PORTABLE TCP/IP SERVER DESIGN

by

Robert Mark Jolliffe

submitted in part fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in the subject

Computer Science

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISORS: Dr W Smuts and Prof J A van der Poll

November 2002

Abstract

There are a number of known architectural patterns for TCP/IP server design. I present a survey of design choices based on some of the most common of these patterns. I have demonstrated, with working code samples, that most of these architectural patterns are readily portable between UNIX and Windows NT platforms without necessarily incurring significant performance penalties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *First law of portability: Every useful application outlives the platform on which it was developed and deployed.* Steven Walli [59]

This dissertation is an investigation of the portability of TCP/IP server architectures between UNIX-like and Microsoft Windows NT derived[1] systems.

1.1 Rationale

Writing communication software is difficult. Whereas this may seem rather an obvious point to make, it has some bearing on the investigations concluded in this research. If writing such software were trivial there would not be a need to consider issues of portability at all. If, by porting the software between platforms, we can avoid the cost of an expensive rewrite and reclaim some of the initial investment in development time, there is much to gained. Indeed, it is this promise of reusing large repositories of existing code which has motivated the development of tools such as Cygwin [31], Uwin [18] and Interix [59](discussed in Chapter 2).

Creating new code offers more opportunities for building system portability into the design. Frequently these opportunities are not siezed. There are probably four major factors which might drive a development effort towards targeting a single platform API, rather than making portability part of the design rationale:

[1] Windows NT derived systems include NT 4.0, Windows 2000 and Windows XP.

- Familiarity with a particular system API. Given that there are so many complexities associated with communication software design, this familiarity is valuable and not easily attained.
- The need to optimise performance. Developers may well shy away from greater abstraction for fear of the application suffering an unacceptable performance hit. I show in Chapter 5 that this fear is not necessarily well founded. Indeed, design choices at the architectural level will likely have a significantly greater impact on performance than a judicious choice of abstraction layer.
- Dependence on unique or particular operating system functionality. Frequently this dependence may not be as strong as it might appear. There is a surprizing overlap of functionality between diverse operating systems. The use of Wrapper Facades [49] provides a powerful example of how to avoid such dependencies.
- The belief that the software will never need to be ported. Whereas much existing code seems to reflect this belief, this can be a costly assumption.

My experience of writing the simple illustrative servers used in this research supports the claim that writing communication software is difficult. The development process is long and error prone. An understanding of the fundamental design patterns of server architectures is certainly useful, but the devil is in the detail. There is a considerable effort in time spent on

- reading and verifying API documentation (which is, more often than not, out of synch with the software)
- tracking down infrequent and intermitent run time errors - running a server which fields 100 requests does not necessarily encounter the occasional problems and memory leaks seen when fielding 1 000 000 requests
- making the resulting implementation as efficient as possible.

1.2 Problem statement

TCP/IP servers make use of low-level operating system calls and services. The significant differences in both the implementation and interface of such services between Microsoft

Windows and UNIX-like systems create difficulties when designing software which is targeted at both platform sets.

1.3 Hypothesis

My hypothesis is the following:

> The use of thin layers to abstract and mask the difference between different operating system interfaces allows one to target both Windows and Unix platforms without necessarily incurring significant performance penalties.

1.4 Scope

The scope of this research is limited to a survey of TCP/IP server architectures using the Berkeley Sockets API. Other interfaces and protocols (such as XTI) are not considered. Such restriction of scope is justified on the grounds that TCP/IP and Berkeley sockets represent a de facto standard for network server design [26][54]. Nevertheless, the architectural issues raised (threads, processes, I/O multiplexing etc) are essentially protocol and interface independent.

The code samples are meant to be illustrative. Whereas every attempt has been made to implement correct code[2], there are instances where robustness may have been sacrificed in the interest of readability.

C++ is used in most of the implementations discussed. Most of the texts dealing with system level programming [55][56][43] use examples coded in ANSI C. A discussion of the particular problems and merits of using C++ is presented in Section 6.2.2.

The list of tools used to provide portable implementations is far from exhaustive. I have instead attempted to select a small range of tools and libraries which are representative of the broader strategies outlined in Chapter 2. Related alternative packages are mentioned in context.

[2]Each of the server programs presented in this dissertation has been run for many hours and handled tens of thousands of connections without aborting or leaking memory.

The discussion on server architectures is mostly restricted to those using synchronous blocking and non-blocking I/O. Specifically, I have not examined the *asynchronous I/O* mechanisms provided by the POSIX Realtime extensions and the approximate Windows equivalent, *Overlapped I/O*, in any detail. Asynchronous I/O was avoided initially because of the very weak support provided by early (v2.2.x) linux kernels. Linux support for the POSIX Realtime extensions has improved considerably since. A proper discussion and comparison of POSIX Asynchronous I/O and the Windows Overlapped I/O model is left for future work.

There are a number of issues relating to portability which have not been considered. Specifically different filesystem and security models are not addressed in this dissertation.

1.5 Organization

This section describes the organization of the chapters within this document.

I describe a range of approaches and tools for targeting these platforms in Chapter 2.

The BSD sockets API has become the *de facto* standard for implementing TCP/IP servers on UNIX and its many derivatives. Winsock is the equivalent API on Windows systems. Chapter 3 provides a brief overview of the sockets interface of BSD UNIX, followed by a discussion of the areas of compatibility and variation with the Winsock API.

Event driven and Concurrent architectures for handling multiple concurrent connections are discussed in Chapter 4. Discussion around the examples in these chapters illustrate the problems, solutions and insights encountered in addressing issues of portability.

Chapter 5 presents a quantitative comparison of the porting tools and techniques presented in earlier chapters. To facilitate the reproduction of a large number of experiments I developed a distributed load generation and monitoring system using CORBA. This system is described fully in Appendix A.

I have included an appendix on electronic media, with all of the source code referenced in this dissertation together with a number of benchmarks referred to in the text.

1.6 Glossary of terms

The following is a short list of terms and acronyms

TCP/IP A family of network protocols forming the backbone of the internet. First widely available release was in 4.2BSD (1983), together with the sockets API. Consists of application layer (telnet, FTP, SMTP, etc), transport layer (tcp [37] and udp [35]) as well as network layer [36] (ipv4 and ipv6) protocols. Figure 3.3 shows the relationship between a TCP socket and the protocol stack.

POSIX Acronym for Portable Operating System Interface. A set of standards relating to operating systems maintained by the PASC working group of the IEEE. The POSIX standards have since been adopted by IEC and the ISO. The standard mostly referred to is IEEE Std 1003.1 (or *Posix.1*) which includes 1003.1-1990 (the base API)[3], 1003.1b-1993 (real-time extensions), 1003.1c-1995 (Pthreads) and 1003.1i-1995 (technical corrections to 1003.1b).

OS I have used OS as an abbreviation for Operating System.

Patterns A trend in software engineering to capture and reuse proven good design models. Based originally on the work of the architect, Alexander[1], but brought into the software engineering mainstream by Gamma, Helm, Johnson and Vlissides (affectionately termed the Gang of Four) [8].

Server I have used the term to describe a process running on a host which accepts connections from one or more Client processes, running on one or more separate hosts, for the purpose of providing a service.

API Application Programming Interface. An API describes the exported visible function calls and structures presented by a library or system. The API is used by a programmer to access the features and services of the system or library.

IPC Inter-process communication. A term encompassing the range of facilities through which processes can communicate on a multi-tasking operating system. Examples include pipes, shared memory, message queues and sockets.

DLL Dynamically linked library. Popular Windows terminology for a shared library. Shared libraries on Windows systems have a `.dll` extension.

[3]This is the standard which the Windows NT Posix subsystem implements.

Portable I use this term to describe a body of source code which can be compiled and executed on a heterogenous mix of target operating system platforms.

RFC Request For Comment - publically available documents communicating the work of the working groups of the Internet Engineering Task Force (IETF). RFC's are accorded different statuses eg: Standard, Proposed Standard Informational, Historical etc.

Chapter 2

Porting strategies

In this chapter I describe some common strategies adopted in porting existing code as well as writing portable code.

2.1 Portability

I use the term portability specifically in relation to Windows NT and Unix. Though there are certainly many portability issues between the different flavours of Unix[20] and the different versions of Microsoft Windows, I do not specifically address them here. The assumption is that most modern Unix systems supply an interface based on the IEEE/ISO standard for operating systems, commonly known as POSIX[1]. I have, as far as possible, restricted my Unix code examples to use only POSIX features. The scope is thus reduced to portability between Windows NT and POSIX compliant systems.

In short, software is expensive to produce. Complex software such as communication software (in which category I place TCP/IP servers) is even more expensive to produce, requiring expensive programming talent and extensive and rigorous testing. Porting such software represents an attempt to recover some of this development cost by reusing existing code to target new platforms. The software that was expensive to produce should not

[1]The POSIX set of standards are not without their problems. In particular, the standards define an interface and say nothing about implementation details. The standards are also evolving. The POSIX Real-time Extensions, for example, are not fully implemented on many systems.

necessarily be expensive to reproduce. To what extent this is true depends largely on the early design rationale. Software that is written with portability in mind from the start is going to be easier to work with, than software that was targeted at a particular platform.

Walli [59] outlines essentially four approaches to moving existing applications to Windows NT:

1. a complete rewrite of the entire application
2. using a UNIX emulation layer
3. using a cross-platform library
4. using a POSIX environment subsystem

We can add to this list a fifth approach, which is to target the code at a virtual machine. The Java experience bears testimony to considerable success with this approach. The recent arrival of the Microsoft .NET virtual machine is likely to make this an increasingly popular strategy.

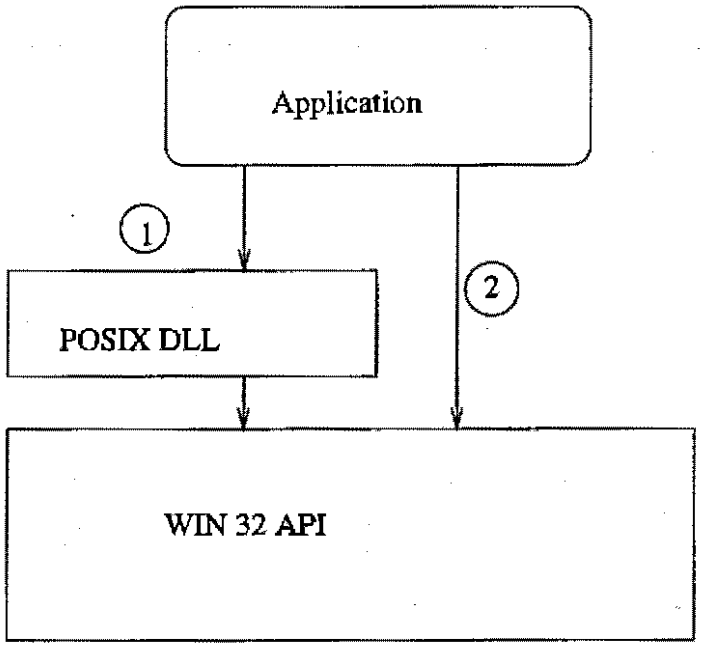
2.2 Emulation layers

Perhaps the easiest way of achieving portability is to use a software layer which provides the interface of one system on another. There is nothing very new with this approach. Michael Franz [7] describes just such an approach in porting the Oberon system to the Apple Macintosh in 1993.

Franz classified four major areas of difficulty in his work on Oberon:

1. Incompatible paradigms
2. Contrasting abstractions
3. Implementation restrictions
4. Performance bottlenecks

I have looked at two freely available examples of emulating POSIX on Win32. In both cases they involve a DLL which provides an interface between POSIX system calls and the underlying Win32 subsystem. The one example is the open source Cygwin [31], created by Geoffrey Noer of Cygnus Solutions[2], and Uwin [18][19] from David Korn at AT&T Research Labs. In the rest of this discussion I refer to Cygwin. It is interesting to note from examining the Cygwin source code, that the hurdles identified by Franz are still very much in evidence.



1. Application talks to Win32 api via unix-like POSIX emulation layer
2. Application is still free to make native system calls

Figure 2.1: Cygwin and Uwin use a Posix porting layer on Windows

Figure 2.1 shows how the Cygwin DLL provides the POSIX emulation layer. One of the benefits of this approach is that once the porting layer is in place, it can be used to port the Unix development tools themselves (such as the gcc compiler, linker, make utility etc). Most of the GNU utilities required to configure and build a software package on a Unix system have been ported to Windows NT in this way. This allows one to use, not just a common source

[2]Cygnus was bought by RedHat in 1999.

base, but also common configuration and Makefiles. The Cygwin development environment has been used to provide a (relatively) simple porting option for many of the "classic" TCP/IP servers such as the apache web server and the telnetd daemon[3]. I show in Chapter 4 how such an emulation layer can be used to provide the fork() system call on Windows NT. These characteristics make the approach particularly suitable for porting existing Unix code to Windows NT.

Although I have only considered emulating POSIX system calls on Windows NT, there are also some interesting examples of going the other way i.e. "doing" Windows on Unix. The WINE package[4] is one such example of a rapidly evolving, open source solution to developing and running Win32 binaries in a Unix environment. A more recent example is the Rotor project. The Rotor project is a port to FreeBSD of Microsoft's .NET environment. A significant component of Rotor is the Platform Adaptor Layer (PAL). The FreeBSD PAL serves to map from Win32 API calls into equivalent functionality on FreeBSD, where it exists, or implements that functionality where it doesn't. In this way the PAL performs the same function as the Cygwin and Uwin libraries - emulating one system on top of another.

2.3 Abstracting the operating system through a library

The previous section describes a process whereby one coerces one system to behave like, or emulate another. Another approach is to use a higher level of abstraction, which presents an interface which is independent of the underlying system dependent implementation details[5]. As long as the program interacts only with this interface, and does not make direct calls to the system API, the challenge is reduced to porting the abstraction layer to different platforms, rather than porting the individual applications. Such an abstraction layer is typically implemented in the form of a library. There are many libraries which provide useful abstractions for communication software. I have made use of two very different libraries to

[3]A comprehensive list of references to successful Cygwin ports can be found at http://sources.redhat.com/cygwin.

[4]available from http://www.winehq.org/.

[5]It could be argued that the POSIX API is meant to provide exactly such an abstraction. In fact the difference is one of degree. The emphasis of the libraries discussed here is on value added abstraction, rather than simply a wrapping of system calls.

illustrate this approach. These are the Tcl library and the ACE library which are described in Sections 2.3.1 and 2.3.2 below.

2.3.1 The Tcl library

Tcl (or Tool Command Language)[33] is an interpreted scripting language created by Dr John Ousterhoudt in 1987. It is best known in combination with a GUI scripting extension known as Tk. Tcl and Tk have found particularly widespread use in interactive communication software which requires a graphical user interface as well as communication capabilities. Examples of such software include Groupkit, a Tcl/TK based toolkit for developing interactive group-ware, as well as many of the Mbone [23] suite of tools.

Tk, like most other GUI toolkits, features strong support for the event driven model of programming. An application might typically initialize and display a set of widgets in a window and then enter an event processing loop. The main application code is written as a series of event handling procedures which are arranged to be invoked in response to user interaction with the widgets. The software component within Tk which implements the event demultiplexing and dispatching is known as the Notifier.

Three features of Tcl which make an unlikely sounding tool an interesting component in the design of TCP/IP servers are:

1. Tcl and Tk have been ported to all major Unix variants as well as Win32 and Macintosh systems.

2. The Tcl language is implemented in terms of a well defined and easily extensible C library. C programs can link against this library to gain access to all of the features available to the Tcl interpreter, including abstractions for creating network channels and the Tcl Notifier.

3. The Notifier is not restricted to window events. Event handlers can also be registered to be called back in response to I/O events (such as the file descriptor corresponding to a connected TCP/IP socket becoming readable or writable) and timer events as well as user defined event sources. The usefulness of this feature is illustrated in Section 4.2, which discusses event driven servers.

Whereas Tcl has proved useful primarily through its provision of an event driven framework, recent versions of Tcl (starting with 8.1) are also thread-safe and offer a number of useful functions to create and manipulate mutexes, thread specific storage, condition variables as well as per-thread event queues. I do not give any examples of using these facilities, but note that they exist and potentially increase the range of problem domains to which we can apply the Tcl library.

2.3.2 The ACE library

ACE is the product of ongoing work by Doug Schmidt [47] and his research team at Washington University, St. Louis. The following description applies:

> The Adaptive Communication Environment (ACE) is an object oriented framework and toolkit that implements core concurrency and distribution patterns for communication software [57]

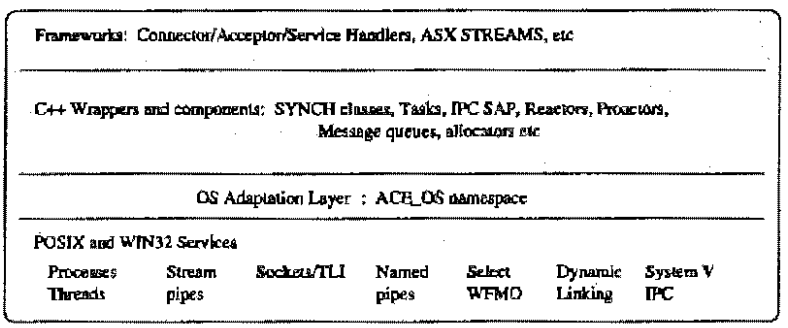


Figure 2.2: ACE services

As can be seen from Figure 2.2, ACE has a layered architecture. The lowest layer is the OS Abstraction layer. This layer hides the detail of different C API's within a comprehensive set of C++ wrappers. Doug Schmidt has written extensively on the subject of using patterns for software design, so it is not surprising to see that much of the ACE library consists of the implementation of various design patterns and frameworks common to the domain of communication software. The C++ wrappers themselves are described in terms of implementations of a Wrapper Facade pattern [49]. Schmidt argues that the use of C++ wrappers to encapsulate low-level functions and data structures can make network code more

concise, robust, portable and maintainable. It is a persuasive argument which is borne out by the many ACE code samples, illustrating solutions to common problems.

One of the drawbacks of using C++ is the problem of different compiler implementations. Whereas the C language has been stable and standardized for many years, the ISO/ANSI standard for C++ [13] is still very young (The first edition was only finalized in 1998). Compiler vendors are still working towards full conformance. Therefore the problem of creating portable C++ class libraries is effectively doubled - it must be portable to different architectures and also to different compilers.[6] Nevertheless, portable C++ code can be written by restricting the code to a subset of standard C++ features.

Other examples of C++ class libraries which encapsulate low-level C API calls on various systems are Rogue Wave's Net.h++ and Threads.h++ libraries, the ObjectSpace `System<Toolkit>`, Microsoft's MFC library and, more recently, Microsoft's .NET framework.

It is beyond the scope of this dissertation to describe all of the features of ACE. The code examples in Chapter 4 should give the reader some idea of the flavour of the library.

2.4 Windows NT environment subsystems

Windows NT (and Windows 2000) uses a microkernel architecture organized as a layered system of modules [6][52]. Figure 2.3 shows a simplified block diagram of this architecture:

The Hardware Abstraction Layer (HAL), kernel and executive run in protected mode together with some of the window manager code which was brought down from the Win32 subsystem to enhance performance in Window NT4.0. The HAL exports a virtual machine interface which is used by the kernel, the executive and device drivers. This is also bypassed for performance reasons by the graphics and I/O drivers.

A variety of subsystems run in user mode on top of this model. The most interesting from our perspective are the environment subsystems. The Win32 subsystem provides the main operating environment and exports the Win32 API to user processes. The environment subsystem approach does allow Windows NT to support other environments. Besides Win32, the original intention was to provide robust support for MSDOS, 16 bit Windows,

[6] The situation regarding standards conformant C++ compilers has improved considerably in recent years.

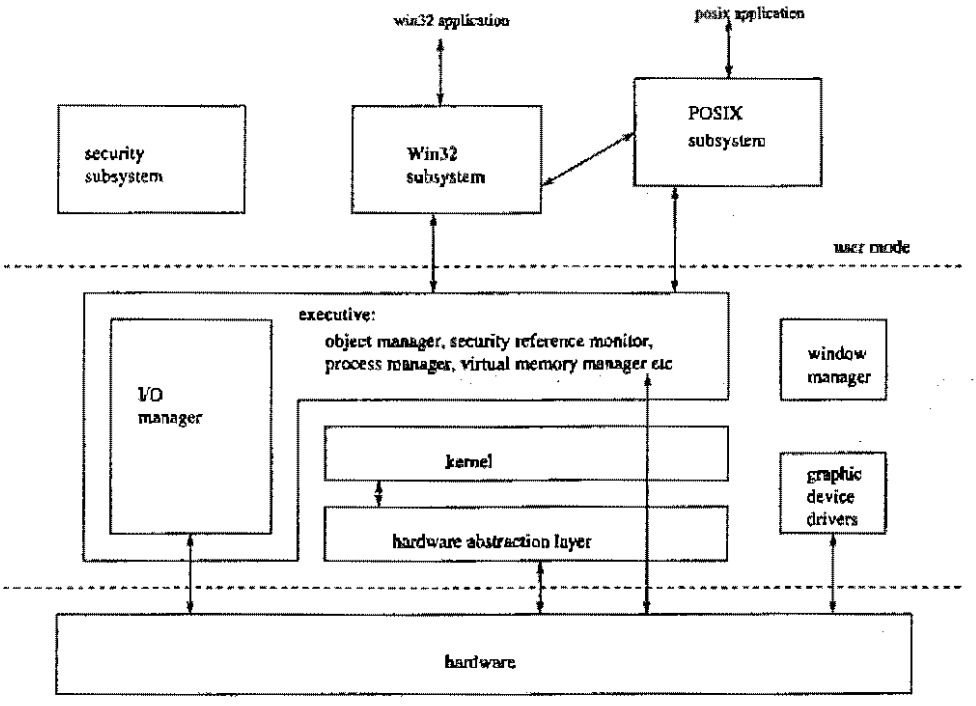


Figure 2.3: Windows 2000 simplified architecture

OS/2 and POSIX applications. The popularity of Win32 has meant in effect that these other subsystems have not been the subject of much ongoing development. The POSIX subsystem was never envisaged to be widely used at all and provides very minimal functionality. Nevertheless, the environment subsystem concept allows for the possible implementation of a more complete, usable POSIX subsystem.

Interix is such an environment subsystem. The original environment was called OpenNT and was developed and marketed by a company called Softway Systems[59]. OpenNT has since been bought by Microsoft and is renamed Interix. Interix is now a key component used in the implementation of Microsoft's Services For Unix (SFU) product.

Interix provides a UNIX-like environment much like Cygwin and Uwin, with shells, utilities and developer tools. The runtime performance of services on Interix is potentially better in that the environment is directly layered on top of the NT executive. In fact, my experiments have shown that, except for the implementation of process `fork()`-ing, there appears to be no significant performance gain over either Cygwin or Uwin.

2.5 Java and virtual machines

One way of gaining program portability is to compile programs down to an intermediate form based on an abstract machine definition. John Gough [10] notes that such approaches date back as far as the 1970's. The intermediate code can be either compiled down to the native machine code of the target platform, or executed via an interpreter which emulates the abstract machine.

2.5.1 Java

The Java Virtual Machine is a well known modern example of a portable abstract machine. Patrick Naughton, one of the early pioneers of the "Java Revolution", cites architecture neutrality as being part of the core design rationale [30]. A Java compiler compiles Java source code into byte-codes which are "binary" compatible with the Java Virtual Machine(JVM). The JVM interprets the byte-codes on the host system at run-time. This architecture is illustrated in Figure 2.4.

Even though Java technology performs considerably better than other cross-platform interpreted systems such as BASIC, Tcl and Perl, the interpreted byte-codes are still many times slower than native machine code compiled with a C or C++ compiler. The situation can be improved by using Just-In-Time (JIT) compiler technology which dynamically compiles byte-codes to native machine code immediately prior to method execution. Per Bothner[5] lists two of the limitations of JIT technology as:

1. The compilation has to be done each time the application executes, which has a negative effect on start-up times.

2. The JIT compiler has to run fast, so is unable to make use of aggressive optimization techniques.

Bothner [5] claims that whereas Java is a "decent" language,

> ... it cannot become a mainstream programming language without mainstream implementation techniques, specifically an optimizing, ahead-of-time compiler.

