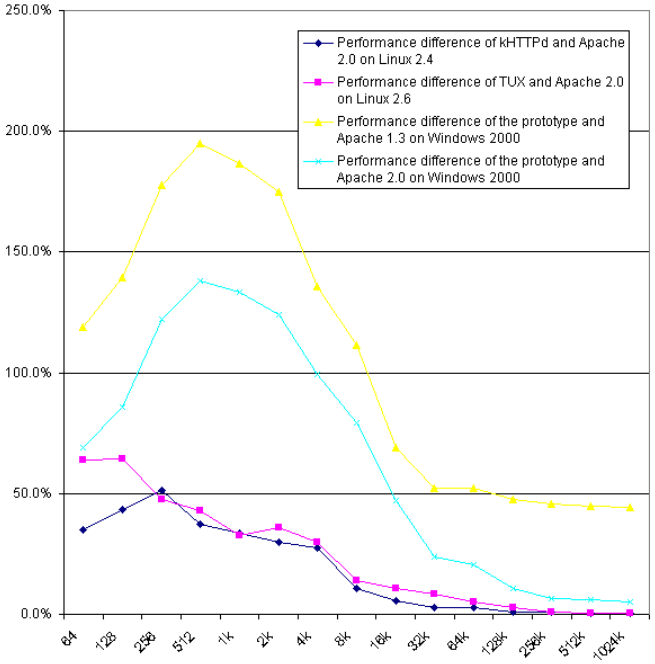


Throughput in kB per second

Measurement on 100Mb/s Ethernet, comparison:

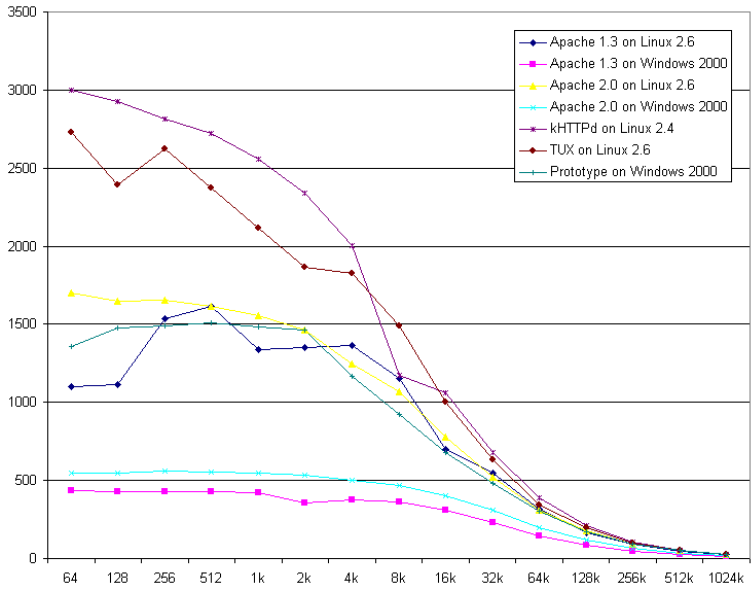


Number of requests per second

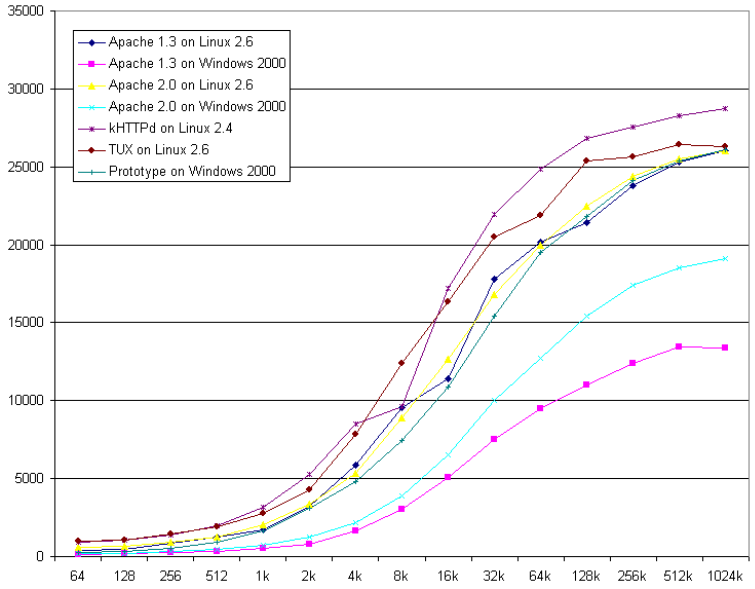


Throughput in kB per second

Measurement on 1000Mb/s Ethernet:

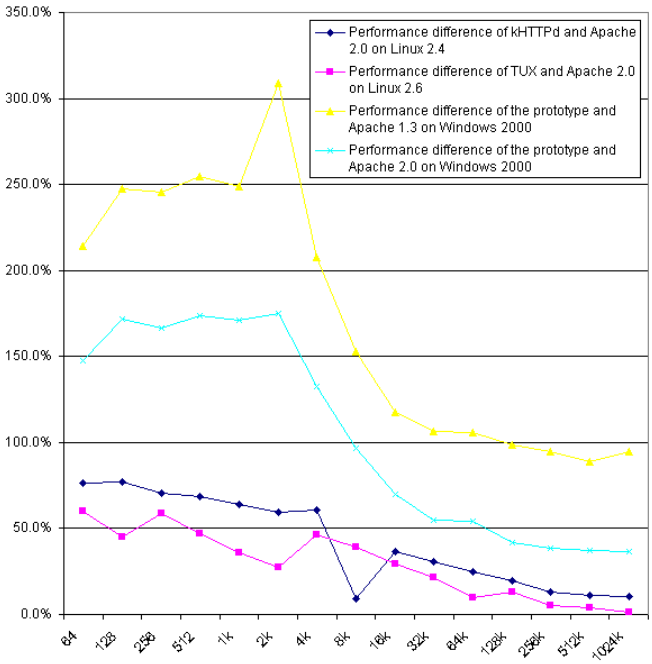


Number of requests per second

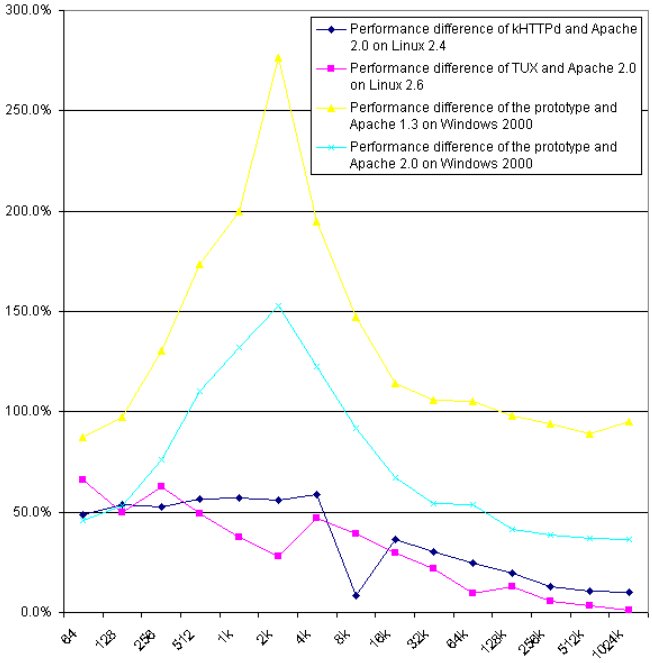


Throughput in kB per second

Measurement on 1000Mb/s Ethernet, comparison:

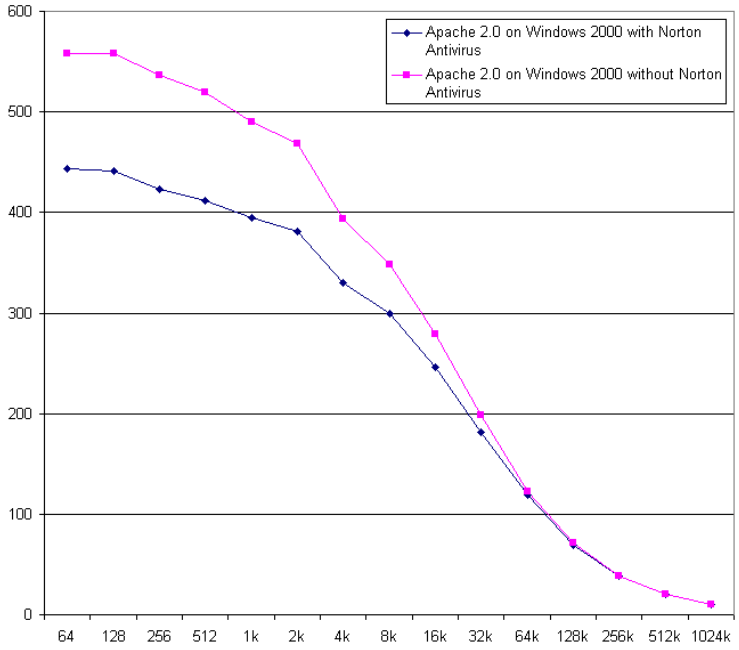


Number of requests per second

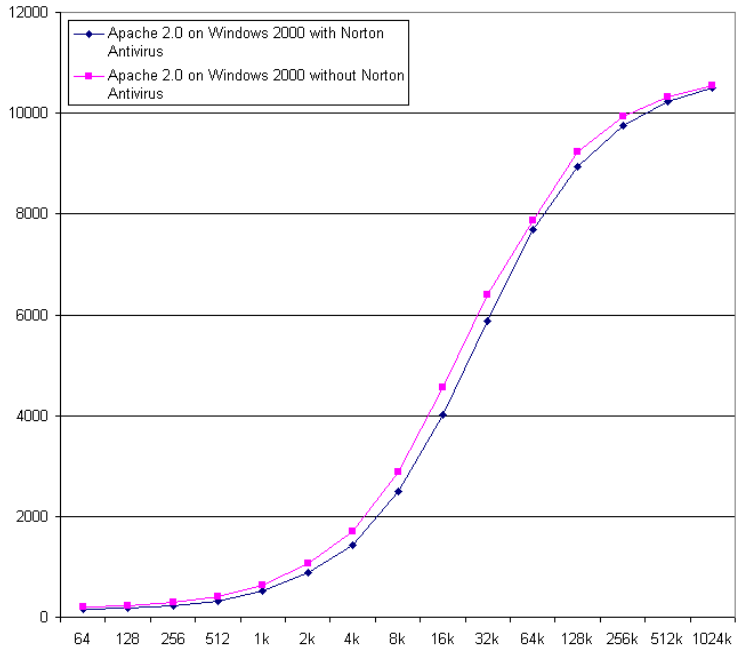


Throughput in kB per second

The effect of an anti-virus filter:

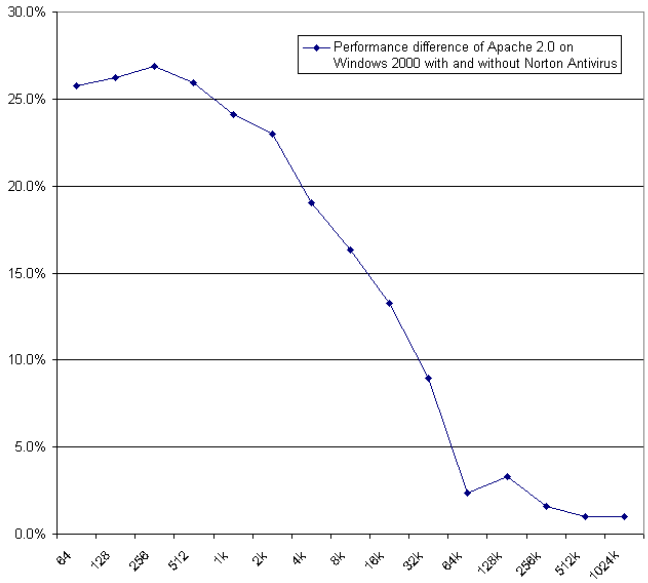


Number of requests per second

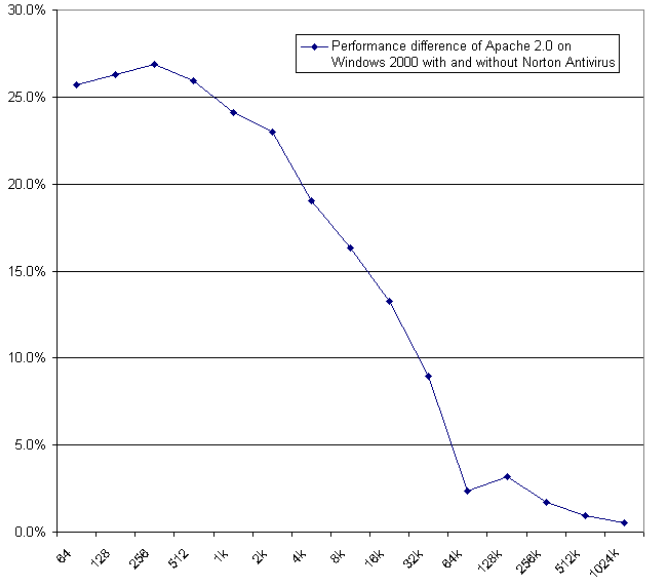


Throughput in kB per second

The effect of an anti-virus filter, comparison:



Number of requests per second



Throughput in kB per second

6 Conclusion

This section describes the conclusions drawn from the project and suggests future work.

6.1 Difficulties

The main difficulty with the project was to implement kernel mode code for Windows. Developing device drivers is more challenging than writing user mode code both on Windows and on Linux, among other reasons is there less support from the debugging tools and a small error in a driver can make the whole operating system unstable or crash. Such errors are often difficult to debug. Another difficulty is that on Windows is the kernel mode interface to network communication in large part undocumented. The TDI interface in itself is documented but this documentation is directed to implementers of new network protocols, there is almost no documentation for developers of TDI clients and most notably is the existing protocols like TCP/IP undocumented. Because of this a major part of the project was developing the kernel mode sockets library.

6.2 Performance

The measurements showed that kernel mode web servers has a substantial performance gain over user mode web servers both on Linux and on Windows.

The factor that is improved most in kernel mode is the latency witch means that a larger number of requests per second can be processed, this is most visible for small files where the kernel mode web servers on Linux is 50% to 75% faster than user mode web servers while the prototype on Windows was as much as 200% to 300% faster than user mode web servers on the same platform, for bigger files decreases the difference to only 0% to 25%. The reason for the big improvement in latency is probably that a kernel mode web server avoids unnecessary process scheduling.

The throughput is also improved by the kernel mode web servers but the difference decreases as bigger the files is. For small and medium sized files has the kernel mode web servers on Linux 25% to 50% higher throughput and the prototype on Windows 100% to 200% higher throughput compared to user mode web servers on the same platform, for bigger files decreases the difference to 0% to 25%. The main reason for the higher throughput from the kernel mode web servers is probably that they avoid unnecessary copying of data in addition to the fact that lower latency also helps getting

higher throughput.

When comparing absolute performance between the two platforms one makes the observation that user mode web servers is faster on Linux than on Windows, all kernel mode web servers is faster than the user mode web servers and the kernel mode web servers on Linux is slightly faster than the prototype kernel mode web server developed for Windows.

I also tried to do measurements on 1000Mb/s Ethernet but the Linux driver for the network card was a beta version and produced unpredictable results, however the diagrams is included and the measurements on Windows is valid and show the same pattern that the measurements on 10Mb/s Ethernet and 100Mb/s Ethernet.