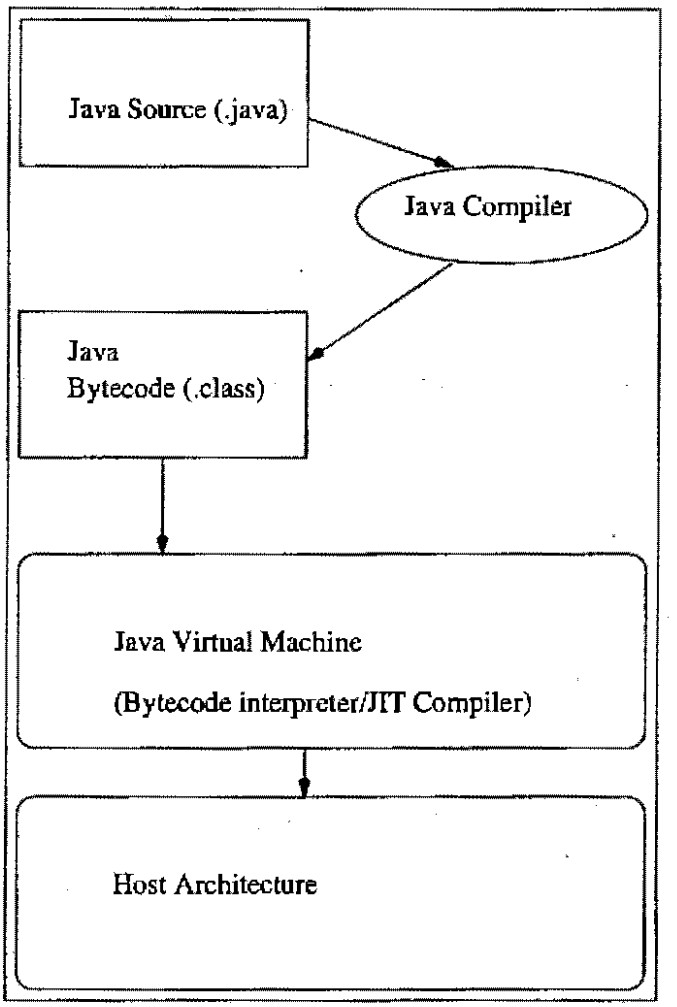


Figure 2.4: Java OS Abstraction using a Virtual Machine

Bothner and his team at Cygnus Solutions successfully produced a Java compiler front end for gcc[7] and a functional run-time library.

Many of the disadvantages of using Java stem from its dependence on the virtual machine. The performance and resource costs may be be too significant for many applications, particularly those requiring high performance, small footprint or both.

Despite these drawbacks, Java is an increasingly popular choice as an environment for building server applications. Java 2 incorporates an implementation of the OMG CORBA specification. There are a number of pure Java web servers which take advantage of the

[7]the Java front-end, gcj, is integrated into gcc versions 2.9X, available from http://gcc.gnu.org/.

successful Java servlet extension[8]. The success of Java in these areas illustrates that raw speed and efficiency is not necessarily the most important factor in server design. Robustness, ease of programming and standardized interfaces are often far more important design criteria.

2.5.2 .NET

Details of the .NET system were released by Microsoft during 2000. The system consists of a number of components, including an object-oriented and garbage collected runtime. The runtime processes an intermediate form known as MSIL (Microsoft Intermediate Language)[28] on an abstract stack machine superficially similar to the JVM[9].

The open source version of the .NET runtime, Rotor, has been successfully ported to FreeBSD and more recently to Linux. The Rotor source code reveals some interesting details:

1. The FreeBSD port is built on top of a platform adaption layer. As noted in Section 2.2 above, this can be used independently of the virtual machine in the much same way as Cygwin and Uwin, to provide a Win32 API on UNIX-like systems.

2. Unlike the JVM, .NET does not support only one front-end language. Currently C, C++, C#, Visual Basic and JScript language front-ends can be used to generate the MSIL.

Unfortunately the Rotor system has only very recently been ported to Linux, so I did not have the opportunity to compare its performance against the other strategies and systems discussed. Such an analysis must be left to future work.

2.6 Summary

In this chapter I have presented four alternatives to code rewriting for creating portable code between UNIX-like and Windows NT based systems.

For each of these approaches I have identified existing and freely available example implementations.

[8] the Tomcat server from the Apache Foundation is probably the best known example.

[9] A notable difference between .NET and the Java VM is that the intermediate code for .NET is always JIT compiled, ie there is no interpreter mode.

The particular problem domain within which I am proposing portable solutions is that of TCP/IP server architectures. Underlying such architectures is a dependence on access to the TCP/IP transport layer provided by the system. Therefore an important area in which all of the tools presented here provide some abstraction or emulation is access to the transport layer. The BSD sockets API has emerged as the *de facto* standard interface for applications to access TCP/IP transport. Windows provides a similar interface through its Winsock layer. The next chapter describes the sockets interface on BSD-derived and Windows systems.

Chapter 3

BSD Sockets

A TCP/IP server is a process which offers a service to a remote process or processes. TCP/IP is a suite of protocols which provide transport and network layer services to facilitate the communication. The BSD Sockets API is the *de facto* standard interface by which processes interact with TCP/IP. Microsoft Windows provides a similar interface known as Winsock.

This chapter provides a brief background to the history and rationale of the BSD Sockets API. This is followed by a brief description of the API, as implemented on UNIX-like systems. In order to better understand the porting problem, Section 3.3 discusses the similarities and variations of the Winsock API.

3.1 Background

When a process is created[1] on a multitasking operating system, the system goes to a great deal of trouble to provide that process a safe, isolated environment in which to execute. Most importantly, a process is allocated a virtual address space which

- is the only memory the process can read or write to, and
- is inaccessible to other user-level process.

Unfortunately, processes which can only compute and move data around in this isolated environment are not very useful. They also need to be able to interact with the world around

[1] using `fork()` on UNIX-derived systems or `CreateProcess()` on Win32.

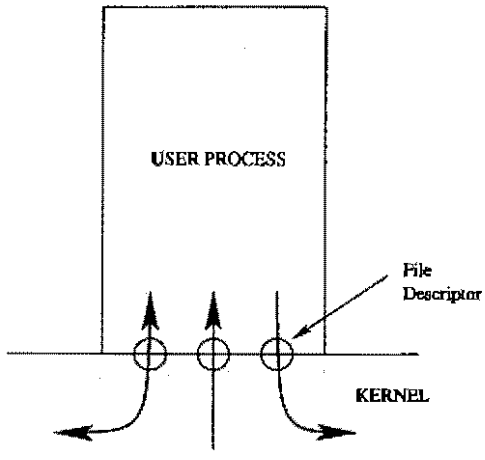


Figure 3.1: Moving data through file descriptors

them. Such interactions include

- reading from and writing to files;
- interacting with devices (and hence indirectly with users);
- communicating with other processes (either local or across a network).

3.1.1 IPC

Sockets provide one means of communicating with other processes (Inter Process Communication or IPC). I have presented this communication paradigm in a general context, because IPC is just an example of the more general problem of moving data in an out of a process address space. In the very first paper published on UNIX by Ritchie and Thomson [42], a novel approach was described which provided a generic solution to the general problem: just treat everything as if it were a file and channel all communication through file descriptors. The "UNIX way" has survived for almost 30 years, and although the concept is certainly becoming strained [17], it remains the most widely used model for IPC.

3.1.2 File descriptors

Figure 3.1 shows how file descriptors provide a controlled interface by which processes can communicate with the outside world. Processes can make use of the `read()` and `write()`

sytem calls to transfer data across the kernel boundary. The prototypes of these two functions are shown below:

```
#include <unistd.h>
int read(int d, void *buf, int nbytes);
int write(int d, void *buf, int nbytes);

                    Return number of bytes read/written or -1 on error
```

In each case the integer parameter, `d`, represents the file descriptor. The variable, `buf`, is a pointer to the area in process memory which contains the data to be written, or the place where data is to be read into, while `nbytes` is the *maximum* number of bytes to be transferred. The kernel maintains a table of open file descriptors in the process control block of each process. Each file descriptor may correspond to any of a number of different kernel entities. It does not have to refer to an actual disk file. Specifically, a file descriptor may refer to:

- One of the standard I/O streams. when the process is created these 3 descriptors are opened by default: `stdin`, `stdout` and `stderr`. In console applications they are usually associated with the console.
- A disk file.
- A special device file - these are usually mapped into the file system under the `/dev` directory. For example, a special file such as `/dev/audio` might be used to read and write audio data to and from a sound card.
- A pipe or fifo - these are the traditional IPC mechanisms for communication between processes on the same machine. Pipes are created with the `pipe()` system call and fifo's with the `mkfifo()` system call.
- A socket - a more general IPC mechanism than a pipe, which can be used between distributed processes.

Whatever the type of the underlying "file", we can read and write to it in a standard way using the generic `read()` and `write()` functions. Similarly we can close the file using the `close()` system call:

```
#include <unistd.h>
int close(int d);
```

Return 0 on success or -1 on error

Closing a file descriptor does not necessarily shut down the associated device. It is possible that more than one process has the file open at any one time since open file descriptors are inherited by child processes. Calling `close()` on the descriptor makes that descriptor unavailable to the current processes and decrements the system wide *reference count* on the file. Only if this reference count becomes 0 does any shutdown sequence occur. We will see that this can have important consequences for socket stream connections (such as TCP sockets) and also how the semantics of closing differs between UNIX-like and Win32 systems.

3.1.3 Shutting down a TCP socket

The shutdown sequence which occurs on TCP sockets is illustrated in the partial state transition diagram[2] of Figure 3.2, taken from RFC793 [37]:

From the diagram it is clear that the sequence of states is different depending on which side initiates the close. The left hand path from the `ESTABLISHED` state is followed by the side which performs the active close. This side's TCP sends a `FIN` segment to the peer, which starts the transition to `CLOSED` on the right hand side. An important consequence of actively closing is that the TCP ends up with it's control block suspended in a `TIME_WAIT` state for a period of typically 1 minute before the structure is removed from memory. This is to prevent new connections reincarnating the connection while there may still be stale TCP segments in the network belonging to the old connection.

The shutdown sequence is initiated automatically when the reference count on a descriptor becomes zero, but sockets can also be explicitly shut down with the `shutdown()` system call.

[2]The same diagram could be drawn using UML state diagram notation, but I have reproduced the original diagram "verbatim" here.

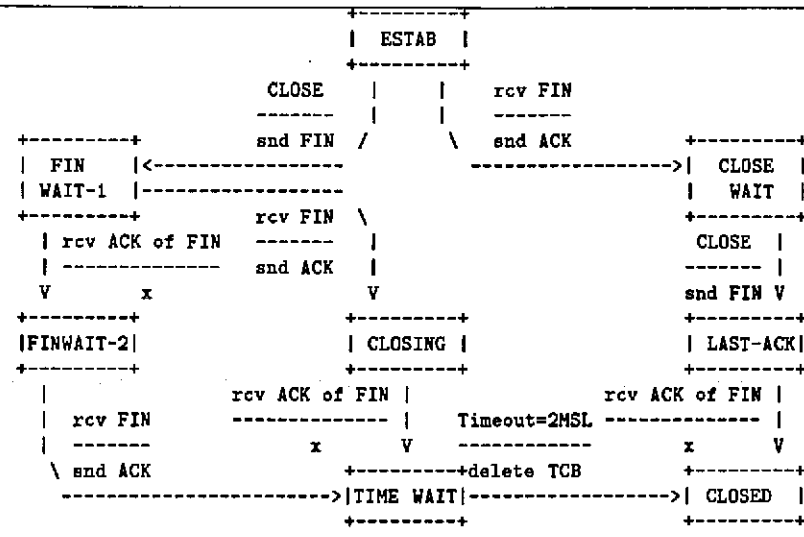


Figure 3.2: TCP Shutdown sequence

3.1.4 Controlling the I/O mode

An additional system call that can be used with all file descriptors, including sockets, is `fcntl()`. `fcntl()` provides an interface for setting or getting various operational parameters on open file descriptors, such as blocking/non-blocking mode, file locking (for synchronization), generation of the SIGIO signal when I/O is possible etc. A common use of `fcntl()` with sockets is for setting the blocking mode, but this can also be achieved with the `ioctl()` system call described below. Because `ioctl()` has a Winsock equivalent and `fcntl()` hasn't, it may be preferable to use `ioctl()` for this purpose if portability to Windows is an issue.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
```

Successful return value depends on cmd or -1 on error

3.2 What is a socket?

The socket was first introduced in the 4.2BSD kernel as a generalized IPC mechanism [24]. Sockets were designed to fit into the generalized UNIX I/O model where everything looks like

a file. The rationale was to allow programs to use the standard `read()`, `write()` and `close()` functions described above on sockets. What differentiates sockets from UNIX System V pipes and fifos, is that the sockets interface was designed to allow communication between processes on the local machine and also remotely through a supported communication domain. The consequent added complexity means that sockets can not be opened using the common `open()` sytem call used for files. The three system calls which can create new sockets are `socket()`, `socketpair()` and `accept()`. We will start this discussion by looking at the `socket()` call. `socketpair()` creates PF_LOCAL sockets, which are not related to networking so they are not discussed here. `accept()` is discussed in Section 3.7 below.

3.2.1 Creating a new socket

The socket() function

The prototype for the `socket()` call is shown below:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Returns fd on success, -1 on error

Whereas sockets are most commonly associated with TCP/IP[3] the interface is in fact very general, hence the three parameters to the `socket()` call:

domain describes the communication domain to be used. This should be set to PF_INET for use with the Internet Protocol (IP). On UNIX-like systems one can also use PF_LOCAL for local IPC. Other commonly supported domains include PF_APPLETALK, PF_X25, PF_ATM, PF_IPX etc. In the rest of this dissertation we shall focus on sockets in the PF_INET domain.

type The two most commonly supported types are SOCK_STREAM and SOCK_DGRAM for stream and datagram oriented communication respectively. Other types such as

[3]Many sockets implementations, including Winsock 1.1, only have TCP/IP support.

SOCK_RAW and SOCK_SEQPACKET are supported on some systems. Not all socket types are supported by all communication domains.

protocol Normally only a single protocol exists to support a particular socket type within a particular protocol family (for example, a PF_INET socket of type SOCK_STREAM implies TCP), so this can be left as 0. Otherwise a protocol from the list in /etc/protocols is specified.

We can call the `socket()` function to create new sockets like this:

```
int sock1, sock2;
sock1 = socket(PF_INET,SOCK_STREAM,0); // a stream oriented TCP socket
sock2 = socket(PF_INET,SOCK_DGRAM,0); // a message oriented UDP socket
```

Before we use `sock1` or `sock2` we should check that `socket()` returned a valid descriptor. If the call fails (eg. too many files already open or an invalid domain/socket type combination) it will return a value of -1. It is up to the application programmer to check for failure, determine the cause of error and take appropriate action. Failure to check return values of system calls is one of the most common causes of error in network programming. Dealing with error conditions is discussed in Section 3.2.5 below.

The place of sockets in the BSD system architecture

Figure 3.3 shows a simplified view of the resulting structures on a BSD type system (Linux is very similar) after a TCP socket such as `sock1` has been created. The most visible entity from the user process point of view is the file descriptor itself. The file descriptor references a unique socket structure. The socket structure itself is a relatively simple C struct. An important member of this struct is a pointer to an underlying protocol control block. The type of this control block is determined by the type of socket which was created - in this case a TCP control block. The TCP control block is yet another C structure which maintains all the state information about the particular TCP connection associated with the socket. The socket implements a convenient abstraction layer between the user process and the underlying network protocol.

Another useful and general functionality provided by this socket layer is buffering. Each socket has send and receive buffers associated with it. This shields the application from

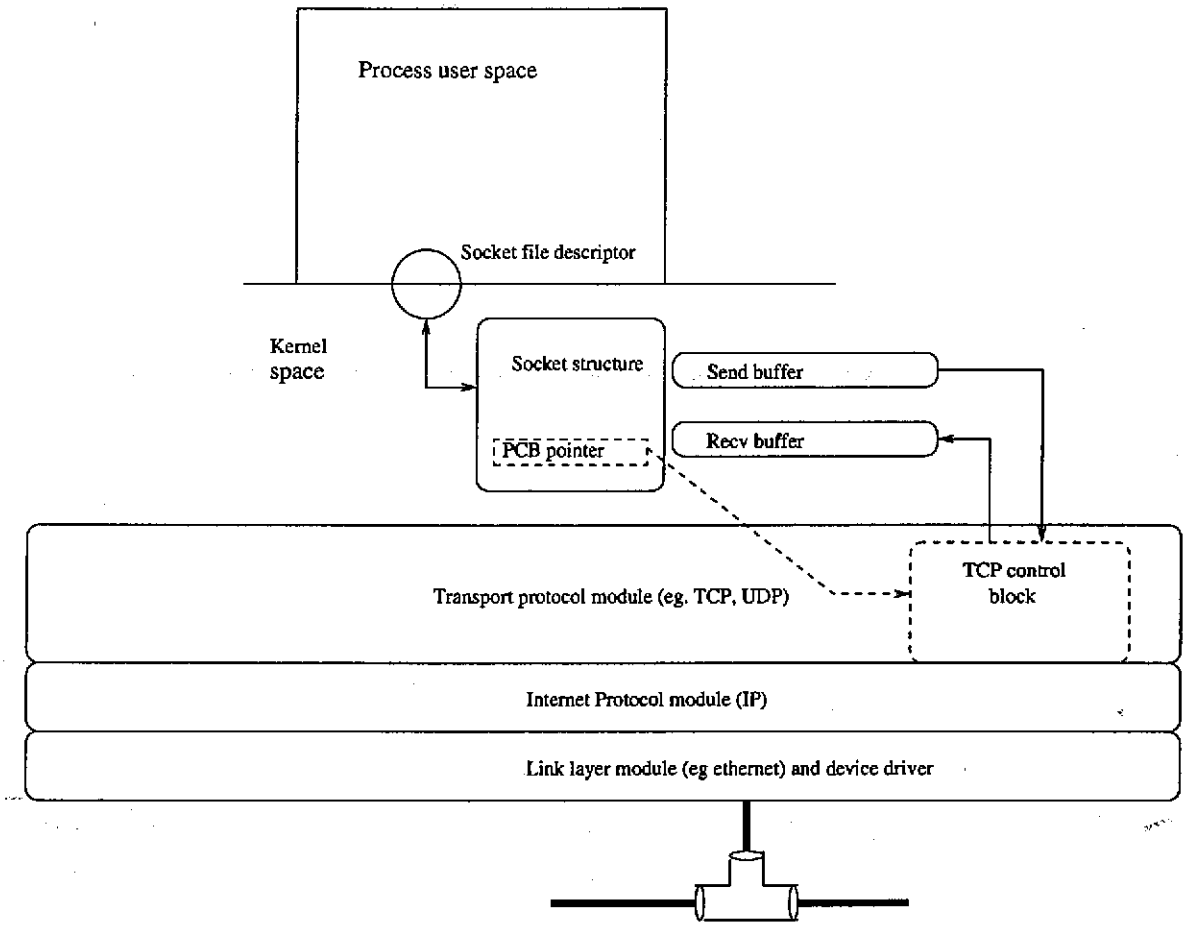


Figure 3.3: Position of a TCP socket

dealing with the complex asynchronicity of the physical network layer (see [50] for a good description of the rationale) as well as the potentially complex detail of the protocol implementation. When a user program successfully writes data to a socket, for example using the `write()` call described above, that data is simply copied down from the user's address space into the socket buffer. The detailed control of the movement of that data down the network stack and out on to the "wire" is taken care of by the participating protocol modules. Similarly, data that comes in along the "wire" is passed up the stack and accumulated into the socket's receive buffer. When the user process reads from the socket file descriptor, it simply copies data from the receive buffer up into its address space.

3.2.2 Socket addresses

Another complication introduced by making IPC communication channels operate across networks is addressing. In order for processes to communicate via sockets they must have a way of finding one another. Put another way, if an application sends out a message onto the internet, that message has to be able to find its way, not just to the correct destination host, but all the way to the correct socket receive buffer within that host, so that the receiving process which created the socket (or perhaps one of its descendents) can read the message through the appropriate file descriptor. Clearly a socket must have some form of address, and in the internet domain that address has two levels:

1. An IP address to identify the host (or, more correctly, the interface on that host) and
2. a port number which identifies the particular TCP (or UDP) control block and socket structure.

Binding

The process of associating an address with a socket is known as *binding*. The system call used to bind an address to a socket is, not unnaturally, called `bind()` and it's prototype is shown below:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr *addr, socklen_t
addrlen);
```

Returns fd 0 on success, -1 on error

In order to bind an address to a socket, we must create the address structure first; then call bind with the socket file descriptor, a pointer to the address and the length of the address structure as parameters. The process is complicated by two factors:

1. Different communication domains use different address structures. The type `sockaddr` is a sort of generic placeholder. For addresses in the PF_INET domain we must create an address of type `sockaddr_in` and cast the pointer to the type `*sockaddr` in the call

to bind. The reason for also providing the address length is that this can and does vary between different address domain types.

2. The address must be organized in a consistent byte order. Different machine architectures have byte ordering which is either big-endian or little-endian. The *endian-ness* on a particular host architecture is known as the host byte order. Because addresses must be interpreted consistently on these different architectures, binary values such as the IP number and port number, are first converted from host to network byte order before being sent down to the socket layer. Network byte order is always big-endian and is thus independent of the host architecture.

The `sockaddr_in` structure is shown below:

```
struct sockaddr_in {
    u_char  sin_len;
    u_char  sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

It is an awkward structure in a number of ways. The `sin_len` member is optional and not all vendors support it. It need not be set and, if it is present, is used only by internal kernel routines. The Posix.1g standard only requires the `sin_family`, `sin_addr` and `sin_port` members [55]. The `sin_zero` member is unused but must always be set to zero. The usual practice is to set the entire structure to zero, before filling in the required parts. The `sin_addr` member is a structure for historical reasons[4]: typically it is simply defined as a 32 bit integer.

The code below shows a socket being initialized and being bound to an address in the PF_INET domain. This small snippet illustrates a number of important points.

[4] 4.2BSD defined it as a union to facilitate access to the different parts of a class A, B or C addresses. With the advent of subnetting, the need for this fell away.

iterative/unix/simple.cpp

```
63    struct sockaddr_in servaddr;
64    memset(&servaddr,0,sizeof(servaddr));
65    servaddr.sin_family = AF_INET;
66    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
67    servaddr.sin_port = htons(Port);
68
69    if (bind(listenfd,(struct sockaddr*) &servaddr,sizeof(servaddr)) <0)
70      {
71        perror("bind");
72        exit (-1);
73      }
```

iterative/unix/simple.cpp

Figure 3.4: Initializing sockaddr_in

Note how `memset()` is used to zero out the structure first. A lot of legacy code uses the BSD `bzero()` function for this. The `memset()` function is a Posix standard function, is more portable (even to Windows) and is thus preferable.

Notice also that the two integer values `sin_port` and `s_addr` are stored in the structure in network byte order. The two functions `htons()` and `htonl()` are used to translate from host-to-network-short and host-to-network-long respectively.

The macro `INADDR_ANY` is used when we do not want to specify the IP address. This is typically done on server sockets, where we would not usually want to hardcode the IP address the server will listen on, particularly on a host machine with multiple interfaces.

The `bind()` function is called on line 69 to bind the address to a socket. Note that the function may fail so we check the return value and take appropriate action. In this simple example we simply print an error message and exit. The `perror()` function is described in Section 3.2.5 below.

3.2.3 Library functions used with addresses

An IP address is a 32 bit integer value. Besides the problem of byte ordering discussed above, it is also not a convenient format for human interpretation. The following two sections

describe library functions for converting between the raw IP address and more user friendly formats.

Conversion to presentation format

A useful set of functions exist for converting between network addresses and the more familiar ASCII formatted dotted string form of an address (eg: "192.168.0.2"). Legacy applications frequently make use of `inet_ntoa()` and `inet_addr()`.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr in);

                              Converts in_addr struct to dotted ASCII format

unsigned long int inet_addr(const char *cp);

     Returns binary equivalent (in net byte order) of dotted ASCII address
```

The newer functions `inet_pton()` and `inet_ntop()` are more flexible in that they can work with different address families, such as AF_INET6, but the former are more portable[5].

Host resolution

These two functions are commonly used to discover host information based on a name or address structure:

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int
type);

                              Returns pointer to a hostent struct or -1 on error
```

[5]Winsock, for example, only supports the legacy functions.

The hostent structure has the following form:

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

After calling gethostbyname(), the h_addr_list is usually the most interesting member of the hostent struct, as we are interested in finding the address(es) corresponding to the host name. Often we are not interested in multiple addresses a host might have. This is why the macro h_addr is defined, which references the first address in the null terminated array.

Address lookup example

The example below illustrates the use of gethostbyname() and inet_ntoa() to perform a simple name lookup:

lookup.cc

```
 7  int main(int argc, char *argv[])
 8  {
 9    if (argc!=2)
10      {
11        cerr << "usage: " << argv[0] << " <host>" << endl;
12        return -1;
13      }

14    hostent *host = gethostbyname(argv[1]);
15
16    if (host)
17      {
18        int i = 0;
19        in_addr *address;
20        while (address = (in_addr*) host->h_addr_list[i])
21          {
22            address = (in_addr*) host->h_addr_list[i];
23            cout << inet_ntoa(*address) << endl;
24            i++;
25          }
26      }
```

```
27    else
28      {
29        cerr << "Error: " << strerror(errno) << endl;
30      }
31    return 0;
32  }
```

lookup.cc

Figure 3.5: Address lookup example

3.2.4 Socket functions

The preceding sections looked at creating sockets and binding them to socket addresses. Sections 3.2.4 and 3.7 below describe the `connect()`, `listen()` and `accept()` functions for establishing connections with stream-oriented sockets. Section 3.7 then describes the family of `recv()` and `send()` functions used for socket I/O. Finally section 3.8 describes functions for setting and reading socket options.

Stream oriented sockets require that a connection be established before data can be read from or written to them. The sequence of function calls used to establish the connection differs depending on whether we are the active or passive participant. The active participant is the side which initiates the connection, usually the client process. The passive participant is the side which waits for and accepts connections, usually deemed the server.

Active connect

There are three steps required to actively establish a connection from a client to a server:

1. create a stream socket (eg. `int fd = socket(AF_INET,SOCK_STREAM,0);`),
2. create a sockaddr structure and populate it with the address of the server socket,
3. call `connect()`, passing a pointer to the sockaddr struct as a parameter, to connect the local socket with the remote socket.

Note that it is not required to bind the socket to an address prior to calling `connect()`. The `connect()` function has the following prototype:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *servaddr, int
addrlen);
```

Returns 0 on success, -1 on error

The `connect()` function will block by default until the connection is established. In the case of TCP, `connect()` initiates the TCP three-way handshake shown in Figure 3.6 by sending a TCP SYN segment.

There are a number of reasons why `connect()` might fail: there may be no reply to the SYN within a given time interval, there may be no process ready to accept the connection on the remote end (in which case the remote TCP returns an RST segment) or there may be no route available to the destination host. These conditions are indicated by the errors `ETIMEDOUT`, `ECONNREFUSED` and `EHOSTUNREACH/ENETUNREACH` respectively. See Section 3.2.5 below on detecting and handling errors.

If the connection is successfully established, `connect()` returns 0 and the client process may read and write through the file descriptor. The connection is uniquely identified by the

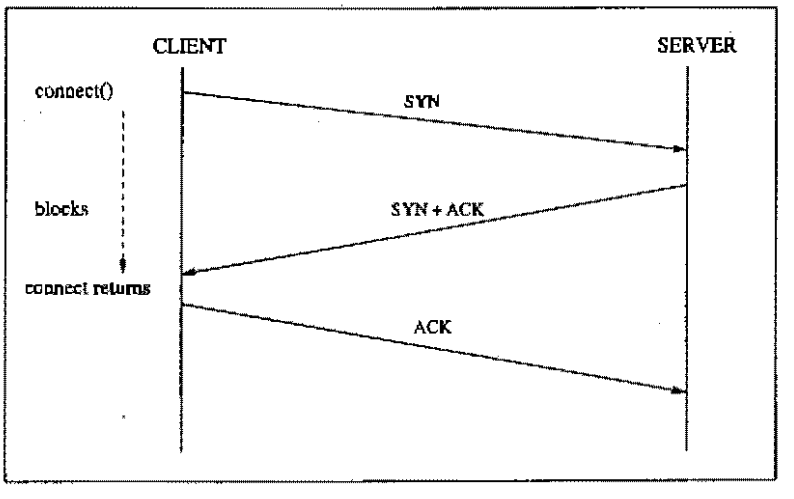


Figure 3.6: TCP 3 way handshake

4 parameters: <client IP, client port, server IP, server port>[6] .

The code below shows an example of a TCP client program using `gethostbyname()` and `connect()`:

client.cpp

```
16  int main(int argc, char* argv[])
17  {
18    if (argc!=4)
19      {
20        fprintf(stderr,"Usage: %s <host> <port> <nbytes>\n",argv[0]);
21        exit (-1);
22      }

23    unsigned int nbytes = atoi(argv[3]);
24    if (nbytes>=MAXBUF)
25      {
26        fprintf(stderr,"nbytes must be less than %d\n",MAXBUF);
27        exit (-1);
28      }

29    int port = atoi(argv[2]);

30    // lookup the address of the server host
31    hostent *host;
32    if (!(host = gethostbyname(argv[1]))) // look up host address
33      {
34        perror("gethostbyname");
35        exit (-1);
36      }


37    // fill in a sockaddr_in structure
38    struct sockaddr_in servaddr;
39    memset(&servaddr,0,sizeof(servaddr));
40    servaddr.sin_family = AF_INET;
41    memcpy(&servaddr.sin_addr.s_addr,(host->h_addr),sizeof(in_addr));
42    servaddr.sin_port = htons(port);

43    // create a socket
44    int connfd;
45    if ((connfd = socket(PF_INET, SOCK_STREAM,0)) < 0)
46      {
47        perror("socket");
48        exit (-1);
49      }
```

[6]This 4 parameter tuple is commonly known as a full association.

```
50
51    if (connect(connfd, (struct sockaddr*) &servaddr, sizeof(servaddr))<0)
52      {
53        perror("connect");
54        exit (-1);
55      }

56    // now we're connected ...
```

client.cpp

Figure 3.7: Active connect

Passive accept

The process of setting up a stream socket to passively accept connections is quite different. This time there are five steps which must be taken:

1. the stream socket is created with the `socket()` function as before;
2. a sockaddr structure must be created and filled in with the server socket address. In the case of TCP we must specify the port the server will listen on and the `inaddr` (usually `INADDR_ANY`);
3. the sockaddr must be *bound* to the socket by calling `bind()`;
4. the socket must be assigned a listen queue by calling `listen()`;
5. the process waits for incoming connections by blocking in a call to `accept()`.

We have already seen in Section 3.2.2 how to bind an address to a socket. The `listen()` function is used to declare a willingness to accept connections and to create a queue for handling incoming connections. The prototype is shown below:

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Returns 0 on success, -1 on error

The backlog parameter is used to specify the number of established connections which can be queued before new client connections are refused. It is complicated by the fact that there are actually two queues: a queue of partially established connections where the SYN has been received from the client and a queue of fully established connections where the final ACK of the 3 way handshake has been received. Different systems interpret the backlog parameter differently in this regard. Historically (eg on 4.2BSD) it referred to the combined length of both queues [55]. Linux kernels, as of v2.2, interpret it to mean the number of the established connections only.

The maximum value of the backlog parameter is also system dependent. Historically it was limited to 5, but most modern systems allow larger values such as 128. One of the differences between Windows NT (and derivatives) workstation and server editions is that the backlog parameter is limited to 5 on the workstation. If a call to `listen()` specifies a larger value than the maximum, this value is silently truncated to the system limit.

After the call to `listen()`, connections from clients can be queued on the listening socket, but we now need a means to dequeue them. The function that dequeues an incoming connection, and creates a new socket in the process, is `accept()`.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peeraddr, int
*addrlen);
```

Returns fd of new connected socket on success, -1 on error

The file descriptor returned from `accept()` refers to a new socket (`accept()` behaves like a factory method for connected sockets). This socket has the same address and port number as the listening socket, but TCP uniquely identifies it, as before, by also considering the address and port of the remote end of the connection.

The second and third parameters to `accept()` are optional. If the `peeraddr` parameter is provided, the structure will be populated with the remote address upon successful return from `accept()`. It is possible to set this parameter to NULL, thus avoiding the copying of

this data from the kernel[7] on `accept()`.

The code below shows the initialization section of a server program illustrating the use of `bind()`, `listen()` and `accept()`. The setting of the `SO_REUSEADDR` socket option is a recommended practice[55] to allow a server which has terminated, to be immediately restarted and bound to the same address, even though existing connections which are using the address may still survive. The use of socket options is discussed further in Section 3.8 below.

simple.cpp

```
34    if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) <0)
35      {
36        perror("socket");
37        exit (-1);
38      }
39
40    struct sockaddr_in servaddr;
41    memset(&servaddr,0,sizeof(servaddr));
42    servaddr.sin_family = AF_INET;
43    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
44    servaddr.sin_port = htons(Port);

45    int opt = 1;
46    if (setsockopt(listenfd,SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <0)
47      {
48        perror("setsockopt");
49        exit (-1);
50      }

51    if (bind(listenfd,(struct sockaddr*) &servaddr,sizeof(servaddr)) <0)
52      {
53        perror("bind");
54        exit (-1);
55      }
56
57    // assign LISTENQ to socket
58    if (listen(listenfd,LISTENQ)<0)
59      {
60        perror("listen");
61        exit (-1);
62      }
63
64    // loop processing client connections
```

[7] If the server connection handler code requires the address of the remote peer it can call `getpeername()` at any stage.

```
65    struct sockaddr_in peeraddr;
66    while(1)
67      {
68        socklen_t addrsize = sizeof(peeraddr);
69        connfd = accept(listenfd, (struct sockaddr*)&peeraddr, &addrsize);
70        if (connfd<0)
71          {
72            perror("accept");
73            continue;
74          }
75
76        printf("Accepted connection from %s:%d\n", \
77               inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));
```

simple.cpp

Figure 3.8: Passive accept

In the code above `LISTENQ` is a constant. The `accept()` function is called with all 3 parameters non-NULL so as to return the information about the remote peer.

Specialized Socket I/O

Even though the `read()` and `write()` system calls can be used to transfer data on a socket, as discussed in Section 3.1, there are six additional specialized I/O functions which exploit socket-specific characteristics. These are shown in the textbox below.

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int s, const void *msg, int len, unsigned int flags);
int sendto(int s, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen);
int sendmsg(int s, const struct msghdr *msg, unsigned int flags);

int recv(int s, void *buf, int len, unsigned int flags);
int recvfrom(int s, void *buf, int len, unsigned int flags,
           struct sockaddr *from, int *fromlen);
int recvmsg(int s,struct msghdr *msg, unsigned int flags);
```

send/recv data on a socket. Returns -1 on error

The `send()` and `recv()` calls are analagous to `read()` and `write()` and can only be used with connected stream sockets. The difference is that these calls allow an additional `flags` parameter. Flags are an OR'd combination of defined constants such as `MSG_OOB` (for reading or writing out of band data) and `MSG_PEEK` (to check the contents of the receive buffer without copying the data). The set of supported flags is system specific so care must be taken to remain portable. Winsock v2, for example, defines only the two mentioned above, whereas Linux defines `MSG_WAITALL`, `MSG_NOSIGNAL` and others which have no equivalent in Winsock.

Connectionless sockets (such as UDP SOCK_DGRAM sockets) typically use `sendto()` and `recvfrom()`, where there is a need to specify a destination address to send to, or to identify a source address on a received message.

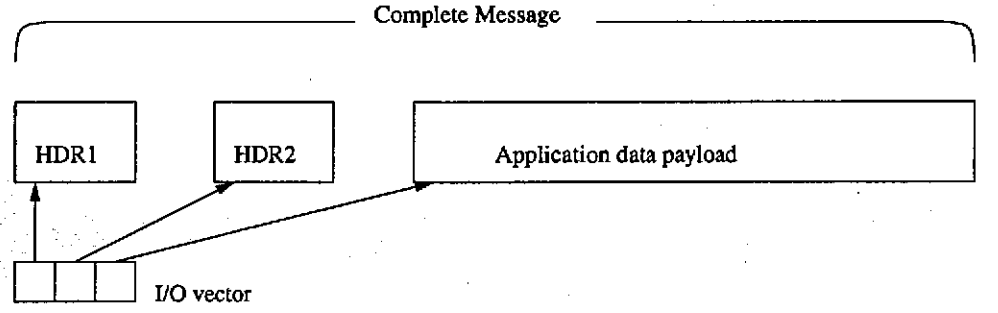


Figure 3.9: Scatter/Gather I/O using `sendmsg()` and `recvmsg()`

Both `sendmsg()` and `recvmsg()` take a pointer to a `msghdr` struct as a parameter instead of a simple `void*`. The `msghdr` is defined in `<sys/socket.h>`. The most notable characteristic of this structure is that it defines a vector of buffers, rather than a simple buffer pointer, which can be used for scatter/gather I/O. This I/O mode is useful to avoid excessive data copying when assembling or disassembling message blocks[8].

Consider, for example, an application which implements a simple protocol stack. An application message is generated and two protocol layers each generate a header to prepend to the message. The resulting message structure is illustrated in Figure 3.9. If we were to use the simple `send()` (or `write()`) to send the message we would either have to call `send()`

[8]The `readv()` and `writev()` system calls provide a more generic method for performing scatter/gather I/O on file descriptors.

three times to send each part, or else we would first have to copy the three parts into a single contiguous buffer space and then call `send()`. Both of these options are unattractive. Using `sendmsg()` instead, we can avoid both of these sources of inefficiency, by simply passing it the vector that points to the three areas of memory containing our composite message. The `sendmsg()` function *gathers* the data and writes it down to the socket send buffer. The reverse process, with `recvmsg()`, is to *scatter* the data from the socket receive buffer into the buffers pointed to by the vector.

Options

Whereas it is convenient for the UNIX programmer to treat sockets, disk drives, disk files, terminals and audio devices as if they were all simply files, there are occasions when it is desirable, or even necessary, to recognize that they are different. The kernel code which implements sockets as well as device drivers generally provides *hooks* which the programmer can use to fine tune the characteristics of the underlying device. These hooks are accessed via the `ioctl()` system call.

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...)
                                                    returns -1 on error
```

Ioctls are extremely system dependent, particularly where they refer to hardware device drivers. Stevens [55] describes the `ioctl()` function as

> "the system interface used for everything that didn't fit neatly into some other nicely defined category."

Ugly though this concept may be, ioctls are frequently used in network programs to obtain information on host interfaces, access to routing tables etc. A common ioctl which is used with sockets is `FIONBIO` to set or clear the nonblocking flag. For example, one can set a socket in non-blocking mode with:

```
int flag = 1;
ioctl(sockfd, FIONBIO, &flag);
```

An additional mechanism for manipulating socket specific options is provided by the `setsockopt()` and `getsockopt()` functions. Again the options supported by different systems varies, but there is a significant overlap of commonly used options. The prototypes for these functions are shown below:

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
           socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
           socklen_t optlen);
```

returns 0 on success, -1 on error

Like `ioctl()`, these are untidy functions. The second parameter, `level`, allows the programmer to specify at which level the option refers to. Typical constants defined for this parameter are `SOL_SOCKET`, `IPPROTO_TCP` and `IPPROTO_IP` which refer to the socket level, TCP level and IP level respectively. The third parameter, `optval`, is defined as a `void*` because the option data value can be of different types. In most cases it is a simple integer, but there are exceptions. The `SO_RCVTIMEO` and `SO_SNDTIMEO` options, for example, set (or get) an option of type `struct timeval`.

The simple server example of Figure 3.8 illustrated the use of `setsockopt()` to set the `SO_REUSEADDR` option on a listening socket.

3.2.5 Handling errors

Most system calls, including those relating to sockets, return either 0 or a positive integer value on success. A return value of -1 indicates an error condition. Robust code should detect and handle such error conditions gracefully, so it is common practice to code such system calls within an `if` statement like the following:

```
if (connfd=accept(sockfd, &addr,  &addrsize) <0)
  ... handle error
```

Code	Symbol	Description
4	EINTR	Interrupted system call
11	EAGAIN	Try again - would block
77	EBADFD	Invalid file descriptor
88	ENOTSOCK	Socket operation on non socket
98	EADDRINUSE	Address already in use
110	ETIMEDOUT	Connection timed out
111	ECONNREFUSED	Connection refused
112	EHOSTDOWN	Host is down

Table 3.1: Some common POSIX error codes

Errno

Whereas the return value of -1 gives an indication of error, it does not give any indication of the cause of the error. In order to determine the cause of the error condition, it is necessary to examine the value of **errno**. The variable **errno** is traditionally defined as a static global integer[9]. This mechanism is adequate for single threaded processes, but is problematic where multiple threads co-exist within the same process. If each thread shares the same global **errno** value, and they are each making system calls, there is no way to determine which thread caused which error. For this reason, most modern implementations define **errno** as a macro which actually refers to a thread specific error value, rather than a global static integer.

The ISO C and POSIX.1 standards list error codes and their corresponding symbolic names. Table 3.1 lists a small selection of common error codes. The UNIX manual pages for the various socket related functions, by convention, have an Errors section which lists the possible error conditions which can arise from that function.

[9]UNIX system calls are implemented via a software interrupt (0x80). The error value is returned on the stack and copied into the global **errno** variable.

Errno helper functions

There are two functions which convert the integer error values into a more human readable format.

```
#include <string.h>
char* strerror(int errnum);

                                returns human readable string describing error
```

The `strerror()` function is useful for composing log messages describing error conditions which may have arisen. For debugging purposes, frequently it is only required to dump a trace message to the standard error stream. The `perror()` function provides a simple means to do this:

```
#include <stdio.h>
void perror(const char*s);

                                        writes error message to stderr
```

`perror()` writes out the string `s`, followed by a colon, followed by the stringified error message to the standard error stream. Typically the string `s`, might contain the name of the function, a timestamp, line number or other useful trace information.

3.3 Windows Sockets - the Winsock specification

Winsock version 1.1 was the standard sockets API on Windows since its release in January 1993. Winsock was an open specification designed by a group of interested industry vendors. Thus, at least initially, Winsock was not owned by Microsoft or even distributed with its popular operating systems. Windows NT prior to version 4.0 did have a built in TCP/IP transport module apparently based on the UNIX System V STREAMS environment[6][41]. The version 1.1 specification was limited in scope to TCP/IP sockets and provided a common basis for TCP/IP stack vendors to provide compatible implementations.

The release of the Winsock version 2 specification in 1996 reflected the major upheavals which had taken place in the network industry. The Internet's popularity had continued to

explode. Microsoft began distributing a free TCP/IP implementation with all its operating sytems, dealing a blow to the majority of 3rd party stack vendors. The last revision of the publically available specification[11] apears to be revision 2.2.2 (dated August 7, 1997). Companies listed in the acknowledgements section of this document include Microsoft, Intel, FTP Software, Distinct, Turbosoft, Motorola, Novell, DEC, ICL, Stardust Technologies and SunSoft. The complete Winsock 2 specification consists of four documents:

1. Windows Sockets 2 Application Programming Interface
2. Windows Sockets 2 Protocol-Specific Annex
3. Windows Sockets 2 Service Provider Interface
4. Windows Sockets 2 Debug-Trace DLL.

The Application Programming Interface is the most interesting from an application programmer's perspective. I have included the publically available Winsock revision 2.2.2 specification in the electronic appendix on cdrom. There has been no further revision to the public specification. Current documentation on Winsock is now integrated into the Microsoft Developer Network (MSDN) documentation. The few minor additions to the original specification are flagged in this documentation as 'Microsoft specific'.

3.3.1 Architecture

Though the scope of this dissertation is restricted to issues relating to the top level API, a brief architectural summary is in order.

One of the more interesting aspects of Winsock v2 is its architectural overhaul to adopt the Windows Open Systems Architecture (WOSA). WOSA separates the API from the protocol service provider, presenting a layered architecture, as illustrated in Figure 3.10. Winsock v2 provides two programming interfaces: the Winsock API and the Winsock SPI (Service Provider Interface). In this model, the Winsock DLL provides the standard API, and an independent vendor can install its own service provider layer underneath. Whereas in Winsock v1.1, vendors would supply a replacement Winsock DLL, WOSA is structured

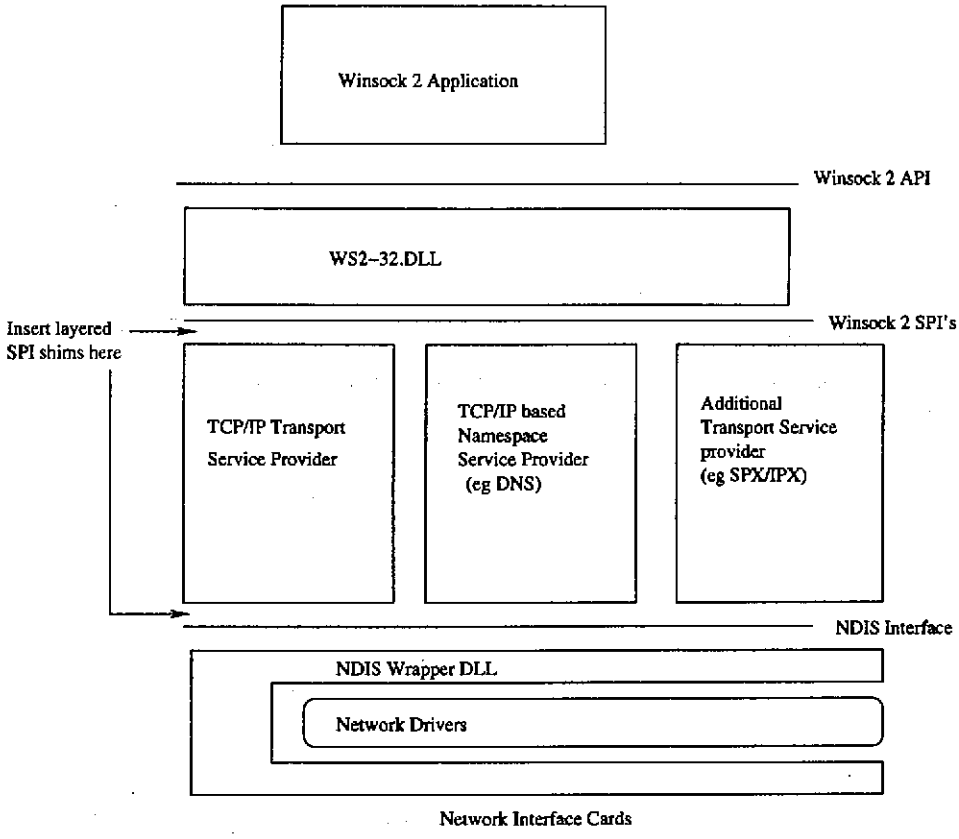


Figure 3.10: Winsock 2 WOSA architecture

much like the UNIX STREAMS environment[10] . The "plug-in" and "pile-on" decoupling of layers presents opportunities for vendors to interleave additional *shim* layers (to provide encryption or accounting services for example) between the Winsock DLL and the base transport service provider.

The bottom layer service provider module in turn talks to the NDIS (Network Driver Interface Specification) interface[6]. NDIS provides an abstraction which shields the transport protocols from the device driver details and vice versa. NDIS is part of the original Windows NT, STREAMS based, network architecture which has survived and been incorporated into the newer WOSA architecture.

[10]Perhaps this is the answer to the mysterious disappearance of STREAMS from the documentation - the STREAMS model was simply dusted off, tidied up and renamed WOSA!

3.3.2 Features

The Winsock specification is too large to provide an effective summary here. The full Winsock API release 2.2.2 specification is provided in the electronic appendix, so I have not attempted to repeat it here. Of particular interest in the specification is the section entitled "Deviation from BSD Sockets", which outlines main programming considerations when porting BSD sockets code. I have given an overview below of features introduced in Winsock 1.1 followed by those introduced in Winsock 2 together with some historical rationale where appropriate.

Two structural differences one finds between Windows sockets and BSD code relate to header files and initialization. Whereas some backward compatible headers do exist, it is normal practice to simply include `<winsock.h>` or `<winsock2.h>`. Before using any sockets functions it is necessary for a process to load the Winsock DLL using the `WSAInit()` function.

Winsock 1.1

The emphasis of Winsock v1.1 was to implement the BSD sockets paradigm for TCP/IP sockets on Windows in a manner which would ease porting of existing BSD sockets applications to Windows. It provided most of the socket primitives such as the `socket()`, `connect()`, `accept()`, `send()`, `recv()`, `shutdown()` , `getsockopt()` and `setsockopt()` calls. Two notable exceptions, `close()` and `ioctl()`, were renamed to `closesocket()` and `ioctlsocket()` respectiveley, to avoid clashes with existing Windows API functions.

The BSD `select()` function was implemented for synchronous demultiplexing of socket events (not generic file events as on UNIX) but its use was initially discouraged. The early Windows 3.1 system, which Winsock v1.1 was targeted at, did not support preemptive multitasking. This meant that any blocking call, even a call to `select()`, could effectively starve the system. In keeping with the Windows GUI message driven paradigm[40], a new function `WSAAsyncSelect()` was introduced, which arranged for messages to be placed on a window's message queue in response to network events occurring on a socket.

```
#include <winsock.h>

int WSAAPI
WSAAsyncSelect (
            IN SOCKET s,
            IN HWND hWnd,
            IN unsigned int wMsg,
            IN long lEvent );
```

This model proved popular due to its easy integration with the Windows GUI programming paradigm[11]. The socket was set automatically to non-blocking mode after calling `WSAAsyncSelect()` to avoid the risk of blocking calls locking up the system. It is somewhat simpler to use than the BSD `select()` in that the message which is placed on the window message queue contains an explicit indication of the event which has occurred (typically FD_READ, FD_WRITE, FD_ACCEPT, FD_CLOSE or FD_CONNECT) together with the relevant socket handle. One of the problems with `select()` is that it only returns an indication of the *number* of interesting events that have occurred. The file descriptor sets still have to be scanned to find the *actual* events which have occurred on each descriptor [4].

Winsock 2

Besides the structural overhaul, Winsock version 2 adds a number of new features to the Winsock specification. Chief among these are:

1. Separation of the sockets interface from the transport and name service providers. A consequence of this is that Winsock 2 sockets are no longer restricted to TCP/IP and can access various name services such as DNS, NIS, X.500 and SAP in a standardized way.

2. Support for advanced Windows NT I/O and synchronization mechanisms. Specifically this means that sockets (like other file handles) can be used with overlapped I/O and Win32 event objects.

[11]The Windows port of the Tcl notifier uses `WSAAsyncSelect()` for the same reason - easy integration with Tk GUI events.