Also for informational purposes I included the results that showed that an anti-virus filter may decrease the performance as much as 25% for small files while the effect is smaller for bigger files. Even though the anti-virus filter scans a file for viruses on read it mainly affects the latency, the reason for this is probably that the scan is scheduled to a system thread while the scan itself is a fast operation.

The conclusion of the measurements is that kernel mode web servers should be worth considering from a performance viewpoint, that the performance gain from a kernel mode web server is bigger on Windows than on Linux when compared to user mode web servers on the same platform and that the prototype performs well compared to the kernel mode web servers on Linux but that there is room for further optimizations.

6.3 Stability

By tradition as much software as possible is placed in user mode while kernel mode is only chosen when direct access to hardware is needed or performance motivates it, for example by device drivers. The most important reason for this is stability, programs that execute in user mode is protected from each other, every process has its own virtual memory context, this means that an error in one process, like using an invalid pointer to write to the wrong memory location, can not affect other processes instead will the process in question be terminated. Software that execute in kernel mode on the other hand has full access to the operating systems internal data structures and the hardware, an error in this code usually brings down the whole system or creates data corruption. Therefore it has been debated if a web server really belongs in kernel mode even if it means some performance gain, however there is a trend to move more and more software to kernel mode, for example does both Windows and Linux runs its network file system servers in kernel mode. A web server can be compared to a simple file server, relative other

file servers is it small and not very complex so the arguments to not run a web server in kernel mode is weak considering today's trends.

6.4 Usability

Another thing to consider when deciding if a kernel mode web server is worth the performance gain is usability. A web server needs a certain amount of administration, for example it need to be configured, also it will produce log files that should be studied and archived. Adding a kernel mode web server that handles some requests and hands over others to the user mode web server increases the administrative work since there are two web servers to take care of. However it would be possible to integrate the two web servers in one product, in some specific cases it is even possible that it is enough with a kernel mode web server.

6.5 Future work

The future work can be divided in creating a product based on this project or doing more research.

If creating a real product the main thing to do is to fully implement the kernel mode web server, the prototype made in this project only handles the HTTP GET request, a product would need to handle the other HTTP requests as well as the different headers that helps web browsers and proxy servers to be more efficient. A product would also need to support log files and have some interface for configuration. There could also be a need to improve the kernel mode sockets implementation to be more optimized and better support the need of a web server.

If instead doing more research an obvious thing to do is to further optimize the interface to the file system and the TCP/IP-stack, for example using the fast-I/O functions when reading from the file system and making the interface to the TCP/IP-stack more event driven could improve performance.

Another thing that would be interesting to investigate is caching of file data so that it could be served directly from the TDI event callback that notifies about the arrival of a request from the client. The TDI event callbacks runs at DPC level, something that can be compared to bottom-half interrupt handlers in Linux, on DPC level is page faults not allowed and it is not possible to make lookups in the file system, to be able to answer a request here one would need to cache files in non-paged memory. An idea is to not implement a full private cache for the kernel mode web server but instead use the file system cache even in this situation, an MDL representing data in the file system cache that has been requested using the `IRP_MN_MDL` minor

function can be used at DPC level if the memory has been made non-paged with a call to `MmProbeAndLockPages()`. Therefore does the private cache only have to maintain a table of file names and MDLs for the file data that has been read and locked before hand. If the requested file is in the table can the file data be sent directly from the receive event handler, if the file is not yet cached can the receive event handler schedule a work item to cache the file using an LRU algorithm and send it, the file is then available in the cache for further requests. It should be noted that the call `TDI_SEND` is allowed at DPC level.

6.6 Commercialisations

A natural commercialisation of this project is to create a product containing a kernel mode web server that can be used together with any user mode web server to increase its performance. However selling software is always difficult and there is competition from free or open source products. It is also possible that a manufacturer of a user mode web server uses this method to implement an integrated product that contains both a kernel mode and a user mode web server that has a common interface for administration.