3. A Quality of Service mechanism for use with transport service providers which support it. This is derived from the flow specification described by Craig Partridge in RFC1363. The structures below illustrate how a flowspec is specified:

```
typedef struct _flowspec
{
        int32   TokenRate;
        int32   TokenBucketSize;
        int32   PeakBandwidth;
        int32   Latency;
        int32   DelayVariation;
        GUARANTEE       LevelOfGuarantee;
        int32   CostOfCall;
        int32   NetworkAvailability;
} FLOWSPEC, *LPFLOWSPEC;

typedef struct _QualityOfService
{
        FLOWSPEC        SendingFlowspec;
        FLOWSPEC        ReceivingFlowspec;
        WSABUF  ProviderSpecific;
} QOS, *LPQOS;
```

4. Scatter and gather operations are supported, not through `sendmsg()` and `recvmsg()`, but rather as part of the generic Overlapped I/O facility. Overlapped I/O is supported through `WSASend()`, `WSASendto()`, `WSARecv()` and `WSARecvFrom()`.

There are other enhancements listed in the specification, such as multicast and socket sharing across processes, but the above represents some of the more important changes from Winsock 1.1.

3.3.3 Error reporting

The WIN32 API is inconsistent with respect to reporting errors when functions fail [19]. Failure can be indicated by a return value of 0 or -1 depending on the function. A number of commands return the exit code with 0 indicating success.

Whereas the Microsoft C library provides both the `errno` constant and the `perror()` function, the recommended (and more reliable) means of retrieving the exit code is via the WIN32 `GetLastError()` function, or the Winsock equivalent, `WSAGetLastError()`.

The Windows exit codes are not the same as the equivalent UNIX `errno` values. A common feature of all the porting libraries considered is therefore, that they provide some mechanism for interpreting WIN32 and POSIX error codes in a platform independent manner. The most common approach is to simply translate the WIN32 error codes into POSIX equivalents. A clear illustration of this process can be seen in the file `tclWinError.c` from the Tcl source code.

3.4 Synchronisation and process control

It should be clear from the foregoing discussion that the Windows sockets API has functional equivalents for most of the BSD sockets functionality. Indeed most of the existing BSD socket calls can be used unchanged, by simply linking against the Winsock functions of the same name. Minor discrepancies, such as with `closesocket()` and `ioctlsocket()`, should be easily handled through the use of preprocessor macros or thin wrapper functions.

Unfortunately, the opening and closing of and exchanging of data with sockets is only one aspect of the design of socket based servers. Most useful servers have to be able to successfully service multiple concurrent client connections. Strategies for implementing architectures which solve this problem involve more fundamental aspects of the underlying operating system than simply the sockets API. In particular we may need to be able to:

- create threads and/or processes;
- communicate and synchronize between these;
- detect and demultiplex events such as I/O readiness and timeouts.

In Chapter 4 we will see how the tools presented in Chapter 2 provide useful abstractions or emulations of these mechanisms on the target operating systems.

Chapter 4

Server Architectures

This chapter describes portable implementations of the following server architectures:

- Iterative service
- Event driven service
- Thread (or process) per connection service
- Thread (or process) pool service

As indicated in Section 1.4, architectures based on Asynchronous or Overlapped I/O models are not considered. The classification above is also somewhat simplistic in that some of the more interesting architectures that have emerged in the research community in recent years employ hybrid models. Examples include the Jaws web server [14], which can dynamically adapt its concurrency strategy based on load conditions, and the Flash [58] web server which uses a blocking thread-pool model for disk I/O and an event driven model for network I/O.

The UNIX event driven servers are all based on the `select()` system call. Whereas it is interesting to look at some of the newer event mechanisms such as `/dev/poll` on Linux[39] and FreeBSD kqueues, it is beyond the scope of this research to do a comprehensive analysis of UNIX event mechanisms. I have thus concentrated on the more primitive `select()` function because it is widely available.

4.1 Iterative server architecture

Each of the servers developed in this and later chapters performs a simple http-like service. The client receives a simple "Hello" message[1] before sending a request to the server in the form of an ASCII encoded decimal number occupying exactly 4 bytes (eg 0600). The server responds by returning a stream of 'A's of the requested length (eg 600 for the example given). Unlike http, the server does not terminate the connection. The client performs the active close, thus ensuring that the TIME_WAIT state occurs on the client side of the connection. This is a useful consideration when testing, if we want to bombard the server with a large number of connections per second. This service is almost identical to that used by Stevens [55] to illustrate server architectures.

The iterative version performs the minimum amount of process control. It uses a single thread of control and blocking I/O, which forces it to serialize multiple connections. The main loop for such a server is shown below:

iterative/unix/simple.cpp

```
82    // loop processing client connections
83    while(1)
84      {
85        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
86        if (connfd<0)
87          {
88            perror("accept");
89            continue;
90          }
91
92        if (send_n(connfd,hello,7)<0)
93          {
94            perror("send");
95            close(connfd);
96            continue;
97          }
98
99        switch (read_n(connfd, inbuf, 4))
100         // read_n must return 4, 0 or -1
101         {
102         case 4:
103           if (send_n(connfd,outbuf,atoi(inbuf))<0)
104             {
105               perror("send");
```

[1]The reason for this initial greeting is to avoid a race condition described in Section 5.5.2.

```
106                 break;
107               }
108             // wait for client to close
109             recv(connfd, inbuf, 1,0);
110             break;
111           case 0:
112             fprintf(stderr,"Odd: client closed\n");
113             break;
114           default:
115             perror("recv");
116             break;
117           }
118         close(connfd);
119       } //while
120     return 0;
121   }
```

iterative/unix/simple.cpp

Figure 4.1: Iterative service

Multiple concurrent connections will be queued on the listening socket queue. These are `accept()`'ed and processed one at a time. Clearly the scheduling is inefficient. Ready connections will be ignored while the active connection is being processed. Given that this process may block, particularly in a wide area, internet environment, the latency experienced by these waiting clients may be unacceptably high. The overall server throughput in these circumstances will also be unacceptably low as available CPU cycles are not being used to process ready connections.

4.2 Event driven server architecture

4.2.1 Overview

Event driven servers deal with multiple connections within a single process. They do not in themselves exhibit any concurrency. All I/O is done in non-blocking mode and the process needs to use some form of I/O demultiplexing and event dispatching mechanism. On UNIX systems this demultiplexing is achieved using the `select()` or `poll()` function calls. There are problems of scalability with both of these functions [3]. Event driven I/O architectures

have historically not been very popular on UNIX platforms, perhaps because of the lack of a proper explicit event delivery mechanism [4]. POSIX Real-time signals [43] provide a model for flexible event delivery and notification, but are not yet widely available on all systems.

On Windows versions prior to Windows 95, there was no pre-emptive multi-tasking, making event driven architectures the natural (if not the only) choice. Winsock 1.1 does have an implementation of the BSD `select()` function, but there are more sophisticated Winsock specific alternatives. Winsock 1.1 introduced a function, `WSAAsyncSelect()`, which causes notification messages to be posted to the application window's message queue when I/O events of interest occur. This is the same mechanism used to handle GUI events.

Winsock 2 with Windows 95/98/NT introduced a new mechanism based on the functions `WSAEventSelect()` and `WSAWaitForMultipleEvents()` or `WaitForMultipleObjects()`, which allow for explicit event delivery to Windows applications without making use of the Windows message queue [26].

In this section I show how the substantial differences between these modes, can be masked by abstracting away from the detail and concentrating on the architectural framework. To illustrate I present an example using the ACE toolkit and an example using the Tcl library. I have also implemented native `select()` based versions of these servers. The source code can be seen in the electronic appendix on CDROM.

4.2.2 Implementation using ACE

ACE provides a wide range of classes which are designed to be flexibly grouped together in collaborations to form application frameworks. Developing software using the ACE library involves identifying the participants and collaborations required to implement the design. If the design follows a well-known pattern[2] then one can almost always find a templatized class within ACE which encapsulates it.

View from the top

Figure 4.2 describes a very high level view of the architecture we are implementing. There are three participating classes in the server:

[2]For more information on factories and other patterns see[8].

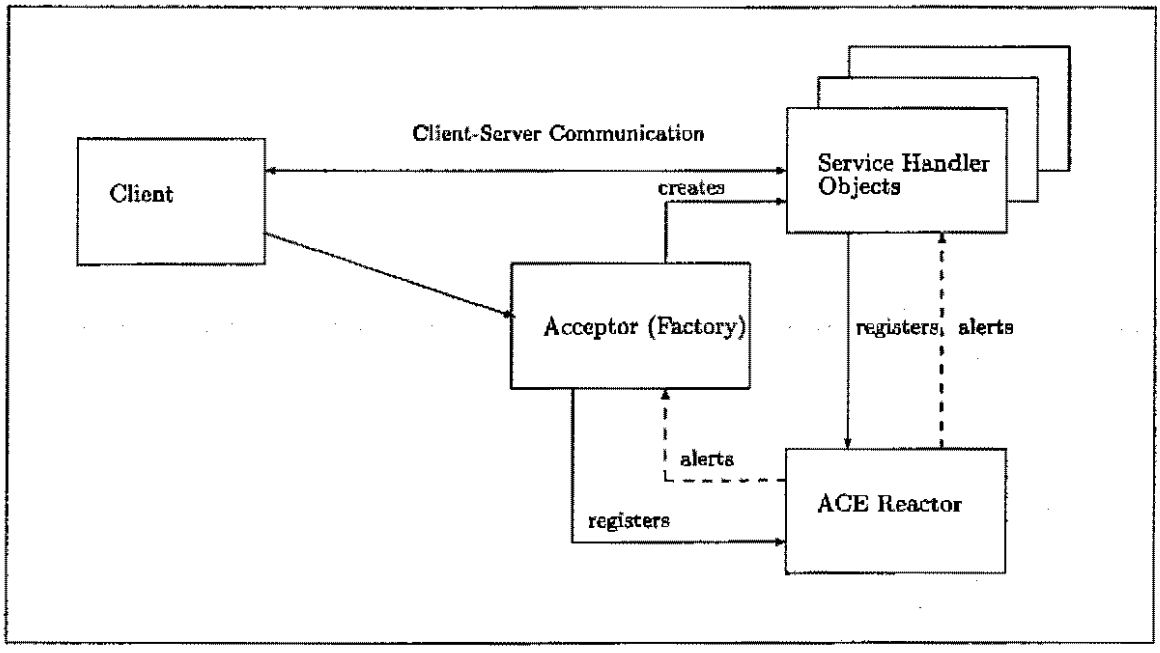


Figure 4.2: An Acceptor Factory

1. An Acceptor object. The acceptor is a factory class, which accepts new connections and creates Service Handler objects to handle these connections.

2. Service Handler objects. These are the workers in the pattern. They are responsible for providing the service to the client. This should be the only part of the design which is application specific.

3. The Reactor. Because we are implementing a single process event driven server we need an object which responds to events and dispatches messages to the Acceptor and Service Handler objects. This is the role of the Reactor. The Acceptor and Service Handlers must register with the Reactor in order to receive notifications.

Concretizing the abstract classes

Having clarified our high level design, we now need to fill in the details. The Acceptor we have discussed above is deliberately a very abstract object. We can concretize it by saying

that we need an Acceptor which listens for TCP/IP connections and creates our own user defined Client_Handler objects.

select/ace_select/server.cpp

```
10  typedef ACE_Acceptor <Client_Handler, ACE_SOCK_ACCEPTOR> Client_Acceptor;

11  static sig_atomic_t finished = 0;

12  extern "C" void handler (int)
13  {
14    finished = 1;
15  }

16  int main (int argc, char *argv[])
17  {
18    if (argc!=2)
19            ACE_ERROR_RETURN((LM_ERROR,"%p\n","port no"),-1);
20    u_short port = atoi(argv[1]);
21    ACE::set_handle_limit(1024);
22    ACE_Select_Reactor *sreactor = new ACE_Select_Reactor((ACE_Sig_Handler*) 0,0,1,0,0);
23    ACE_Reactor *reactor = new ACE_Reactor(sreactor);
24
25    Client_Acceptor my_acceptor;

26    if (my_acceptor.open (ACE_INET_Addr (port),
27                          reactor,
28                          ACE_NONBLOCK) == -1)
29      ACE_ERROR_RETURN ((LM_ERROR,"%p\n","open"),-1);


30    ACE_Sig_Action sa ((ACE_SignalHandler) handler, SIGINT);

31    // Process events ...
32    while (!finished)
33      {
34        reactor->handle_events ();
35      }

36    delete reactor; delete sreactor;
37    return 0;
38  }
```

select/ace_select/server.cpp

Figure 4.3: ACE event driven server main()

The result, implemented in code, is shown in Figure 4.3. The strength of ACE is shown in the typedef at the top of the file. This simple declaration actually does most of the work of defining our framework. Inside the `main()` function we simply declare a reactor, declare an acceptor of our user defined type, register it with the reactor and listen for events. All that remains is to implement the `Client_Handler` class.

The `Client_Handler` (shown in Figure 4.4 below)is derived from an `ACE_Svc_Handler`. This is a necessary relationship in order to allow our `Client_Handler` object to register with the reactor.

select/ace_select/client_handler.h

```
 9  class Client_Handler :
10  public ACE_Svc_Handler <ACE_SOCK_STREAM, ACE_NULL_SYNCH>
11  {
12  public:
13    Client_Handler (void) {;}
14    // void destroy (void);
15    int open (void *acceptor);
16    int handle_close (ACE_HANDLE handle,
17                      ACE_Reactor_Mask mask);
18  protected:
19    int handle_input (ACE_HANDLE handle);
20    int handle_output (ACE_HANDLE handle);

21    int bytes_to_send, bytes_sent;
22    int bytesread;
23    char out_message[10000];
24    char in_message[5];
25    ~Client_Handler (void) {;}
26  };
```

select/ace_select/client_handler.h

Figure 4.4: ACE event driven client handler

There is a lighter weight solution by inheriting from an `ACE_Event_Handler` instead, but there are some additional benefits one derives from using the `Svc_Handler`. Chief among these is that the `Svc_Handler` provides us with a built in `ACE_Stream` object which we can use to communicate with the client. The `Svc_Handler` is also derived from an `ACE_Task` object, which would allow us to very easily adapt this server to a multi-threaded design[3].

[3]The ACE class hierarchy is quite extensive and beyond the scope of this work. The interested reader is

Providing the service

To provide our application specific functionality we simply override the `destroy()`, `open()`, `handle_close()`, `handle_input()` and `handle_output()` methods. For the sake of brevity, I only show the `handle_input()` method in Figure 4.5. Note that the reactor ensures that these methods are called in response to I/O events on the underlying socket. We remain registered with the reactor so long as we return 0 from these methods.

select/ace_select/client_handler.cpp

```
 4  int
 5  Client_Handler::open (void *_acceptor)
 6  {
 7    Client_Acceptor *acceptor = (Client_Acceptor *) _acceptor;
 8
 9    // for the sake of simplicity, assume this send won't block
10    peer().send(hello, sizeof("Hello\n"));
11    // get ready to read
12    bytesread = 0;
13    if (reactor ()->register_handler (this,
14                                      ACE_Event_Handler::READ_MASK) == -1)
15      ACE_ERROR_RETURN ((LM_ERROR,
16                         "(%P|%t) can't register with reactor\n"),
17                        -1);
18    return 0;
19  }

20  int Client_Handler::handle_input (ACE_HANDLE handle)
21  {
22    in_message[4]='\0';
23    int nreceived = peer ().recv (in_message+bytesread,
24                                  4-bytesread);
25    switch (nreceived)
26      {
27      case -1: // Read error
28      case 0:  // Peer closed it's end
29            this->peer().close();
30            return -1;
31      default:
32        bytesread += nreceived;
```

directed towards [57].

```
33        if(bytesread == 4)
34          {
35            if ((bytes_to_send = atoi(in_message))){
36              bytesread = 0;
37              bytes_sent = 0;
38              handle_output((ACE_HANDLE) this->get_handle());
39            }
40            else  // atoi failed: received strange input from client ?
41              return -1;
42          }
43        return 0;
44      }
45  }
```

select/ace_select/client_handler.cpp

Figure 4.5: ACE Client Handler implementation

Reflections

This little example serves to illustrate how we can use an object-oriented toolkit like ACE to build a portable event driven server. The end result is a clean and extensible design, built with flexible and reusable objects. The underlying patterns in the ACE framework components help to ensure a high degree of robustness, by taking care of much of the error-prone detail. The reactor shields us from the platform specific details of I/O demultiplexing and event dispatching. The reactor can be parameterized to use either `poll()` or `select()` on UNIX. The default on Windows NT is to use `WaitForMultipleObjects()`.

Note from Figure 4.3 how ACE also provides platform neutral wrappers for socket address structures and signal handling functions.

4.2.3 Implementation using Tcl

Some useful tools

The Tcl library is a C library which provides implementations for the Tcl commands used in the interpreter as well as a number of utility functions. In this section I show how we can

make use of some of these functions to build a portable server using C++. The following is a list of some of the Tcl functions we use:

Tcl_OpenTcpServer A useful function which creates a server socket and arranges a callback when clients are accepted. A much simplified version of the acceptors discussed in the previous section.

Tcl_CreateChannelHandler A function to arrange for callbacks to be registered for events on Tcl I/O channels. On UNIX we can use `Tcl_CreateFileHandler`, which works on "normal" file descriptors. Unfortunately, this function is not provided on Windows, probably because of the SOCKET handle problem, so we have to use the more abstract, and more heavyweight, channels if we want to maintain portability. There is no reason why the Tcl core functions cannot be extended to provide a `Tcl_CreateFileHandler` function on Windows (Don Libes did this to facilitate porting Expect to NT), but I leave it for future investigation.

Tcl_DoOneEvent The driver of Tcl event driven programs. Checks the event queue of the Tcl Notifier to see if any registered events are due to be serviced. If so, it arranges for the first event in the queue to be serviced. Otherwise it simply blocks. The Tcl Notifier uses `select()` on UNIX and `WSAAsyncSelect()` on Windows platforms. If Tcl is compiled with threads enabled, the notifier runs in its own thread.

The factory pattern revisited

Figure 4.6 shows a simple main function for our event driven server. Normally, using C, one would create handlers for the connected channel directly in the `acceptHandler` function. Building on my experience with ACE I decided to treat the `acceptHandler` as a factory method for `ServiceHandler` objects. This decision exposed an interesting problem (and fortunately its solution) which I have since come across a number of times when interfacing C++ with C API calls.

select/tclserv/main.cpp

```
12  void acceptHandler(ClientData interp,Tcl_Channel connch,char* hostname,int port)
13  {
14    // Create a new Connection object
```

```
15    ServiceHandler* Sh = new ServiceHandler(connch, (Tcl_Interp*)interp);
16  }

17  int main(int argc, char* argv[]) {
18    // signal(SIGINT,sig_handler);
19    cerr << "starting tcl server on port "<< argv[1]<< endl;
20    int port = atoi(argv[1]);
21
22    Tcl_Interp* interp = Tcl_CreateInterp() ;
23    ServiceHandler::init(5000);

24    // A handy convenience function which creates a listening
25    // socket and causes our acceptHandler to be invoked in
26    // response to new connections
27    Tcl_OpenTcpServer(interp,port,NULL,acceptHandler,NULL);

28    // Start Tcl Event Loop
29    while(1){
30      Tcl_DoOneEvent(0);
31    }
```

select/tclserv/main.cpp

Figure 4.6: Tcl Event Driven Server main()

The problem with static functions

Figure 4.7 shows the interface to my ServiceHandler class. The class encapsulates the connected client channel together with the data members (buffers and pointers) required to manage the I/O.

select/tclserv/tcliohandler.h

```
5  class ServiceHandler {
6   public:
7    ServiceHandler(Tcl_Channel connch, Tcl_Interp* interp);
8    ~ServiceHandler();
9    static void init(int outbuf_size)
10     {
11       out = new char[outbuf_size];
12       // just so we can see what's going on ...
13       memset(out,'A',outbuf_size);
14     }
15
```

```
16    int readRequest();
17    int processRequest();
18    static void ioHandler(ClientData c, int mask);
19   private:
20    // data specific to the connection instance
21    int nbytes;
22    int bytesleft;
23    int bytesread;
24    char *in;
25    static char* out;
26    Tcl_Channel conn;
27    Tcl_Interp *_interp;
28  };
```

select/tclserv/tcliohandler.h

Figure 4.7: Tcl ServiceHandler interface

In Figure 4.8 we see that the constructor makes a call to `Tcl_CreateChannelHandler` to register interest with the Notifier. The problem is that the callback function passed as the third parameter to this function **must** be declared as static. This means that our callback (`ServiceHandler::ioHandler`) does not have access to the data members of our `ServiceHandler` instance. Fortunately the Tcl function allows us to send an application specific parameter as the fourth argument to the function. I make use of this to pass a pointer to the current instance (the *this* pointer) to the static ioHandler.

select/tclserv/tcliohandler.cpp

```
 7  ServiceHandler::ServiceHandler(Tcl_Channel connch, Tcl_Interp* interp){
 8    this->conn = connch;
 9    thie->bytesread = 0;
10    this->in = new char[5];
11    Tcl_SetChannelOption(NULL,connch,"-blocking","0");
12    Tcl_Write(this->conn,const_cast<char*>(hello),sizeof("Hello\n"));
13    Tcl_Flush(this->conn);
14    Tcl_CreateChannelHandler(this->conn,TCL_READABLE,ServiceHandler::ioHandler,this);
15  }
```

select/tclserv/tcliohandler.cpp

Figure 4.8: Tcl Event Driven Server: ServiceHandler constructor

The static `ioHandler()` function is shown in Figure 4.9. It is now possible to refer to specific data or member functions of the ServiceHandler instances by dereferencing the local `ServiceHandler` pointer, `sh`.

select/tclserv/tcliohandler.cpp

```
50 // ioHandler is static
51 // we receive  a reference to a ServiceHandler instance via Clientdata
52 void ServiceHandler::ioHandler(ClientData c, int mask){
53   ServiceHandler* sh = (ServiceHandler*) c;
54   int numread = 0;

55   switch(mask){
56   case TCL_READABLE:
57         sh->readRequest();
58       break;
59   case TCL_WRITABLE:
60         sh->processRequest();
61   }
62 }
```

select/tclserv/tcliohandler.cpp

Figure 4.9: Tcl Event Driven Server: ServiceHandler::ioHandler

Reflections

The problem with static functions outlined above is interesting to reflect upon because it occurs in many places where one attempts to wrap C system API calls in C++ classes[4]. Using these same function calls in a C style program (without classes) results in a simple elegant construction, whereas routing the dispatch back to a C++ object method call is problematic. Schmidt[49] talks of cases of "impedance mismatch" between C and C++ when mixing the two languages. This phenomenon is indeed one of those mismatch cases. I have presented the solution outlined above, using a static *reflection* method, in a pattern format to the PLOP2001 conference in Illinois [16]. The full paper is presented as Appendix E.

[4]For example, we see the same thing with the POSIX `pthread_create()` function which requires the thread entry point to be declared as a static C function.

4.3 Concurrent architectures

Comparing the iterative server of Figure 4.1 with the event driven servers of Figures 4.3 and 4.6, one is struck by the simplicity of the service implementation of the iterative version, using synchronous, blocking I/O. We saw how the introduction of non-blocking I/O solved a problem for us at the cost of some complexity. Recall the reason that we had to use non-blocking I/O was that our server was running as a single thread in a single process. If we could arrange for the service to be implemented in a separate process or thread, we would be able to use the simpler blocking I/O in this service without hampering our server's ability to accept new connections. Doug Schmidt's description of the "Half-Synch Half-Asynch"[50] gives a good background on how and why we want to do this.

The servers presented in this chapter illustrate some of the variations on the multiple process and multiple thread theme.

4.3.1 Classical implementation - multiple processes

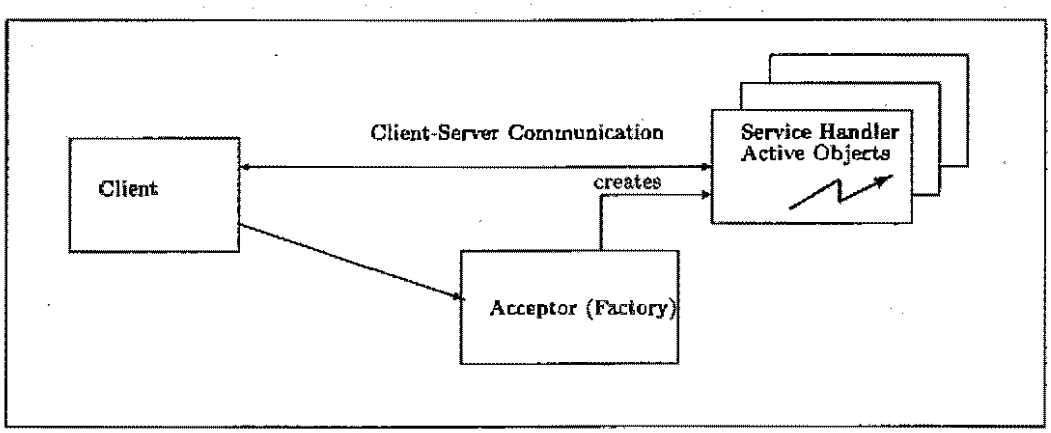


Figure 4.10: Create thread or process per connection

The simplest implementation of the architecture illustrated in Figure 4.10 is using the UNIX `fork()` function. The parent process accepts connections and forks a new process to handle each connection. A basic implementation of our "A" service using the "new process

per connection" architecture is shown in Figure 4.11 below.

concurrent/ace_forker/forker.cc

```
 6  // Reap dead children - avoid "zombies"
 7  void handler(int)
 8  {
 9    while (ACE_OS::waitpid(-1,NULL,WNOHANG) >0);
10  }

11  // main function is similar to iterative server
12  int main(int argc, char* argv[])
13  {
14    if (argc !=2)
15      {
16        ACE_OS::printf("Usage: forker <port>\n");
17        ACE_OS::exit(-1);
18      }
19
20    int Port = ACE_OS::atoi(argv[1]);
21    ACE_INET_Addr addr(Port);
22    ACE_SOCK_Stream peer;
23    ACE_SOCK_Acceptor myAcceptor(addr);

24    char inbuf[5];
25    char outbuf[10000];
26    ACE_OS::memset(outbuf,'A',sizeof(outbuf));
27    inbuf[4]='\0';

28    //install sig handler to catch child exits
29    ACE_Sig_Action sa ((ACE_SignalHandler) handler, SIGCHLD);
30
31    int pid;

32    while(1)
33      {
34        myAcceptor.accept(peer);
35        // fork() on accept
36        if ((pid = ACE_OS::fork() )==0) {
37          // this is the child
38          myAcceptor.close();
39          peer.send_n(hello,sizeof("Hello\n"));
40          switch (peer.recv_n(inbuf,4))
41          {
42          case 4:
43            peer.send_n(outbuf,ACE_OS::atoi(inbuf));
44            peer.recv(inbuf,1);
45            break;
46          case 0:
47            ACE_OS::fprintf(stderr,"Odd: client closed\n");
```

```
48            break;
49          default:
50            ACE_OS::perror("Recv");
51            break;
52          }
53
54          peer.close();
55          // child exit
56          ACE_OS::exit(0);
57        }
58        // this is the parent
59        peer.close();
60      }
61  }
```

concurrent/ace_forker/forker.cc

Figure 4.11: New process per connection

The `fork()` function is the standard way to create processes on UNIX systems. It is a unique function call in that it returns two values. The fork causes a copy of the running process to be made. The only way of distinguishing between the parent and the child is the return value of `fork()`. A return value of 0 indicates we are the child. The return value in the parent is the process id of the child process[5].

There is no native equivalent to `fork()` in the Win32 API systems.

Process creation on Windows NT

The standard way to create processes on Windows NT is using the `CreateProcess()` or `CreateProcessEx()` function calls. This new process is not cloned from a running process. Each new process is independently loaded and started up from an executable file on the filesystem. This startup time is a strong argument against this new-process-per-connection model being used in a Windows environment.

Cygwin and UWin (see Section 2.2) do provide an emulation of fork() for Windows. The Cygwin version is open source so we can see exactly how it has been achieved. The

[5]It is often important for the parent to keep track of child processes. In our example, the only important thing is for the parent to catch and clean up after the child exits. This is the reason for the signal handler. See [55] for a thorough explanation of cleaning up "zombies".

new process is started in the standard Windows way by loading a new invocation of the running program (as reported by `argv[0]`). This new process is created in a suspended state (a Windows feature for which I know no UNIX equivalent), and its data segment is overwritten with a copy of the parent's data. Once the new process has been suitably hacked to resemble the execution state of the parent (with the exception of the return value of `fork()`) its execution is resumed. The result is a crude and inefficient, but functional, emulation of forking [31].

The `fork()` emulation is useful because there are so many UNIX servers which depend on this mechanism. Almost all of the "classic" servers such as the apache web server, the innd news server, sendmail, inetd, telnetd and a host of others are examples of forking daemons. Porting them to Windows without a `fork()` facility involves a substantial rewrite of the concurrency mechanism. In a small scale scenario, where the expense of process creation is not critical, a literal port of these types of server using the likes of Cygwin is a worthwhile and relatively painless exercise. Where a better performance characteristic is required, some other concurrency strategy needs to be used. The "official" Windows port of the apache web server from the Apache project (http://www.apache.org/) makes use of multiple threads instead of processes, though both Cygwin and UWin ports are maintained based on the current UNIX source code.

Process pools

Even on UNIX, where the cost of process creation is more reasonable, forking a new process per connection is not the most efficient strategy. The strategy persists because it has two great strengths:

1. Simplicity. There are very few complications to consider. The process uses blocking I/O so there is no need for the complex buffer management typical of event driven servers. The flow of control is easy to follow and debug.

2. Robustness. A server based on short-lived processes has an inherent stability. Once the service process has performed its task, all resources allocated to it are reclaimed by the operating system on exit. Using the operating system to do the garbage collection reduces the complexity of the program considerablity.

A compromise solution is to pre-fork a pool of processes. This removes the overhead of process creation on a per-connection basis, but introduces the new problem of how to allocate incoming client connections to service processes. I have not shown a code example here since the basic strategies are similar to that of the multi-threaded servers below.

Pre-forking negates some of the robustness qualities of the short-lived new-process-per-connection described above. If a process is going to stay around for longer, it needs to take more responsibility for the management of its resources. Again there are compromises one can reach. A managed process pool can create and cull processes dynamically to adapt to server load as well as limiting the lifespan of individual processes to only handling a fixed number of requests before being reclaimed by the operating system. This is a model used by apache, which though far from being the fastest of web servers, certainly has a reputation for stability.

4.3.2 Lightweight processes - threads

Threads can be user level or kernel level. This discussion only applies to kernel level threads. Threads offer a lighter weight alternative to processes for implementing designs based on active objects. The creation time is considerably less than for a process, as is the overhead of context switching. The API's for creating threads on Windows and UNIX are different, but sufficiently close to make porting an easier proposition than is the case with multiple processes. There are however, many more hazards to be aware of:

1. Threads do not release resources automatically on exit the way processes do.

2. Functions called by multi-threaded programs must be thread-safe i.e. they have to deal with re-entrancy. This property must also be true of any libraries the program is linked against. This can still be a problem with many legacy libraries.

3. Great care must be taken to synchronize access to shared resources. Different mechanisms are used on different platforms to achieve this. The primary mechanism on Windows is the CRITICAL SECTION[44]. This is similar to the Java 'synchronized' sections. Condition variables, which are a widely used technique on POSIX systems, are not available on Windows.

4. The behaviour of signals with threads is platform dependent. Many experts (see usenet news://comp.programming.threads) advise against any mixing of signals with threads. The interactions are potentially complex and best avoided.

Nevertheless there are benefits. With threads one gets the convenience of non-blocking I/O without the excessive overhead of process creation and context switching. This is still less efficient on single processor machines than event driven designs which have no context switching overhead at all. The choice between the two models in this case is difficult. Factors one might consider are:

- What is the nature of the service? Event driven servers tend to favour short-live services [3][14]. Connections that are likely to stay around for longer would benefit from a thread, or even a process, being allocated to them.
- What are the other functional requirements of the system? A TCP/IP client, for example, may require a GUI which uses an event loop (such as that provided with Tcl/Tk). In this case it may be simply convenient to integrate non-blocking I/O into the existing event loop. This is also typical of many Windows servers which throw up a GUI control panel of some sort.

The most common API for threads is the POSIX 1003.g specification (commonly known as pthreads). Although many UNIX flavours carry their own thread variations, almost all of them support the POSIX pthreads API. Windows threads do not, but there are a number of freely available pthreads libraries which essentially wrap native Windows threads functions in standard pthreads calls. I have used the Win32 Pthreads library from Cygnus Solutions with some success.

Again there are benefits in abstracting away from the low-level API and using an Object-Oriented wrapper library when programming threads across platforms using C++. The illustrative examples which follow use the ACE library. I have taken this approach because the code is shorter, it's easier to write and it runs on UNIX and Windows platforms.

Thread per connection

The thread per connection model is similar to the process per connection model discussed in Section 4.3.1 above. Instead of forking a new process to handle the incoming client

```
... initialize listening socket ...
...

  while(1) {
        connfd = accept(listenfd,(struct sockaddr*) NULL, NULL);
        if (pthread_create(&tid,NULL,&worker,(void *) connfd)≠0){
           cout ≪ "Trouble!  can't create thread " ≪ endl;
           close(connfd);
        }
  }


static void* worker(void* arg) {
   ... perform service
}
```

Figure 4.12: Thread per connection

connection, we create a new thread. A section of a pthreads program implementing this strategy is shown in Figure 4.12. A simple wrapper around the Windows thread functions is sufficient to get this type of program to compile on Windows.

Note that the thread main function is a static C function. This gives rise to the same problem discussed in the previous section when trying to encapsulate a thread into a C++ class.

ThreadPool version 1

As with multiple process servers, one can avoid the cost of creating a new thread for each connection by pre-spawning a pool of threads. The most common variation on this theme is to have one thread responsible for accepting connections and dispatching them to worker threads who perform the service. Figure 4.13 illustrates the idea.

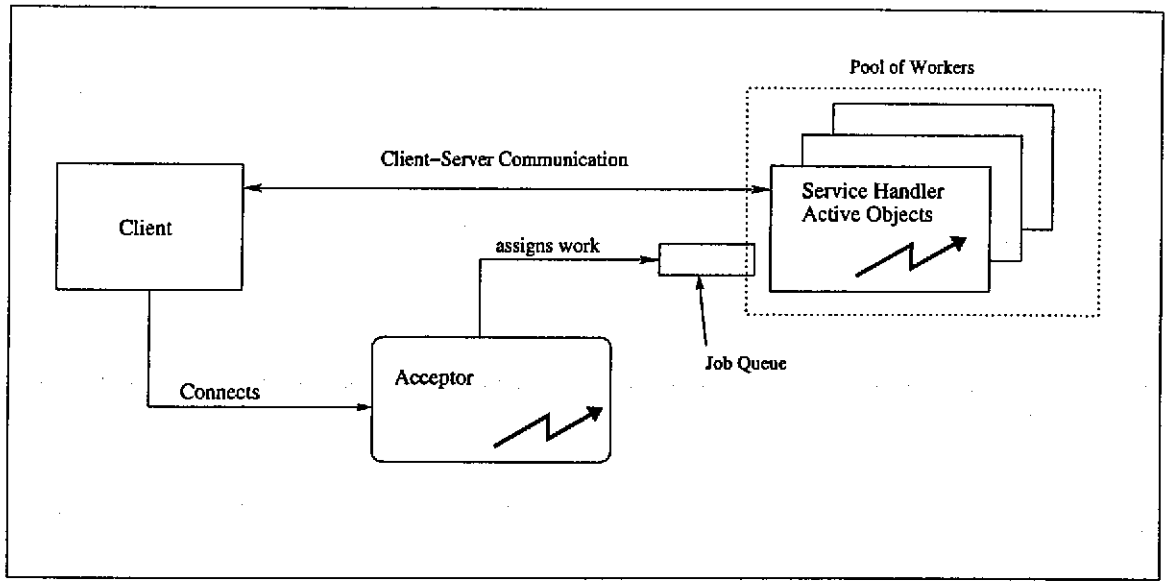


Figure 4.13: Pre-threaded: Acceptor enqueues, workers dequeue

concurrent/ace_msgq/simple.cpp

```
 6 int main(int argc, char* argv[])
 7 {
 8         if (argc != 3){
 9                 cerr << "usage server <port> <nworkers>" << endl;
10                 ACE_OS::exit(-1);
11         }
12
13         int Port = ACE_OS::atoi(argv[1]);
14
15         ACE_INET_Addr addr(Port); //listening socket
16         ACE_SOCK_Stream *peer;    //client connection
17         ACE_Message_Block *mb;    //for passing to our threads
18
19         ACE_SOCK_Acceptor myAcceptor(addr);
20
21         // create worker threads
22         Worker ThrPool(ACE_OS::atoi(argv[2]));
23         //start them up ...
24         ThrPool.open();
25         while(1)
26         {
27                 peer = new ACE_SOCK_Stream;
28                 myAcceptor.accept(*peer);
29                 //make a new message block containing reference to peer socket
30                 mb = new ACE_Message_Block ((const char*)peer,sizeof(peer));
```

```
31                  //put it on the thrpool queue
32                  ThrPool.putq(mb);
33          }
34          return 0;
35  }
```

concurrent/ace_msgq/simple.cpp

Figure 4.14: ThreadPool ver 1 main()

Figure 4.14 shows the main() function of a thread pool based server. Connections are accepted and a reference to the connected socket is placed onto a message queue.

Figure 4.15 shows the interface to the worker class which is responsible for de-queueing messages and servicing the connection. I have made use of an ACE_Task class to do this, as each task has an associated message queue. The constructor is used to set the number of threads assigned for this task. Hence the Worker class actually represents a pool of threads. The ACE_Task object takes care of dispatching messages from a single message queue to the waiting threads.

concurrent/ace_msgq/Worker.h

```
 9  // Worker descends from an ACE_Task
10  // this is how we get our message queue for free !
11  class Worker : public ACE_Task<ACE_MT_SYNCH>
12  {
13  public:
14    Worker (size_t n_threads) : n_threads_(n_threads){;}
15    int open (void * = 0); // start up our threads
16
17    /* Our worker method */
18    int svc (void);
19
20  protected:
21     size_t n_threads_;  // Number of threads in the pool.
22     char outbuf[10000]; //shared output buffer
23  };
```

concurrent/ace_msgq/Worker.h

Figure 4.15: ThreadPool ver1 Worker interface

Figure 4.16 shows the implementation of the Worker class. Here we see that the open() method is being used to activate the threads. The activated threads start in the svc() method, where they block waiting for messages. On de-queueing the message the thread goes on to service the request using blocking I/O. Notice that we do not delete the message object. The messages are reference counted. The release method will cause the message to be deleted when there are no further references to it.

concurrent/ace_msgq/Worker.cpp

```
 3  /* Open the object to do work.  Next, we activate the Task into the
 4    number of requested threads.  */
 5  int
 6  Worker::open (void *unused)
 7  {
 8    return this->activate (THR_NEW_LWP,
 9                           n_threads_);
10  }
11
12  /* Our svc() method waits for work on the queue and then processes
13    that work.  */
14  int
15  Worker::svc (void)
16  {
17    ACE_Message_Block *message;
18
19    const char* hello="Hello\n";
20    char inbuf[5];
21    char outbuf[10000];
22    ACE_OS::memset(outbuf,'A',sizeof(outbuf));
23
24    ACE_SOCK_Stream *peer;
25
26    for (;;)
27      {
28        /* Get a message from the queue. Blocking  */
29                  if(getq (message)==-1){
30                    cerr << "Oops" << endl;
31                    message->release();
32                    continue;
33                  }
34
35                  //message should contain our SOCK_STREAM reference ..
36                  peer = (ACE_SOCK_Stream *)message->base();
37
38                  // Do the job.
39                  peer->send_n(hello,sizeof("Hello\n"));
40                  while (peer->recv_n(inbuf,4) >0){
```

```
41                          int nbytes = ACE_OS::atoi(inbuf);
42                          peer->send_n(outbuf,nbytes);
43                  }
44                  peer->close();
45                  // Clean up ...
46                  delete peer;
47                  message->release();
48      }
49
50    return 0;
51  }
```

concurrent/ace_msgq/Worker.cpp

Figure 4.16: ThreadPool ver1 Worker implementation

There are other methods for communicating between threads, but the message queue model works well and is easily transferable to a multi-process architecture as well.

ThreadPool version 2

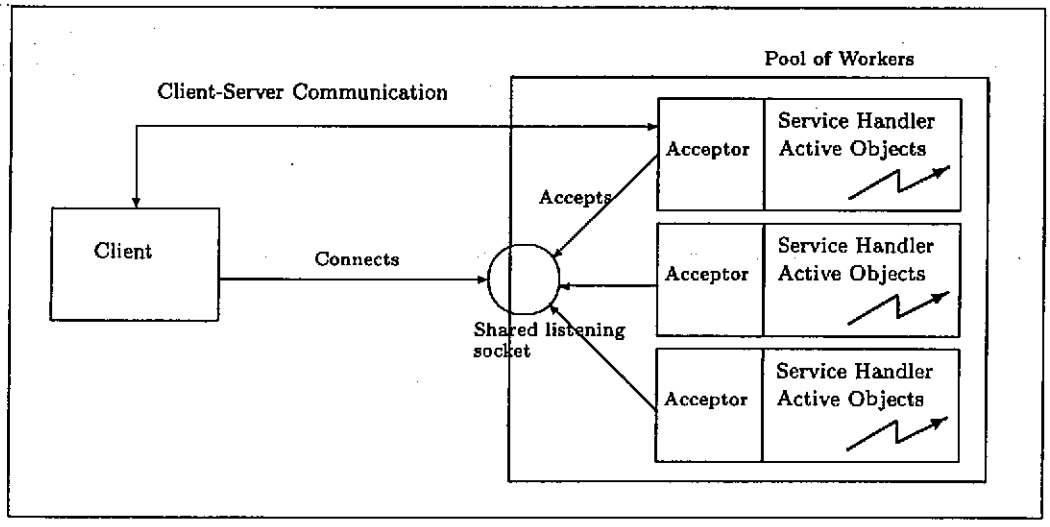


Figure 4.17: Pre-threaded: Workers compete for jobs

Stevens [55] has a variation on the thread pool idea which I have reinterpreted here using ACE. Our listening socket is already maintaining a queue of client connections (see Section 3.2.4). It may seem wasteful to have an acceptor thread de-queueing connections off the listening socket and then en-queueing them again to pass on to worker threads. Figure 4.17 shows the variation where the workers themselves each call `accept()`.

The `main()` function in Figure 4.18 below does little more than to create the worker threads. In a real application the originating thread may do more in terms of managing the thread pool. In this simple example it merely creates them and goes to sleep. The interface for the Worker class is not substantially different to the previous version. We do need to pass an extra parameter to the `open()` method to indicate to the worker threads which port to listen on.

concurrent/ace_thrpool_accept/simple.cpp

```
 6  int main(int argc, char* argv[])
 7  {
 8          if (argc != 3){
 9                  cerr << "usage server <port> <nworkers>" << endl;
10                  ACE_OS::exit(-1);
11          }
12
13          u_short port = ACE_OS::atoi(argv[1]);
14          u_long nthreads = ACE_OS::atoi(argv[2]);
15          // create worker threads
16          Worker ThrPool(nthreads);
17          //start them up ... listening on port
18          ThrPool.open(port);
19          ThrPool.wait(); //block until all threads exit
20          return 0;
21  }
```

concurrent/ace_thrpool_accept/simple.cpp

Figure 4.18: ThreadPool ver2 main()

Figure 4.19 shows the `svc()` method of the worker threads blocking in a call to `accept()`. Only one thread will successfully dequeue the connection. There is a danger of a problem

known as a "thundering herd" [55], whereby all the threads are woken on the arrival of a connection. One of them succeeds in accepting the connection and the rest go back to sleep again. This activity causes a lot of unnecessary system overhead and can be prevented by providing some sort of locking synchronization around the call to accept().

concurrent/ace_thrpool_accept/Worker.cpp

```
 3 /* Create listening socket and activate workers */
 4 int Worker::open (u_short port)
 5 {
 6   ACE_DEBUG((LM_DEBUG,"Opened listener\n"));
 7   listener.open( ACE_INET_Addr(port) );
 8   ACE_OS::memset(outbuf,'A',sizeof(outbuf));
 9   return this->activate (THR_NEW_LWP,n_threads_);
10 }
11
12 /* Each thread competes to accept connections */
13 int Worker::svc (void)
14 {
15   const char* hello="Hello\n";
16   char inbuf[5];
17   ACE_SOCK_Stream peer;
18
19   for (;;)
20     {
21       /* all threads block here: synchronize access to accept?*/
22       _lock.acquire();
23       ACE_DEBUG((LM_DEBUG,"%t: Got the lock\n"));
24       listener.accept(peer);
25       _lock.release();
26       // Do the job ...
27       peer.send(hello,sizeof("Hello\n"));
28       while (peer.recv_n(inbuf,4) >0){
29          int nbytes = ACE_OS::atoi(inbuf);
30          peer.send_n(outbuf,nbytes);
31       }
32       peer.close(); // Clean up ...
33     }
34   return 0;
35 }
```

concurrent/ace_thrpool_accept/Worker.cpp

Figure 4.19: ThreadPool ver2 Worker Implementation

The _lock used in Figure 4.19 is a parameterized type which can be set to use mutex locking, file locking, semaphore locking or CRITICAL_SECTION locking (on Windows NT). The mutex semantics should ensure that only one of the threads waiting to acquire the mutex will actually be woken. This version of our thread pool server is potentially more efficient than the previous version (doing away with the need for message queues).

Java threads

We could not use Java to implement an event driven server because of the lack of language support. Java does, however, provide integrated language support for threads. I have implemented and tested both the thread-per-connection and thread-pool architectures in Java. A Java version of the multi-threaded architecture of Figure 4.17 is shown below. It uses the same strategy of each thread blocking in accept().

java/thread_pool/ServerJMTpool.java

```
 4  public class ServerJMTpool extends Thread
 5  {
 6          public static void main(String[] args) throws IOException
 7          {
 8                  ServerSocket s = new ServerSocket(60002,30);
 9                  for(int n=0; n< 10; n++) {
10                      PoolConnectHandler t = new PoolConnectHandler(s);
11                      t.start();
12                  }
13                  // currentThread().suspend();
14                  System.out.println("Running ...");
15          }
16  }
```

java/thread_pool/ServerJMTpool.java

Figure 4.20: Java ThreadPool main thread

The code for the startup class is shown in Figure 4.20. As before, the main thread simply creates a ServerSocket and then creates the worker threads.

The ConnectHandler thread is shown in Figure 4.21. It implements the same service as our C++ versions.

java/thread_pool/PoolConnectHandler.java

```
27      // the thread's entry point ... very simple
28      public void run()
29      {
30          try {
31              // just accept connections and service them
32              while(true){
33                  connfd = listenfd.accept();
34                  service();
35              } // loop forever
36          } catch (IOException e) {e.printStackTrace();}
37      }
38
39      // reads the 4 bytes from client and sends back reply
40      private void service()
41      {  ...
```

java/thread_pool/PoolConnectHandler.java

Figure 4.21: Java ThreadPool: ConnectionHandler

This Java version runs in the Java VM on Windows and Linux. The performance measures recorded in Chapter 5 show it to be somewhat heavier than the native binary equivalents, but probably adequate for a wide range of applications. I have also successfully compiled it with gcj (see Section 2.5) on Linux to run as a native binary. Preliminary results indicate that it is at least twice as efficient as the same server running with JIT enabled in the Java VM. Support for the libgcj Java run time library on Windows is still at an alpha stage hence I have been unable to present comparative results here.

4.4 Summary

In this chapter I have demonstrated the portability of the classical event driven and concurrent server architectures. Both Windows and Linux (and other UNIX flavours) provide the essential OS mechanisms to implement these architectures. Using intermediate library code it is possible to target source code at abstractions of these mechanisms and thereby maintain a common source code base between platforms.

Whereas there are clear benefits to this approach, there must surely be a penalty in terms of performance. In the following chapter I have quantified that penalty within a particular application context.

Chapter 5

Performance comparison

5.1 Introduction

I have illustrated in previous chapters that both Windows NT and Linux provide native support for implementing single-threaded and multiple threaded TCP/IP servers. The system API's for thread creation and synchronization are different, but not markedly so. The semantics of event demutiplexing and dispatching differ more significantly. I have shown, however, that it is possible to implement the fundamental design patterns using higher level API's (ACE, Tcl, cygwin, uwin etc) which essentially mask the differences between the target system interfaces. The forces in favour of using such API's are compelling:

1. the portability problems are contained within a single software layer
2. the implementation of this layer is likely to be robust, particularly if the source is freely available, mature and widely used
3. there is frequently extra added value to be gained by using the higher level API, such as a reusable framework of objects (ACE) or the availability of an embedded interpreter and related library functions (Tcl).

There is of course a price to be paid. Forces which act against the use of a higher level API include:

1. There may be a significant effort required on the part of the developer, or development team, to learn yet another API. In the often subtle, and always complex, domain of

communication software this developer effort may indeed be a high price to pay. David Korn [18] cites this as a major factor which influenced his decision to create a POSIX layer on NT, rather than to adopt or create a different higher level API.

2. There will be some runtime cost in terms of efficiency as a result of the extra layer of abstraction.

A successfully engineered and portable software solution to a particular server design problem will have to resolve both of these forces. I have not attempted to measure the cost of learning communication API's, but merely note, from my own and other experience, that there is a cost. In this chapter I present an approach to measuring runtime efficiency. I use this approach to evaluate the porting strategies introduced in earlier chapters. Though it would be proper to look at runtime efficiency in terms of time and space i.e., CPU and memory utilization, the scope of this study is restricted to CPU utilization.

5.2 Hypothesis revisited

The inclusion of an extra layer of abstraction implies that more instructions will need to be executed to do an equivalent amount of work. Therefore by accessing the underlying system calls via a higher level API, we can reasonably expect that we will be consuming more CPU cycles than if we access the system calls directly.