A different way to commercialise this project is instead to focus on the implementation of the kernel mode sockets library, this could be further developed to a general product that can be used to implement high performance network servers or distributed file systems, an idea is to publish the kernel mode web server for free as an example of a product using the library, if it receives high results in benchmarks it could help marketing the library.

6.7 Comments

Finally I would like to say that this project was very interesting to work with, I have learned a lot about the TCP/IP protocol and its implementation on both Windows and Linux. It was also interesting to do the benchmarks of different web servers and see how the hardware and the operating system support affects there performance. I think that the project has succeeded in implementing and evaluating a High performance kernel mode web server for Windows and that it has created knowledge about network communication from kernel mode in Windows that has general use in further work.

References

[1] Department for Applied Physics and Electronics at Umeå University. http://www.tfe.umu.se/.

[2] Open Systems Resources Inc (OSR). http://www.osr.com/.

[3] Sysinternals. http://www.sysinternals.com/.

[4] GTC Security AB. http://www.gtcsecurity.com/.

[5] Mindcraft. Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 Upgraded to the Linux 2.2.2 Kernel. http://www.mindcraft.com/whitepapers/nts4rhlinux.html.

[6] Arjan van de Ven. kHTTPd: Linux HTTP Accelerator. http://www.fenrus.demon.nl/.

[7] Ingo Molnar. TUX: Threaded linUX http layer. http://people.redhat.com/mingo/TUX-patches/.

[8] Red Hat. Red Hat Content Accelerator Manual. http://www.redhat.com/docs/manuals/tux/.

[9] Philippe Joubert, Robert B. Kingy, Rich Neves, Mark Russinovichz, John M. Tracey. High-Performance Memory-BasedWeb Servers: Kernel and User-Space Performance. Proceedings of the 2001 USENIX Anual Technical Conference, Boston, MA, June 28, 2001. http://sysinternals.com/files/webserver.pdf.

[10] Linuxant. DriverLoader for Wireless LAN devices. http://www.linuxant.com/driverloader/.

[11] NdisWrapper. http://sourceforge.net/projects/ndiswrapper/.

[12] The MSDN Library. http://msdn.microsoft.com/library/default.asp.

[13] The Microsoft Windows DDK Online. http://www.osronline.com/ddkx/ddk.htm.

[14] Jeffrey Mogul, K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. http://www.cs.brown.edu/courses/cs161/papers/livelock.pdf.

[15] David Mosberger. httperf: A Tool for Measuring Web Server Performance. http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.

[16] Mindcraft. WebStone: The Benchmark for Web Servers. http://www.mindcraft.com/webstone/.

[17] Standard Performance Evaluation Corporation (SPEC). SpecWeb99: The Web server benchmark from the Standard Performance Evaluation Corporation. http://www.spec.org/osg/web99/.

[18] Gentoo Linux. http://www.gentoo.org/.

A Abbreviations

APC Asynchronous Procedure Call.

API Application Programming Interface.

BSD Berkeley Software Distribution.

CGI Common Gateway Interface.

DDK Driver Development Kit.

DLL Dynamic Link Library.

DMA Direct Memory Access.

DPC Deferred Procedure Call.

EA Extended Attribute.

FSD File System Driver.

FTP File Transfer Protocol.

GUID Globally Unique IDentifier.

HAL Hardware Abstraction Layer.

HTTP HyperText Transfer Protocol.

IFS Installable File System.

IIS Internet Information Server.

IOCTL I/O ConTroL code.

IP Internet Protocol.

IrDA Infrared Data Association.

IRP I/O Request Packet.

IRQL Interrupt ReQuest Level.

MDL Memory Descriptor List.

MIME Multipurpose Internet Mail Extensions.

NDIS Network Driver Interface Specification.

NIC Network Interface Card.

SAN System Area Network.

SDK Software Development Kit.

SEH Structured Exception Handling.

SMB Server Message Block.

TCP Transmission Control Protocol.

TDI Transport Driver Interface.

URL Uniform Resource Locator.

WSA Windows Sockets API.

WSH Windows Sockets Helper.

WSP Windows sockets Service Provider.

WSU Windows Sockets Upcall.