The hypothesis I proposed in Chapter 1 was that the run time cost of using a higher level API is not neccesarily prohibitively high. In this chapter I will show that this is substantially true.

5.3 Supporting argument

How do we interpret a performance speed up or a degradation? Jeffrey Mogul [29] makes a scathing critique on the "brittle" nature of many of the metrics quoted in operating systems research. In particular, when claims are made about performance, it is to be expected that the measurements supporting those claims should be repeatable, comparable to existing measures and that they should have some relevance to existing applications.

I am confident that the measurements I have made here are repeatable and the full sources of all my sample servers and test environment is made available as an appendix, precisely to facilitate this. I have based my approach on that used by the late Richard Stevens in Unix Network Programming Volume 1 [55]. Whereas the approach is known to have its limitations, many of which I address in the following sections, the Stevens book (commonly referred to simply as UNPV1) is widely known and referred to. By basing my approach on Stevens I provide some reference for comparison. In the text that follows I refer to his method as the UNPV1 method.

There remains a clear danger when using synthetic applications (as I have done) that we end up none the wiser as to what the observed effects would be on a real application. Essentially the server programs which I compare, like the ones of UNPV1, illustrate skeletal architectures rather than functional, real applications. In Section 5.4 below, I present a case that differences observed in these skeletal architectures are in fact exaggerated differences. As we add more layers of application functionality, the degree of degradation seen as a result of selecting a portable implementation is expected to become proportionally less.

5.4 Amdahl's Law in reverse

Patterson and Hennessy [15] show that the performance gain that can be obtained by improving some feature of a computer can be calculated using Amdahls's Law. Amdahls's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

For computation bound tasks, the following relationship expresses this idea:

$$Speedup = \frac{T}{T_e} \tag{5.1}$$

Where T is the execution for the entire task without using the enhancement and T_e is the execution for the entire task using the enhancement when possible.

Speedup tells us how much faster a task will run using the machine with the enhancement as opposed to the original machine.

Because the enhancement is only usable for a fraction of the time, we can derive the following relationship to express the ratio of the execution times and hence the overall

speedup:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \quad (5.2)$$

This gives us a useful quick way of calculating the speedup based on two factors: the fraction of time an enhancement is used and the speedup achieved while using the enhancement.

Consider for example a server that does network I/O for 10% of the time - perhaps the remaining 90% is taken up doing disk I/O and computation. We can calculate the speedup we would get by doubling the performance of the network I/O:

$$Fraction_{enhanced} = 0.1$$

$$Speedup_{enhanced} = 2$$

Substituting into equation 5.2 yields:

$$Speedup_{overall} = \frac{1}{(1 - 0.1) + \frac{0.1}{2}} = 1.05$$

We can see that there is a diminishing return here. Even though we have doubled the performance of the network I/O we only see a performance improvement of 5%.

Amdahl's Law has two important implications on the interpretation of the results I have obtained in measuring the cost of portability:

1. The nature of my test servers is essentially artificial. They do no additional work other than the moving of data from the server to the client(s) and establishing the minimum infrastructure in terms of threads, processes and event handlers to accomplish this task. The impact of any slowdown (or speedup) that we can measure on these skeletons will in fact be far less on a "real" server which is also doing other work.

2. Whereas we have seen that the overall performance improvement gained by improving a feature is limited by the amount of time that feature is used, we can also reasonably expect that the performance degradation experienced by incorporating a slower feature is also similarly limited.

By masking operating system calls in portable wrapper functions we are not enhancing our servers so that they will exhibit an increased speedup. In fact we expect the reverse to

be true. There should be some overhead involved in calling the wrapped function, which will result in a decreased speedup, or penalty. Either way we expect the law of diminishing returns to be true. If a program spends 10% of its time executing the de-enhanced (or wrapped) sections of a program, and those sections are twice as slow as the native (unwrapped) sections, we can still apply Amdahl's Law and equation 5.2:

$$Speedup_{overall} = \frac{1}{0.9 + \frac{0.1}{0.5}} = 0.909$$

By incorporating components which are twice as slow, but only using them for 10% of the time we expect to see a speedup of around 0.9. In this case we interpret the fractional speedup as a penalty. Our new server should only experience a performance degradation, or penalty, of around 10% as a result of using the slower functions.

In reality the cost of wrapping a native call in a function wrapper should be very low. Measurements taken using ACE (see Appendix C) suggest that the absolute cost of the extra function call on my test system should be somewhere between 0.005μs and 0.009μs. Where these function calls have been inlined, as is the case in many of the ACE wrapper classes, their is no reason to expect any degradation at all.

5.5 Experimental method

In his book, Unix Network Programming Vol 1[55], Stevens compares the efficiency of server designs by examining CPU usage. While conceding that this is not the most sophisticated way of measuring performance, he contends that it provides us with a reasonable basis for comparison. The primary focus of my investigation is comparison, so I have adopted a similar approach. While measuring essentially the same quantities as Stevens does, I have improved on the system of measurement as well as the accuracy of the measures. There is an important difference in intent - whereas Stevens essentially constructs a "shootout" between architectures (single threaded, multi-threaded, multi-process etc), my intention is to compare different implementations of the same architectures.

5.5.1 The UNPV1 method

The UNPV1 method measures different architectural implementations of a simple http-like service. The first stage in the process is as follows:

- Run the simplest possible iterative server - this server is expected to consume the least amount of CPU cycles per request as there is a minimum of control overhead.
- Generate a fixed amount of work for it to do eg. 20000 consecutive requests for 4000 bytes of data.
- Record the amount of CPU time (system time and user time) used by the process to perform the task. The UNPV1 servers have a SIGINT handler installed - the handler uses the `getrusage()` system call to report the CPU utilization times of the process in response to the SIGINT signal.

The figure we get from this activity represents a baseline figure. Our simple server has done the minimum amount of work required to service the requests. There is no process control overhead such as process forking, thread creation, context switching or event handling. There is also no overhead from indirect function calls resulting from using a "portability" layer[1]. We expect that servers which have any of these features will require more CPU time to perform the same fixed amount of work. It is this extra CPU time which is of interest as it is a measure of the *cost* of the particular feature or technique[2].

The approach is basic and has some limitations. The analogy here is of measuring total work done, rather than power being used at any given instant. Stevens is not "stress" testing the server by systematically increasing the connection *rate* until the server fails [14]. This type of measure is interesting, but also closely related to factors external to the software itself such as the underlying network, the processor speed, the speed and amount of system memory etc. By simply providing a "normal" load condition[3] and looking at the CPU usage

[1] In fact Stevens does make extensive use of wrapper functions in `libunp` to incorporate testing of return values and reporting errors.

[2] A further benefit of Stevens approach is that the measure is not unduly influenced by the scheduling priority used. Like his experiments, all of my test servers were run using the default scheduling priority and nevertheless produced highly consistent results.

[3] Effectively the connection rate is being clocked by the server, making it impossible to raise the load above what the server is prepared to handle.

cost, we are reasonably assured that variations observed are attributable to variations in the software design of the server.

The nature of the "normal" load is an issue. I have chosen to standardize on a request size of 4000 bytes, which is similar to the original UNPV1 experiments. The client sends 4 bytes ("4000") and the server replies by sending 4000 bytes ("AAAAA...A"). This is a similar pattern to what you might see with a typical http request. There are of course many other types of service. An ftp server might typically transfer many megabytes of data in a single session. Design choices which are good for one type of service do not necessarily translate to another. For example, process and thread creation times are significant issues in these http-like servers, which typically provide a short lived service. These factors become less significant for longer lived services.

Thus, while the information I have gathered using this technique gives us interesting and worthwhile information about the server characteristics, it does not tell us about performance of all types of servers under all circumstances. We will see, however, in Appendix A that the architecture of my test environment is such that it is easily extendable to incorporate different types of test loads. The measurement is essentially non-intrusive, without requiring access to the source code of the server under test.

5.5.2 Some problems with the UNPV1 method

My early attempts to replicate the UNPV1 results exposed some difficulties which I highlight below:

The servers require instrumentation

That the servers require instrumentation i.e they are self measuring, containing the code to trap SIGINT, call `getrusage()` etc, is not a problem in the UNPV1 context. I particularly wanted to avoid building in instrumentation so as to generalize the system of measurement. It should be possible to measure the utilization of any process without that process necessarily co-operating with the measurement.

CPU utilization of threads were not properly reported on Linux

The problem of acquiring utilization statistics for threads on Linux relates to the Linux interpretation of a thread. On Linux a thread is viewed exactly as a process from the scheduler's perspective. Hence it runs with its own pid (strictly a POSIX violation). It is thus incorrect to assume (as is done in UNPV1) that the CPU utilization reported for a process reflects the utilization of all the threads running within that process. This is a Linux peculiarity which is easy to compensate for. We simply take the same view as the scheduler, and treat all running tasks (be they threads or processes) as independently scheduled units which need to be accounted for individually. Unfortunately, for reasons which are addressed in the following paragraph, there is so much CPU time which is being used outside of the context of the process that this *fix* eventually proved unnecessary.

The reported statistics per process are unreliable

The reliability of the reported statistics was a particular problem. In particular, on Windows, there is a marked and variable difference between the CPU utilization for the process, and the utilization of the system as a whole. Given the nature of the operating system, this is to be expected. Much of the work that is being done on behalf of the process is being done by the I/O subsystem and others. These do not accumulate their time back to the calling process. The interix servers (see Table 5.3 in Section 5.7.1) are an extreme case. Because Interix is implemented as a separate environment subsystem, apparently no CPU time at all is accounted to the calling process. Whereas it might have been possible to trace down all the times being accumulated by the various tasks, it is certainly simpler to view the overall system utilization instead of the per process utilization. According to my observations this figure gives a fairer reflection of all the work being done on behalf of the process. The slight error of overestimation as a result of the occasional disk cache flush is considerably less than the gross error of underestimation in the per-process statistics.

On Linux kernel version 2.2 there is a very close agreement between the usage reported per process and the overall usage, with the latter being equal to and occasionally very slightly greater than the former. Unfortunately the same does not hold true of early version 2.4 series kernels (my test kernel was 2.4.3). Like Windows, there is a large discrepancy. Changes to the kernel to better support SMP (symmetric multiprocessing) have lead to changes in the

scheduler code which appears to be the culprit here. The effect is visible, even with a non-SMP configured kernel. As with Windows, the fairest reflection of utilization can be gained by reducing additional system activity to a minimum and recording overall system usage. The UNPV1 method under-reports on this system - the most convincing evidence for which can be seen by comparing the simple Linux iterative server to the fork per connection server. Clearly the latter should be consuming more cycles in kernel mode, but the process utilization recorded with `getrusage()` and the `proc` filesystem shows less[4]. The overall system CPU usage provides a more correct picture of actual work being done here.

There is a race condition which favours slow servers

The Linux kernel version 2.4.3 is also afflicted with long latencies and surprizingly (shockingly) expensive context switches. This phenomenon is well known and heatedly debated on the Linux kernel mailing lists. A thorough investigation into this behaviour must unfortunately be left for future work. My understanding is that these delays are a consequence of the kernel holding long duration spinlocks, during which time it cannot be preempted. If the thread which owns the spinlock is pre-empted prior to releasing the lock, all other threads on all CPUs will be deadlocked [2]. Consequently, interrupts must be disabled while holding the lock, which has an adverse effect on latency. The scheduler code is protected by such a spinlock - I/O bound processes such as the servers under test here should rarely use their full quantum and therefore the scheduler code will be revisited frequently . Reports on the Linux kernel mailing lists suggest that this can lead to as much as 30% of all time being spent in the scheduler function itself. Though this extreme situation should only be found with a busy run queue and an SMP configured kernel, the discrepancies I have recorded between per process utilization and the overall utilization points a strong finger in this direction. There is currently considerable work being done on the scheduler algorithm as well as preemptable kernel (for low-latency) patches, so it is probably safe to assume that these effects are transient.

A "positive" spinoff of these unfortunate effects is that it exposes a race condition in the

[4]Note that this server *does* wait for all exited children so we are supposedly accumulating child utilization as well.

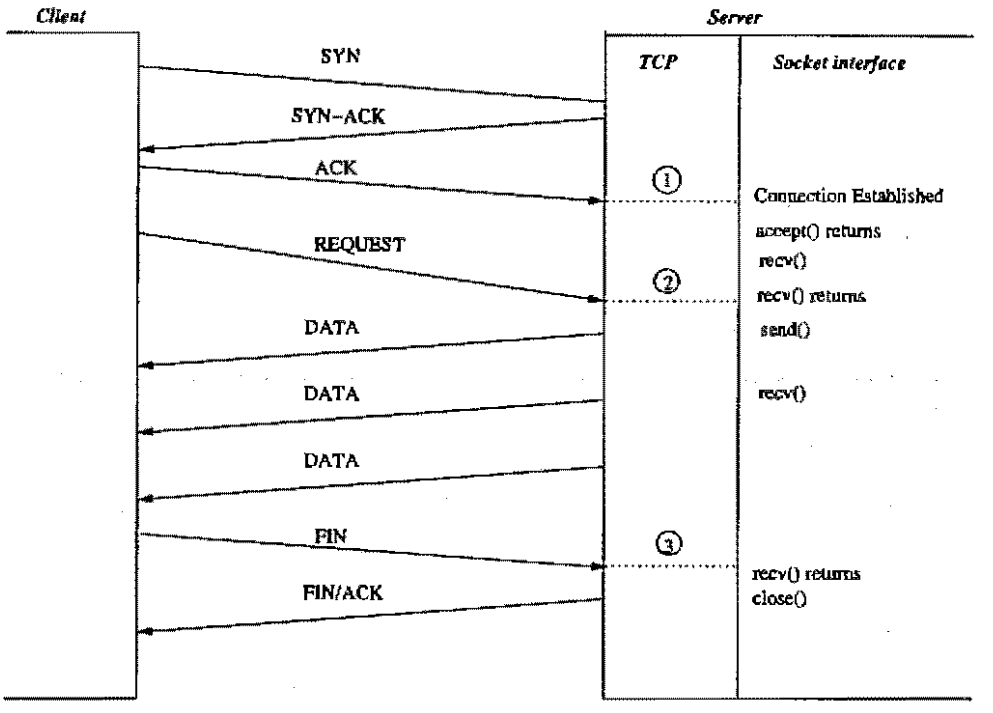


Figure 5.1: Timing of TCP events using UNPV1 protocol

original UNPV1 test protocol. Consider the timing diagram of figure 5.1[5]. There are three events indicated in the server TCP which might cause a blocked server process to unblock.

1. A connection is fully established on a listening socket. A blocked `accept()` call would unblock at this point.

2. Data is available in the connected socket's receive buffer. A blocked `recv()` would unblock at this point.

3. The client has closed. Again a blocked `recv()` will return.

The UNPV1 client connects to the server, then immediately sends off its request. Considering that a TCP connection can be fully ESTABLISHED [37] but still not `accept()`'ed this leads to two possible scenarios:

[5]All the ACKs are not shown - only those segment arrivals which cause the server TCP to propagate an event to the server process are relevant to this discussion.

1. The server accepts the incoming connection quickly, then blocks in a `recv()` waiting for the incoming request.

2. The server is slow to accept the connection. By the time it does, the request is already in its socket receive buffer, and it does the `recv()` without blocking[6].

The first case is probably the norm, but the second case can and does happen. In particular, the Java version of the basic iterative server consistently does this. It takes longer to `accept()`, both because it is interpreted and slower, but also because the Java ServerSocket always copies the address of the client connection up into user space. Using the native `accept()` one has a choice to make the second parameter NULL and avoid the copy. The net result of being slower is that it comes out of `accept()` to find the client's request has already arrived so it can immediately recv() without blocking.

Is the race significant? Unfortunately, on Linux 2.4.3 it is *very* significant because of the huge cost involved in being rescheduled. The java server, which should on all counts perform slower than a C/C++ equivalent, performs *considerably* better! It uses less CPU time (15-20%) and achieves a higher throughput than the native coded version. The reason it performs better is precisely because it is slower. By running slower it avoids having to block in `recv()` and benefits handsomely as a result.

5.5.3 Modifications to the UNPV1 method

Given the discussion above, there is clearly a need to modify the approach in a number of ways. I have taken the following steps:

- To avoid having to instrument the servers, I use a `harness` process to launch the server under test. Armed with the pid (or Windows handle) of the server process, it is then relatively easy to read the statistics from the `proc` filesystem on Linux or the Windows performance counters on NT. I use the `proc` filesystem under Linux rather than `getrusage()`, because `getrusage()` only records child usage after the child has

[6]This is very similar to X.25 call user data. The Winsock AcceptEx() function provides a useful feature to take advantage of this condition and avoid the race. AcceptEx() can be configured to accept() and read() a specified number of bytes in a single operation, either blocking or overlapped.

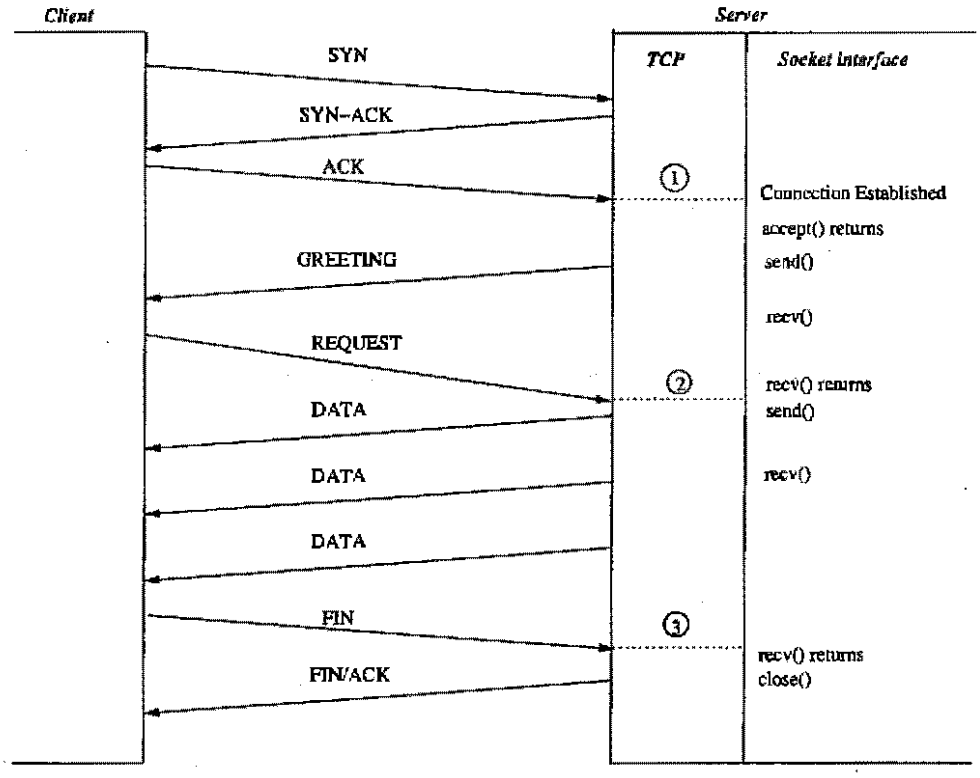


Figure 5.2: Modification of UNPV1 protocol

exited (and been waited upon). It is sometimes useful to be able to measure something without having to kill it first - this is perhaps a similar difference as that between an X-ray and a post-mortem. Fuller details of the system can be found in Appendix A.

- Further to the discussion in Section 5.5.2, the figures used for comparison are not the per process figures, but the overall system usage.

- The race referrred to above is avoided by adding some extra user level protocol. The server sends a short greeting to the client after accepting the connection. The client does not make its request before receiving this short message[7]. The exchange is illustrated in Figure 5.2.

With these modifications in place, the method produces consistent, believable and useful information with which to test the hypothesis.

[7]This application level protocol construct is not unusual.

5.6 Scope

I have implemented and tested 35 different variations of the basic http-like server described above, 20 on Windows and 15 on Linux.

Name	Description	Source (on attached CDROM)
Iterative servers		
nativewin	iterative server using native Win32 API only	iterative/win/nativewin.cpp
cygsimple	cygwin port of iterative server	iterative/unix/simple.cpp
uwinsimple	uwin port of iterative server	iterative/unix/simple.cpp
ACE_simple	ACE port of iterative server	iterative/ACE/ACE_simple.cpp
interix	Interix port of iterative server	iterative/unix/simple.c[8]
JSimple	Java port of iterative server	java/iterative/
Fork per connection servers		
cygforker	cygwin port of forking server	concurrent/forker/lin_forker.cpp
interixforker	Interix port of forking server	concurrent/forker/forker.c
Thread per connection servers		
winthread	native thread-per-connection server	concurrent/winthread/winthread.cpp
ace_thr	ACE thread-per-connection server	concurrent/ace_thr/
ServerJMT	Java thread-per-connection server	java/thread_per_conn/
Thread pool servers		
winthrpool	simple native thread-pool server	concurrent/winthread/winthrpool/
ace_thrpool	simple ACE thread pool server	concurrent/ace_thrpool/
ace_msgq	managed ACE thread pool server	concurrent/ace/ace_msgq/
ServerJMTpool	simple Java thread-pool server	java/thread_pool/
Event driven based servers		
winselect	native windows select server	select/winselect/
cygselect	cygwin port of unix select server	select/select/
uwinselect	uwin port of unix select server	select/select/
ace_select	ACE select server	select/ace_select/
tclserv	Tcl event driven server	select/tclserv/

Table 5.1: Windows servers

These variations are illustrative of the different porting strategies described in earlier chapters, but the list is not exhaustive. The following outlines some of the restrictions on the

scope of this experimental work:

- The range of servers tested is naturally limited. The names by which they are referred to in the text, plus accompanying brief descriptions, are given in tables 5.1 and 5.2.

Name	Description	Source (on attached CDROM)
Iterative servers		
simple	iterative server using POSIX API only	iterative/unix/simple.cpp
ACE_simple	ACE port of iterative server	iterative/ACE/ACE_simple.cpp
jsimple.sh	Java port of iterative server	java/iterative/
Select based servers		
select	native select server	select/select/
ace_select	ACE select server	select/ace_select/
tclserv	Tcl event driven server	select/tclserv/
Fork per connection servers		
lin_forker	fork per connection server	concurrent/forker/lin_forker.cpp
ace_forker	ACE fork per connection server	concurrent/ace_forker/
Thread per connection servers		
pthreadd_per_connection	POSIX thread-per-connection server	concurrent/pthread
ace_thr	ACE thread-per-connection server	concurrent/ace_thr/
jmt.sh	Java thread-per-connection server	java/thread_per_conn/
Thread per connection servers		
pthreadd	simple POSIX thread-pool server	concurrent/pthread
ace_thrpool	simple ACE thread pool server	concurrent/ace_thrpool/
ace_msgq	managed ACE thread pool server	concurrent/ace/ace_msgq/
jpool.sh	simple Java thread-pool server	java/thread_pool/

Table 5.2: Linux servers

- Like Stevens, I have used a fixed set of parameters (20000 requests per sample, each returning 4000 bytes) for these experiments. The rationale for choosing the number of requests is twofold:

  1. The number should be large enough to generate reasonably long times. The average run time for a sample is approximately 150sec. With a timer resolution of 10ms this was deemed to be adequate.

2. The number is bounded by the local (ephemeral) port range setting of the client[37]. The default setting of 1024-4999 would only allow some 4000 connections within a 2 minute interval. On my FreeBSD client I have increased the range to 13000-60000, which would make the upper limit 47000 connections per 2 minute interval[9]. 20000 requests is comfortably within this limit.

The choice of request size is somewhat arbitrary, but similar to the original UNPV1 size. Though it would certainly be interesting to see how these servers handle different request sizes (in particular long running ftp-like loads scaling to Mbytes) this must unfortunately be left for future investigation. I have assumed that the performance of the short service (the entire payload is delivered in 3 TCP segments with MSS 1460 bytes), is essentially latency bound. Wrapped functions take longer to execute and contribute to this latency. The observed variations using a short service like this should be a stronger indicator of the efficiency of the wrapped software layers, than variations in throughput for longer running services. The transfer of a 50MByte file is likely to be more strongly influenced by factors such as choice of architecture (single-threaded, fork-per-connection etc), buffer sizes, TCP and driver implementation etc, than whether the server is written in Java or C.

5.7 Results

The relative performance of the servers under test is clearly seen in the tables and graphs which follow. I present the Windows servers first, followed by the Linux servers. Each point on the graphs represents the total CPU usage (ie. time spent in system and user mode) for a complete run of 20000 requests. In order to readily compare similar architectures, the iterative servers are grouped on one graph, the select based servers on another and so forth. Tables 5.3 and 5.4 show the aggregated figures, on Windows and Linux respectively, for 50 runs of each server.

[9] sysctl net.inet.ip.portrange.first=13000; sysctl net.inet.ip.portrange.last=60000.

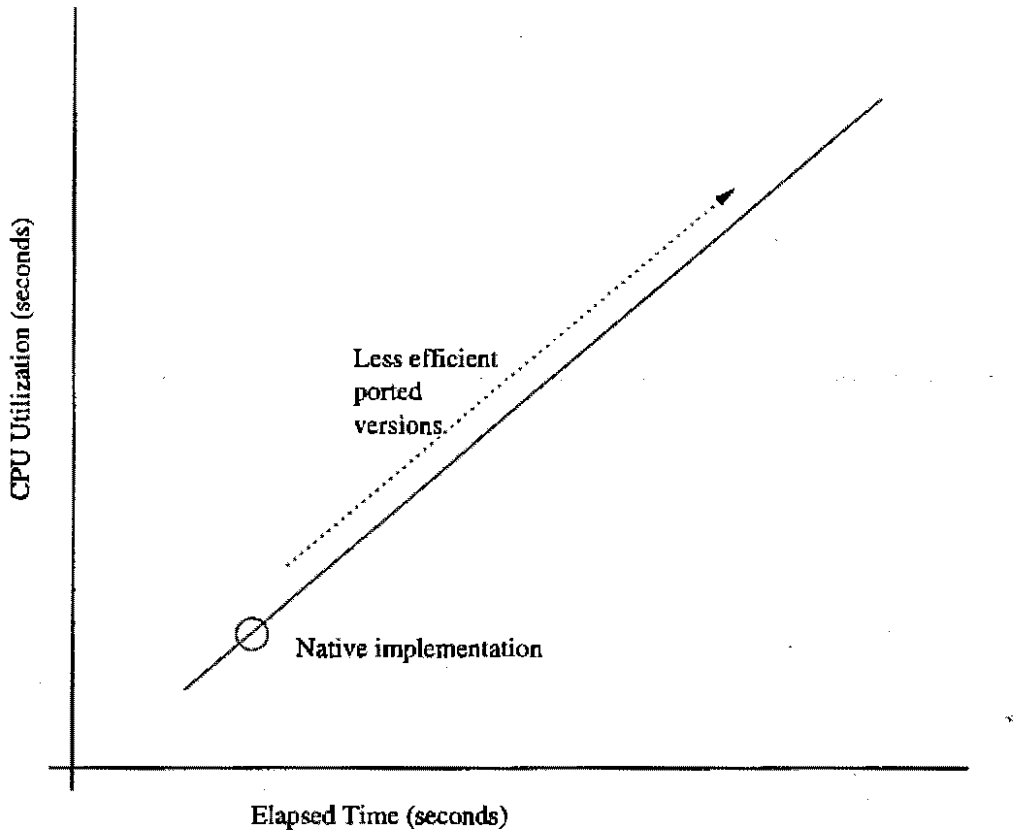


Figure 5.3: Expected profile of ported servers

Figure 5.3 shows what we would expect to see when comparing servers of the same architecture. If the architectures are identical we expect that the same type and number of system calls are being made. Less efficient ports will be spending more time in user space - which should manifest itself in a proportional increase in elapsed time.

In most cases the difference in performance between servers is clearly visible from the graphs. The interesting cases are where there is significant overlap - ie. the ported or portable version appears to perform similarly to the native one. There are a number of cases (using ACE on Linux below) where this is the case. We can use a standard hypothesis test of the difference between two means [22] in these cases to see if the measured data suggests that one performs better than the other or not. For two servers, A and B, with recorded mean CPU utilization times μ_A and μ_B we form two hypotheses:

$$H_0 : \mu_A = \mu_B \tag{5.3}$$

and

$$H_1 : \mu_A > \mu_B \tag{5.4}$$

where H_0 is the *Null Hypothesis*. If we accept H_0 we are assuming that there is no contradiction between the two means, and that any difference can be ascribed soley to random factors. Otherwise we accept hypothesis H_1, that the CPU utilization of A is indeed greater than that of B.

To decide whether to accept H_0 or H_1, we first calculate the standard errors of the difference of means,

$$s_{(\overline{x}_A - \overline{x}_B)} = \sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}} \tag{5.5}$$

where s_A is the standard deviation of sample A, size n_A, and s_B is the standard deviation of sample B, size n_B.

The Z score is calculated using

$$Z = \frac{|\overline{x}_A - \overline{x}_B|}{s(\overline{x}_A - \overline{x}_B)} \tag{5.6}$$

We know (from normal distribution statistical tables) that a Z score of 1.65 or more indicates that 95% of the population supports H_0, ie. there is indeed a difference between the two means. Thus if we calculate a score of less than 1.65 we can say that, at the 5% level, there is nothing to suggest that the means are different. According to [22] such conclusions are valid for relatively large sample sizes ($n > 30$). The means and standard deviations I present here are all for sample sizes of 50.

5.7.1 Windows

Table 5.3 shows the measured times of all the servers on the Windows 2000 platform.

Windows Iterative Servers		
iterative	mean(sec)	stdev(s)
elapsed	164.03	1.36
user	1.50	0.15
system	88.36	0.52
user+system	89.85	0.55
cygsimple	mean(sec)	stdev(s)
elapsed	165.95	1.34
user	1.85	0.14
system	89.52	0.67
user+system	91.37	0.72
uwinsimple	mean(sec)	stdev(s)
elapsed	175.38	2.85
user	4.90	0.21
system	95.13	0.48
user+system	100.02	0.41
interix	mean(sec)	stdev(s)
elapsed	183.00	1.71
user	5.13	0.29
system	97.59	0.99
user+system	102.72	1.12
fsimple	mean(sec)	stdev(s)
elapsed	165.22	1.62
user	3.89	0.32
system	92.11	1.04
user+system	95.99	1.23
ace_simple	mean(sec)	stdev(s)
elapsed	165.22	1.15
user	1.61	0.16
system	89.83	1.24
user+system	91.44	1.29

Windows Event Driven Servers		
winselect	mean(sec)	stdev(s)
elapsed	118.31	4.95
user	3.06	0.15
system	87.76	0.51
user+system	89.82	0.52
ace_select	mean(sec)	stdev(s)
elapsed	115.19	4.14
user	3.40	0.23
system	89.84	0.44
user+system	93.25	0.52
fdisero	mean(sec)	stdev(s)
elapsed	119.97	4.83
user	5.36	0.24
system	92.68	0.60
user+system	98.55	0.59
cygselect	mean(sec)	stdev(s)
elapsed	131.22	3.11
user	8.54	0.29
system	119.11	0.43
user+system	127.75	0.50
uwinselect	mean(sec)	stdev(s)
elapsed	122.20	13.07
user	5.47	0.39
system	94.09	0.57
user+system	99.56	0.66

Windows Fork per connection		
cygforker	mean(sec)	stdev(s)
elapsed	409.55	1.39
user	92.53	1.28
system	315.26	1.19
user+system	408.09	0.85
interixforker	mean(sec)	stdev(s)
elapsed	154.95	1.84
user	12.82	0.37
system	140.23	0.58
user+system	153.05	0.55

Windows Thread per connection		
winthread	mean(sec)	stdev(s)
elapsed	121.31	3.69
user	2.95	0.17
system	92.66	0.50
user+system	95.60	0.52
ace_thr	mean(sec)	stdev(s)
elapsed	127.46	13.90
user	6.40	0.82
system	97.36	0.87
user+system	103.76	0.77
ServerJMT	mean(sec)	stdev(s)
elapsed	136.89	2.02
user	8.07	0.32
system	99.70	0.59
user+system	107.77	0.56

Windows Thread pool		
winthrpool	mean(sec)	stdev(s)
elapsed	117.24	8.87
user	1.62	0.13
system	87.13	0.55
user+system	88.75	0.55
ace_thrpool	mean(sec)	stdev(s)
elapsed	115.50	3.21
user	1.73	0.16
system	87.36	0.55
user+system	89.09	0.60
ace_msgq	mean(sec)	stdev(s)
elapsed	118.59	4.05
user	1.96	0.15
system	87.81	0.52
user+system	89.77	0.52
ServerJMTpool	mean(sec)	stdev(s)
elapsed	128.16	3.82
user	4.26	0.25
system	91.15	0.62
user+system	95.41	0.68

Table 5.3: Windows Results

Iterative servers

Figure 5.4 shows the native windows, Cygwin, Uwin, Interix, ACE and Java versions of the iterative server. The trend is similar to what was expected from Figure 5.3 with the exception of the Java server. This server almost matches the throughput of the native version at this load level, but consumes more CPU cycles to do it. With an increased connection rate it can be expected that, as the CPU becomes a more scarce resource, this server will not scale as well.

From the graph the ACE and Cygwin versions appear to be very close to the efficiency of the native version. The detailed figures shown in Table 5.3, however, illustrate that in fact there is a small, but significant performance hit in each case, which is obscured on the graph by the relatively wide spread of results for the ACE server. The fact that some servers show a greater variability of CPU usage than others may be significant or may be incidental. The question remains unanswered within the scope of this research, but warrants further investigation.

The Interix version appears to be particularly sluggish, which is a little surprizing. Given that Interix implements the POSIX system calls at the subsystem level, I had expected that it would at least outperform the Uwin and Cygwin versions, which perform the POSIX emulation above the Win32 subsystem. The POSIX interfaces provided by all three (Interix, Cygwin and Uwin) was complete enough to allow each of the versions to be compiled without change from the Unix source code.

Select based servers

The servers in Figure 5.5 all illustrate a common pattern in the relationship between the elapsed time and the cpu utilization. The distributions of elapsed times show the characteristic long tails which one would expect in measures of network performance. Various factors, probably mostly related to the lower layers of the network stack, can and do cause throughput to be occasionally and variably impaired. Therefore we see a cluster of points at the head of each distribution followed by a tail of straggling points representing the slower runs. The relative insensitivity of the CPU utilization to the variation in elapsed time, lends it credibilty as a reliable and predictable metric to compare performance.

The `select()` based servers illustrate some interesting strengths and weaknesses among

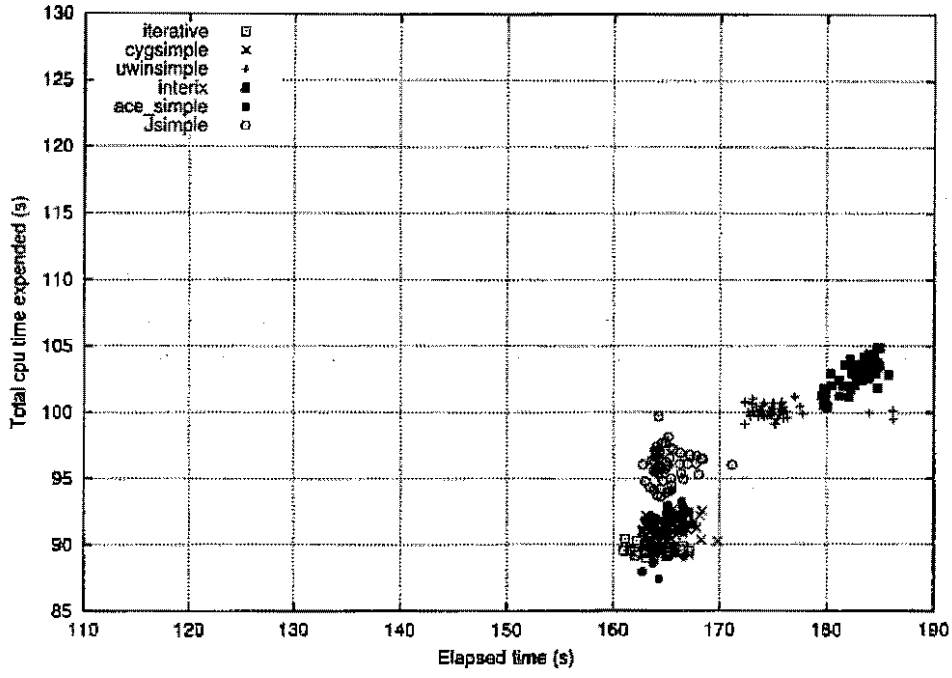


Figure 5.4: Windows iterative servers

the porting layers. Whereas the ACE version continues to perform very close to the native windows equivalent, cygwin fares considerably worse in this case. Cygwin does not use the native Win32 `select()` function to implement Unix `select()` semantics. Its version of `select()` is an elaborate wrapping of the Win32 `WaitForMultipleObjects()` function, involving multiple threads and synchronization event objects. It is clear from the source code and ChangeLogs that a great deal of effort has been put into getting `select()` to work properly with stdio channels and to respond correctly to signals. Even without having the source code for Uwin, it is clear that the implementation of the Uwin `select()` uses a different technique to Cygwin. David Korn tells us in [19] that the Uwin `select()` was originally implemented using a technique similar to that used by Cygwin, but was later changed to use the Windows message queue. This is similar to the Windows version of the Tcl event notifier[10]. Both the Cygwin and Uwin versions are compiled directly off the UNIX source and appear to emulate the UNIX behaviour flawlessly.

[10]The significant overlap of the `tclserv` and `uwinselect` results in Figure 5.5 suggests a similar implementation.

It is interesting to note that the ACE version actually achieves a better throughput than the native version. The **ace_select server** is implemented using the ACE_Select_Reactor component. Its architecture is thus very similar to the others, but not identical. The strategy used to demultiplex and dispatch events within the Reactor (which results in more frequent calls to `select()`) appears to be favoured by these experimental conditions. These strategic choices are discussed in greater detail in section 4.2.

The ACE_Select_Reactor and the Tcl Notifier are also more complete components with a richer set of functionality than the simple demultiplexing component underlying the other servers. The extra overhead in **ace_select** (which is slight, but nevertheless apparent) and **tclserv** is thus to be expected. The Tcl version used here is 8.0p2, which runs the notifier code in a separate thread. Experiments with the latest stable release version (8.3.4) were disappointing - a version of tclserv compiled against 8.3.4 runs in 6 threads on Windows 2000, which results in a considerable performance hit. One reason for using a non-blocking event driven model is to avoid the overhead of multiple threads - here we clearly get the worst of both worlds.

Unfortunately, the source for these servers makes use of a number of ISO Standard C++ features which are not supported in my current Interix development environment, so we cannot see how Interix compares here. It would also have been interesting to see how the new Java non-blocking I/O performed relative to the others, but the jdk version 1.4 was released too recently to be used in these experiments.

Concurrent servers

There are three categories of servers addressed under this category: the classical new-process-per-connection model, the lighter thread-per-connection equivalent and a simple thread pool model.

The POSIX `fork()` system call is one of the more defining differences between the Win32 API and Unix-like systems. Cygwin, Uwin and Interix each provide a `fork()` implementation on Windows. The Linux **forker** program was compiled on Windows using these three tools. Unfortunately, the Uwin version could not complete a run of 20000 connections, so results for it are not given here. It is not clear to me where the problem lies, or whether the problem is solved on later Uwin versions. After fielding about 1000 connections, it simply freezes up

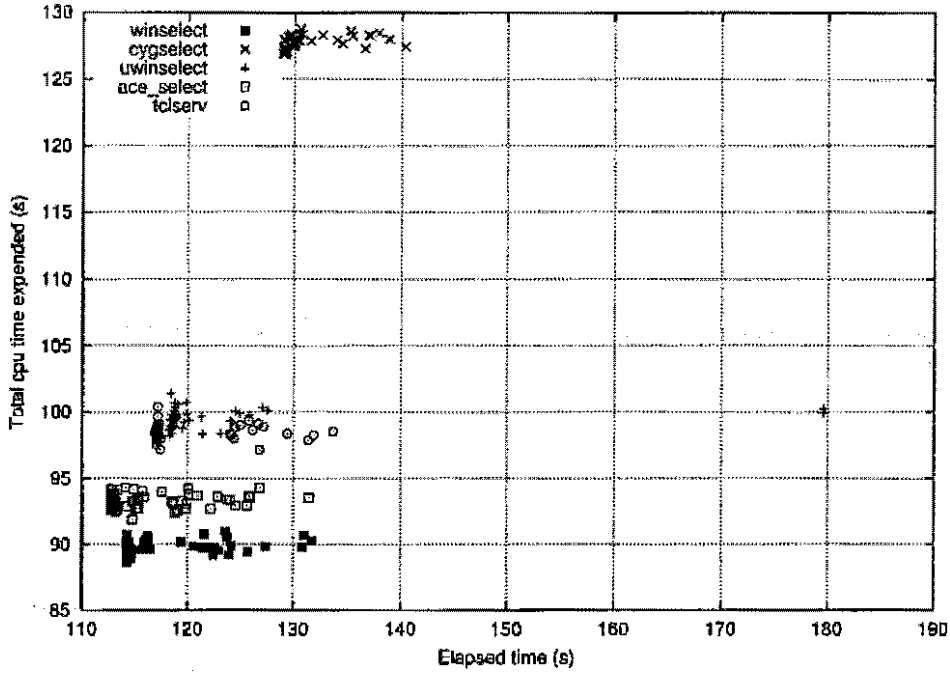


Figure 5.5: Windows select based servers

and refuses to fork any further.

Both Cygwin and Interix handle the task without error, but with very different performance characteristics. Because there are only these two examples, and the fact that they perform so differently, I have not presented them together on a graph, but the figures are clear from Table 5.3. Cygwin is *very* slow and CPU intensive. This is not surprizing, given the complexity of emulating `fork()` using the Win32 API. The Interix version shows that the Windows 2000 kernel interface *does* provide a reasonably efficient way of creating processes other than through the Win32 `CreateProcess()` function. Although it uses some 50% more CPU time than the thread-per-connection servers, this is much more acceptable than the 300% difference when using Cygwin's `fork()`.

Given the difficulty with forking, this server architecture is bound to be penalized the most on the Windows platform. It is clear, however, that if performance is not the key concern, such servers can be ported reliably to Windows platforms. The performance penalty is likely to be much less on pre-forked process pool servers, where the cost of process creation is a one-off cost. Particularly given the Cygwin iterative server's impressive performance in

the iterative tests, there is good reason to believe that a Cygwin process pool architecture would perform well. Unfortunately time did not permit testing this architecture and it remains to be shown in future work how viable the architecture is on Windows platforms.

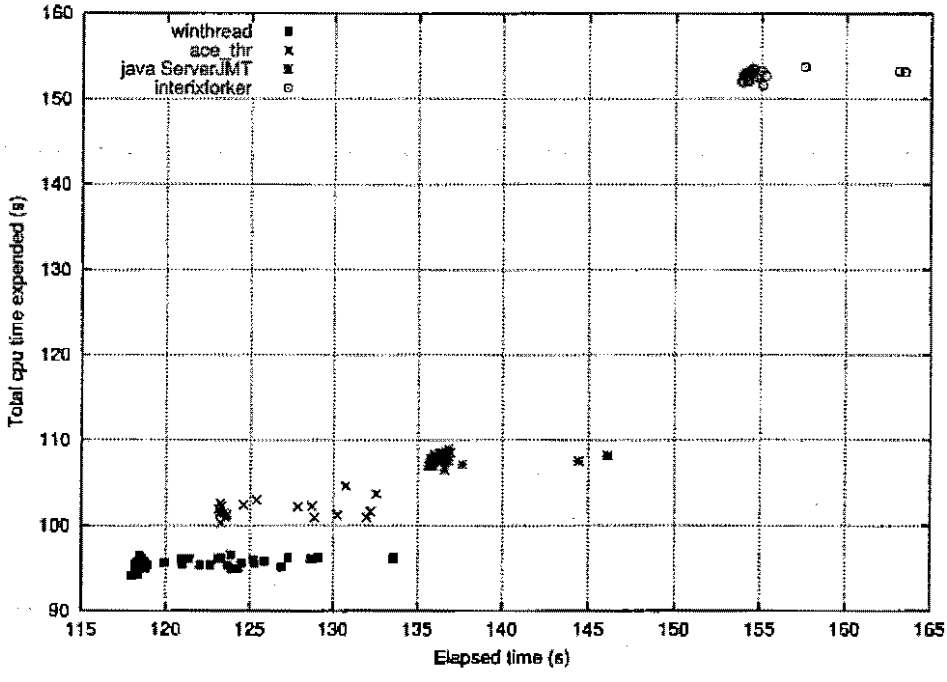


Figure 5.6: Windows thread-per-connection servers

There are only three examples of Windows thread-per-connection servers. Neither Uwin nor Interix supported the Pthreads interface at the time of writing. Current versions of Cygwin support a Pthreads library for Windows which is a thin wrapper around the Windows threads API , but this was not tested. The Tcl library also has had a platform independent threads API since version 8.2.x, but it is not enabled by default and was not tested here.

The three examples presented are the native version (**winthread**), a Java version (**ServerJMT**) and an ACE version (**ace_thr**). The Interix forking server (**interixforker**) is shown on the same graph (Figure 5.6) to illustrate the relative cost of creating a new process per connection *vs* creating a new thread. It is perhaps unfair to make a strict comparison between these three. What they have in common is that they each create a thread to handle each incoming connection, but the implementations differ considerably. The Java version is predictably the slowest and heaviest, but with only 13% more CPU time than the native

version, it is quite respectable. It is interesting to compare the CPU time *vs* elapsed time charcteristics to that seen for the iterative servers. Here we have used 13% more CPU time than the native version which has resulted in a 13% increase in elapsed time. The iterative version showed less than 7% increase in CPU time, resulting in an increase in elapsed time of less than 1%. It would appear that there is a significant cost, in terms of CPU utilization and wall clock time, to the creation of Java threads.

The ACE version performs better than the Java one. Nevertheless the CPU usage difference between **ace_thr** and winthread is greater (8.5%) than the differences seen when using ACE in the other examples. This is because **ace_thr** does not simply use a one to one mapping of ACE wrapped versions of the Win32 API. An intersting feature of **ace_thr** is its use of a Strategy Acceptor [47] to decouple the concurrency strategy (in this case thread-per-connection) from the service handler code. It is an elegant and flexible design pattern, but it does add a small amount of overhead.

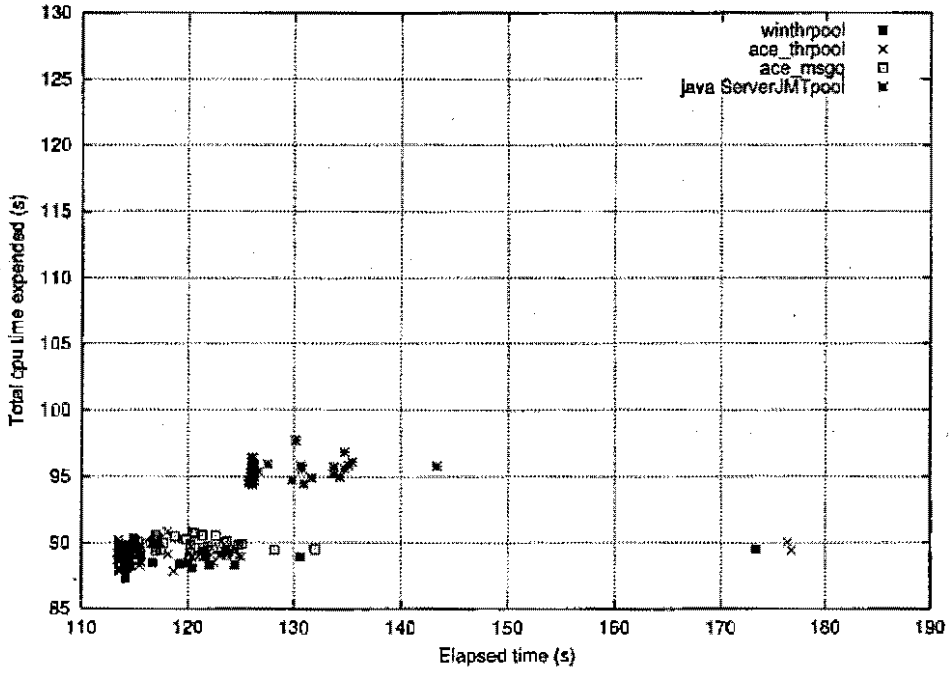


Figure 5.7: Windows thread-pool servers

The thread pool servers also feature a native version (**winthrpool**), a Java version (**ServerJMTpool**) and an ACE version (**ace_thrpool**). They all implement the simple

architecture of creating a listening socket and then pre-spawning a pool of threads which each block in a call to `accept()` on the same listening socket. This architecture has the potential of suffering from the *thundering herd* [55] problem, but with a small pool of 5 threads the effect seems not to be significant. The simple arrangement has each thread acting autonomously and in isolation from one another - effectively a number of iterative server threads sharing the same listening socket. It does not lend itself to dynamic tuning (such as shrinking or growing the size of the pool in response to load conditions). For comparative purposes I have also included an example of a more flexible architecture, **ace_mesgq**, which has a controlling thread to accept new connections and feeds these off to the waiting pool via a message queue. The graph of the performance of these four servers is shown in Figure 5.7.

As before, the Java version is the slowest, but nevertheless respectable - only 7.5% more CPU usage than the native windows equivalent. This is an improvement on the thread-per-connection result, confirming that Java thread creation is relatively heavy.

The performance of the **ace_thrpool** server is very close to that of the native version. Comparing the means using equations 5.5 and 5.6 , we can calculate a Z score of

$$Z = \frac{89.09 - 88.75}{\sqrt{\frac{0.60^2 + 0.55^2}{50}}} = 2.95$$

which is greater than 1.65, indicating that even though they are close, we have to concede that, at the 5% level, the ACE version is slightly slower. The mean elapsed time is actually less than that of the native version, but the large standard deviations of these elapsed time readings due to the long tails, means that there is little we can read into them.

The results for **ace_mesgq** show that the cost of implementing this slightly more complex, but more flexible, architecture is very small. It uses approximately 1% more CPU time and 1% more elapsed time than the native windows implementation of the simpler architecture.

The best performers of all the servers measured on Windows were **winthrpool** and **ace_thrpool**. It is interesting to note that they perform marginally better than the event driven servers, which is similar to what we see on the Linux platform below. Unfortunately we do not have sufficient information to generalize over the general merits of the event-driven *vs.* thread-pool approach on either platform. These results were obtained with a fixed size pool of ten threads and a maximum of five concurrent connections. Banga [3] has shown that

`select()` scales badly with very large numbers of simultaneous connections on BSD derived systems, particularly in a wide area internet environment where many of the connections are passive for much of the time. How well the thread pool approach scales under similar conditions and whether the trends are similar on both Windows and Linux is an unanswered question. Comparing architectures is not the central purpose of this study, but this is a question which merits future work.

5.7.2 Linux

Results for the Linux servers are shown in Table 5.4. The distributions of recorded times are presented graphically in the sections that follow.

Iterative servers

Figure 5.8 shows that the CPU utilization of the ACE iterative server is indistinguishable from that of the native Linux version. Table 5.4 shows that the mean CPU utilization, 80.58s, is actually slightly lower than the native version's 80.64s. The Z score calculated from equation 5.6 is 1.12, which is less than 1.65. We can say with 95% certainty that there is nothing to suggest that the ACE version performs any differently to the native one.

The Java version uses 12% more CPU time than the native version, which is higher than the 7.5% cost that we saw on Windows. The relatively high standard deviation (2.28s) of the Java server is a characteristic we can see in all the Java servers in Table 5.4. Looking at the graphs (Figures 5.8, 5.10 and 5.11) we see that the Java times seem often to form two distinct groups. This is most obvious in Figure 5.10. Revisiting the raw data, it is noticeable that in a series of runs, the first run consistently consumes more time than subsequent ones. Clearly the high startup cost of the Java Virtual Machine is exaggerated when the cache is cold.

Linux Iterative Servers		
simple	mean(sec)	stdev(s)
elapsed	177.52	0.32
user	0.05	0.02
system	80.59	0.33
user+system	80.64	0.33
ace_simple	mean(sec)	stdev(s)
elapsed	177.64	0.37
user	0.09	0.03
system	80.49	0.33
user+system	80.58	0.34
jsimple.sh	mean(sec)	stdev(s)
elapsed	191.17	0.94
user	2.30	0.71
system	88.23	2.01
user+system	90.54	2.28

Linux Event Driven Servers		
linselect	mean(sec)	stdev(s)
elapsed	117.61	1.02
user	0.23	0.05
system	79.74	0.41
user+system	79.97	0.42
ace_select	mean(sec)	stdev(s)
elapsed	115.95	0.12
user	1.03	0.12
system	80.17	0.51
user+system	81.20	0.50
fciserv	mean(sec)	stdev(s)
elapsed	118.02	0.51
user	1.66	0.13
system	80.36	0.45
user+system	82.02	0.46

Linux Fork per connection		
lin_forker	mean(sec)	stdev(s)
elapsed	122.09	0.11
user	1.72	0.14
system	89.73	0.54
user+system	91.45	0.48
ace_forker	mean(sec)	stdev(s)
elapsed	126.76	0.18
user	9.07	0.26
system	102.51	3.13
user+system	111.58	3.19

Linux Thread per connection		
pthreadd_per_connection	mean(sec)	stdev(s)
elapsed	121.59	0.12
user	0.43	0.06
system	81.53	0.64
user+system	81.96	0.65
ace_thr	mean(sec)	stdev(s)
elapsed	122.24	0.14
user	2.45	0.17
system	83.20	0.56
user+system	85.66	0.54
jmt_sh	mean(sec)	stdev(s)
elapsed	131.15	0.46
user	7.15	1.05
system	89.58	0.44
user+system	96.73	1.21

Linux Thread pool		
pthreadd	mean(sec)	stdev(s)
elapsed	121.23	0.13
user	0.10	0.03
system	79.45	0.59
user+system	79.54	0.59
ace_thrpool	mean(sec)	stdev(s)
elapsed	121.55	1.14
user	0.14	0.04
system	79.51	0.64
user+system	79.64	0.63
ace_msgq	mean(sec)	stdev(s)
elapsed	121.10	0.17
user	0.46	0.08
system	80.03	0.71
user+system	80.49	0.68
jpool.sh	mean(sec)	stdev(s)
elapsed	130.71	0.19
user	2.75	0.46
system	85.20	0.83
user+system	87.95	0.92

Table 5.4: Linux Results

Some additional investigation was made into using the gcc Java frontend, gcj, on Linux, to compile the Java code down to a native binary format. The results were encouraging. The same Java iterative server was compiled using gcj and subjected to the same tests. The following times were recorded (in seconds):

Elapsed mean	Elapsed stddev	Total CPU mean	Total CPU stddev
183.38	1.23	82.34	0.41

This represents only a 2% penalty in CPU utilization and 3% penalty in elapsed time, compared to the native iterative version. At the time of writing, gcj and the associated libgcj library was still too immature on Windows to use it to compile the servers in this study, but clearly there is some potential in using this approach to efficiently use Java simply as a platform neutral library. Given that object code produced by gcj can be linked to object code produced from C/C++ source, one would only need to use Java for platform dependent parts of the code. Similar good results were produced for the thread pool server below, but problems were experienced with the thread-per-connection server. More work is required to verify the stability of libgcj.

Select based servers

The trade-offs here are much the same as in the Windows case. The native **linselect** uses less CPU time than the ACE and Tcl versions, but the ACE version delivers a slightly better throughput. As was the case on Windows, the longer elapsed times on the Tcl and native versions compared to **ace_select** are a result of extracting all the events returned by `select()` and queueing them before dispatching.

Concurrent servers

Three categories of server architecture are presented in this section: Fork-per-connection, thread-per-connection and thread-pool based servers.

The native forking server, **lin_forker**, predictably uses more CPU time than the thread-per-connection servers, but the difference is not as striking as with the emulated `fork()` on Windows. Creating a new process on Linux is certainly heavier than creating a new thread, but the `fork()` is clearly optimized.

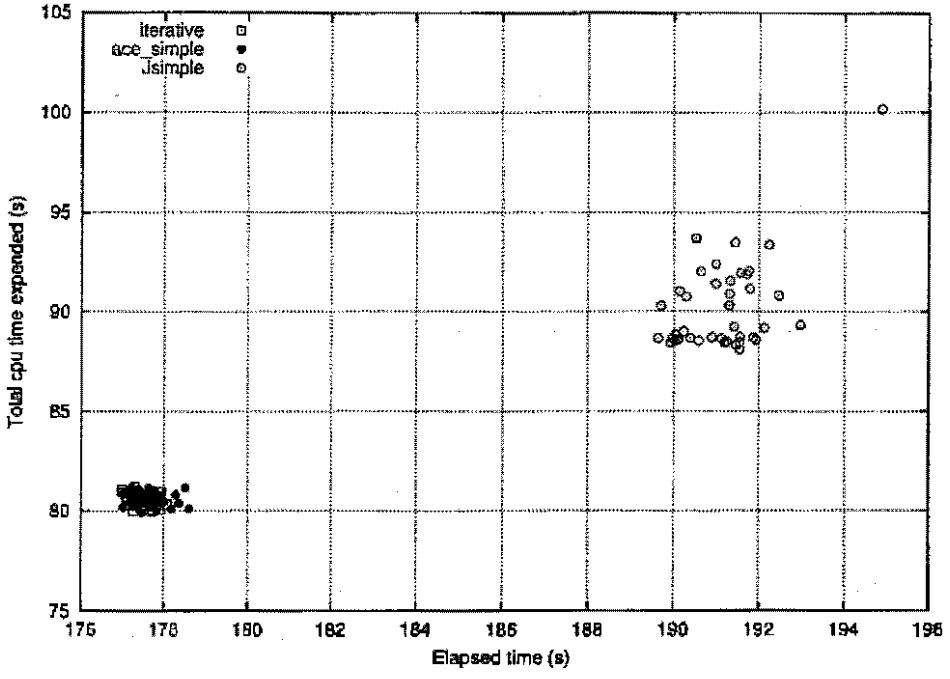


Figure 5.8: Linux iterative servers

The ace_forker provides a caveat here: forking a large process costs more than forking a small process. I have not considered memory usage thus far in the comparisons made, but when processes have to `fork()` frequently, the size of the process has a clear impact on the CPU utilization. Whereas the ACE toolkit has shown itself to be equal, or almost equal, to native versions of the other server architectures, there is a significant cost to be seen here.

Figure 5.11 shows that the CPU utilization of the ACE thread pool servers are indistinguishable from that of the native Linux Posix threads version. The mean for the native version is 79.54s as against 79.64s for the equivalent ACE thread pool server and 80.49s for the ACE message queue thread pool server.

Comparing pthreadd with ace_thrpool, the Z score calculated from equation 5.6 is 0.819, which is less than 1.65, indicating that, at the 5% significance level, there is in fact nothing to suggest that the ACE version performs any worse than the native one.

Comparing pthreadd with ace_msgq, the Z score calculated from equation 5.6 is 7.46, which is significantly larger than 1.65. The ACE message queue server uses only slightly more CPU time, but the difference is significant at the 5% level.

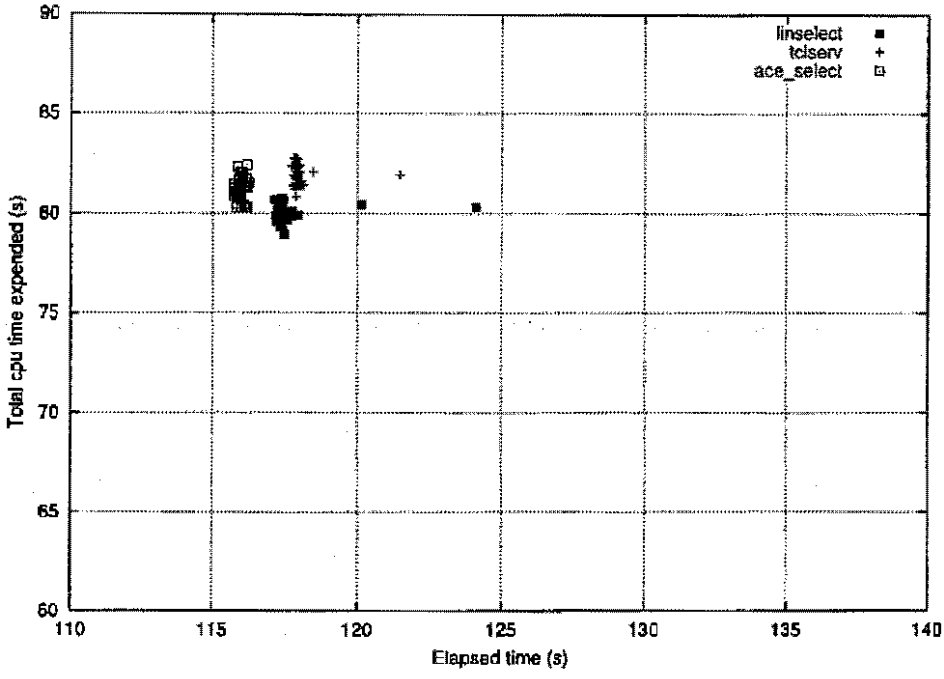


Figure 5.9: Linux select servers

A comparison with the select() based servers is interesting. There is very little difference in terms of CPU utilization, but the elapsed times of the thread pool servers are significantly higher. The higher elapsed times are probably indicative of the increased interrupt latency referred to in Section 5.5.2. The more threads that are running, the more time will be spent scheduling (during which time interrupts are disabled). This issue and the question of how well each architecture scales is left for future work.

The Java version is predictably the heaviest, with a cost of 11% CPU utilization and 8% elapsed time. Again, it is intersting to see the performance improvement we can get by compiling the Java thread pool server to native code using gcj:

Elapsed mean	Elapsed stddev	Total CPU mean	Total CPU stddev
123.04	0.18	81.39	0.48

The table above shows that the compiled Java code only incurs a 2% penalty in CPU utilization and 1.5% penalty in elapsed time, which is a considerable improvement over running the byte code in a Java Virtual Machine.

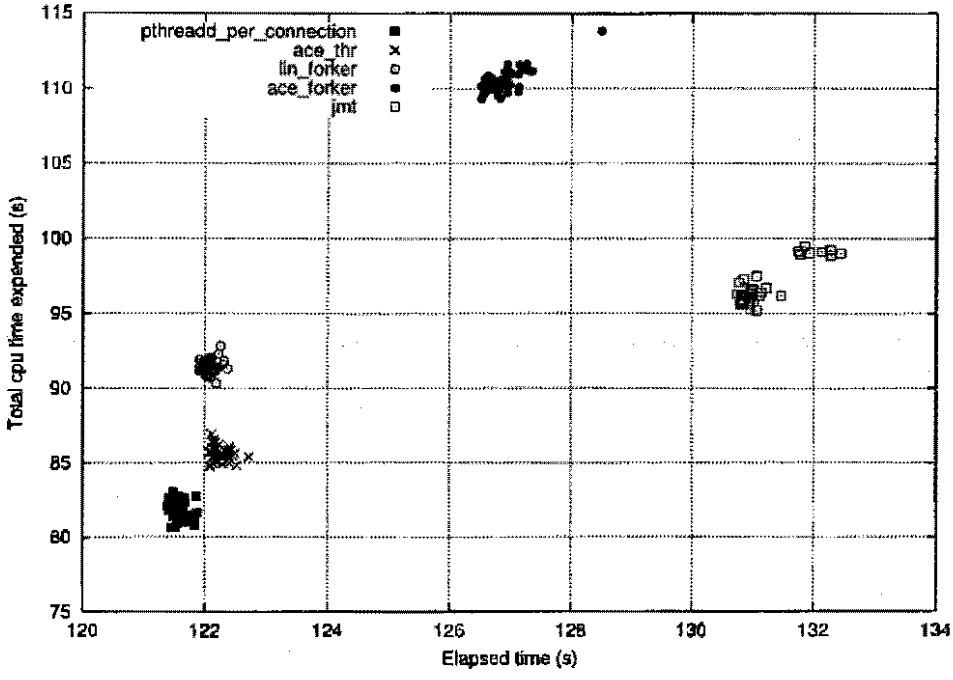


Figure 5.10: Linux thread-per-connection servers

5.7.3 General observations

Having looked at the set of Windows and Linux results there are a number of general observations which should be made:

- There are fewer Linux servers than were tested under Windows. We looked at a number of POSIX emulation tools under Windows. Under Linux we only looked at native versions, ACE, Java and Tcl versions of the various architectures.
- The Windows servers have performed generally better than their Linux equivalents. Reading too much into this comparison would be a little reckless. There are, in both instances, a myriad of tcp options, compiler optimizations and other tuneable parameters which might prove a case either way. Comparing Linux 2.4.3 and Windows 2000 is not the purpose of this study. I have the humbler objective of showing that server architectures are portable, and that the cost of portability, in performance terms, is not necessarily prohibitively high.
- Using ACE on Linux appears to incur no CPU utilization penalty in a number of cases.

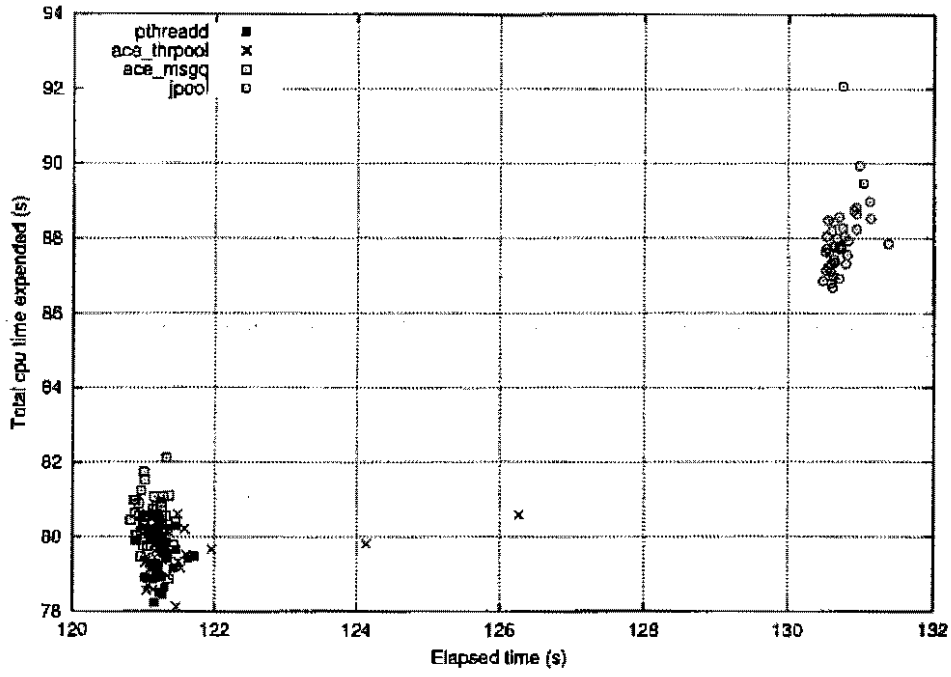


Figure 5.11: Linux thread-pool servers

This in itself is not surprizing, but it does raise the question of why there should be a consistent, if tiny, performance cost associated with the same servers on Windows. There are a number of possibilities, which I have not verified, but which may account for this slight difference:

1. The ACE_SOCK_Stream `close()` method always makes a call to `shutdown()` before `closesocket()` on Windows platforms to avoid losing data that may be still in the socket send buffer. There can never be any unsent data with these servers, because the client initiates the close and then only after it has received all the data. The redundant call to `shutdown()`, multiplied 20000 times may have some impact.

2. It is possible that there is a more efficient combination of compiler optimization switches for the Visual C++ compiler than the ones I have used. For example, I have seen that disabling C++ exceptions can lead to significant performance improvements.

3. I did not compile and use a static version of the ACE library on either platform. It is possible that the relative cost of linking against position independent code in a Windows DLL is higher than linking against shared library code on Linux.

Chapter 6

Conclusion

In Chapter 1 the difference between the system APIs of Windows and UNIX-like systems was identified as a problem when designing TCP/IP server software targeted at both platform sets. In Chapter 3 we saw that the sockets interface to the TCP/IP stack is reasonably similar in both cases. Whereas there are differences, they are not major barriers to portability. In Chapter 4 we saw that the required supporting infrastructure, in terms of event and concurrency mechanisms, present far greater challenges to portability.

I proposed the hypothesis that, by avoiding programming directly to the native system API, but instead making use of thin abstraction layers, it is possible to maintain portable source code which implements common server design patterns. Further, I proposed that such implementations should not necessarily incur significant performance penalties.

This chapter presents a summary of my conclusions. Section 6.1 deals with portability at the level of server architecture. This is followed by a review of implementation options in section 6.2. In this section I show that whereas my hypothesis is largely true, there are important caveats. Almost inevitably, in a work of this nature, there is far more unconcluded than there is concluded. Section 6.3 proposes directions for future work.

6.1 Portable architectures

Simple iterative servers are trivially ported between platforms which provide some form of BSD sockets interface. The Winsock 2 implementation of BSD sockets is reasonably similar

to Unix implementations. Areas of difference such as the treatment of error codes, ioctl options and the semantics of `close()/closesocket()` and `shutdown()` are addressed in Chapter 3. They do not present a significant challenge in terms of portability and can easily be accommodated with simple Wrapper Facades and pre-processor macros.

In order to be useful, servers must usually be able to handle a number of concurrent connections (referred to as the *capacity* of the server). Architectural patterns to support concurrent connections fall into three categories:

- Single process event driven (SPED)
- Thread-per connection
- Threadpool

Multiple process servers fall into one or other of the latter two categories. For each of these, the service handler code is dispatched in its own thread context. Whether the threads exist within a single process, or are distributed across processes, the resulting architecture is generally one of the latter two from the list above.

The following two sections describe the portability of event driven and multi-threaded servers respectively.

6.1.1 Event mechanisms

The primary portability issue with event driven servers revolves around the different demultiplexing mechanisms. Winsock provides a `select()` function which is sufficiently similar to the BSD derived equivalent to be a useful common denominator. Both Windows and Unix provide a range of alternative mechanisms to `select()`. On Windows there is `WaitForMultipleObjects()`, Asynchronous Procedure Calls (APCs) as well as the older message based asynchronous I/O of Winsock version 1. The various Unix-like flavours support different mechanisms such as the POSIX `poll()` system call, signal driven I/O as well as the newer explicit kernel event queue mechanisms supported by BSD kernels. Fortunately the process of building software frameworks to dispatch service handlers in response to events, allows one to decouple the dispatching mechanism from the actual platform-specific event

detection mechanism. The resulting component (such as the Tcl Notifier and ACE reactor discussed in Section 4.2) presents a platform-neutral interface.

6.1.2 Threads and processes

Forking a new process to handle an incoming connection is a common idiom on Unix-like platforms. In Section 5.7.1 we saw that this idiom does not translate well to Windows platforms. The Win32 API lacks the `fork()` system call and uses `CreateProcess()` to create new processes. `CreateProcess()` must load its process image from a disk file and is thus similar to a combined `fork()` plus `exec()`. POSIX emulation toolsets such as Cygwin and Uwin provide an emulated fork, but such emulations are necessarily inefficient - two levels of copying are required: the process must be created in a suspended state, which still involves copying the image from the disk file, and then this image must be overwritten with that of the parent process. The Uwin version proved to be unreliable under sustained load conditions. Table 5.3 in Chapter 5 shows that the Cygwin version of the forking server performed successfully, but at considerable cost in terms of CPU cycles and througput.

The Windows 2000 kernel primitives for creating processes may well have a means of avoiding the double copying referred to above. Interix is a POSIX environment subsystem implemented on top of the Windows 2000 kernel, ie bypassing the Win32 environment subsystem. The Interix version of the forking server consumed less than 40% of the CPU cycles used by the Cygwin version. Whereas it is still considerably more heavyweight than the single process servers it is clearly more efficient than emulating `fork()` through the Win32 API.

The fact that `fork()` has been implemented on Windows using these two approaches is probably more significant than the efficiency of the implementations. Even on Unix platforms, the fork-per-connection model is not the best model to use when performance is the major criterion. The Cygwin implementation, which was by far the slowest of all the servers tested, still managed to service 20000 connections in 409 seconds - that is 48 connections per second which may well be adequate for a wide range of applications.

Having multiple threads within a single process presents fewer problems. In section 4.3.2 we saw that the thread creation semantics of POSIX Pthreads calls and Win32 calls are very similar. They introduce no substantial barriers to porting multithreaded programs

between Unix-like and Windows systems. Synchronization primitives, such as mutexes and semaphores, are present and also exhibit similar behaviour. The lack of an equivalent to POSIX condition variables on Win32 systems does present a non-trivial challenge, which is discussed by Schmidt [46].

Given the similarities, it may seem surprizing that neither Cygwin, Uwin nor Interix provided a Pthreads interface. In fact this has less to do with the difficulty of porting Pthreads, and more to do with the thread safety of the respective runtime environments. This is an area which has been under active development (at least in the Cygwin and Uwin projects) since this work began. At the time of writing Cygwin does provide a substantial portion of the Pthreads API.

6.2 Portable implementations

Given that the architectures discussed can be ported between platforms, the question remains of how best to implement these and what the associated costs and benefits are. There are two different scenarios which were presented in Chapter 2. In the one case we attempt to leave the existing platform specific code untouched, and instead provide an intermediate porting layer to support the code on the foreign platform. In the other, we consider the case of writing software from scratch with portability as a specific design goal. The merits of these two approaches are presented in sections 6.2.1 and 6.2.2 below.

6.2.1 Making existing code portable

Two approaches to making existing Unix code portable to Windows systems were presented. Using a POSIX emulation layer (Section 2.2) and using a POSIX Windows NT environment subsystem (Section 2.4). Two examples of emulation layers were used: Cygwin (Dll version , an open source project from RedHat, and Uwin, a commercial product from AT&T Research Labs. The environment subsystem used was Interix, formerly from Softway Systems, now owned by Microsoft Corporation[1]. A brief overview of these systems was given in Chapter 2.

[1]The latest Interix distribution from Microsoft has been incorporated into their SFU (Services For Unix) product.

Each of the systems reviewed provides more than just access to a UNIX-like API. They also provide (indeed require) a development environment including header files, libraries, shells and other utilities as well as a supporting runtime environment. Whereas this may be a considerable amount of infrastructure for a simple porting project, there are significant benefits:

- No source code changes were required when recompiling code originally developed on Linux. The Windows versions of iterative, fork-per-connection and select based servers in Table 5.1 were all compiled using the same source as the Linux versions in Table 5.2.
- Having a fairly complete runtime environment means that one can also make use of existing Makefiles and configure scripts. Though this was less of an issue with my simple servers, it can contribute significantly to the maintenance effort for larger software projects.

A significant feature of all three systems is support for the `fork()` function, as discussed in Section 6.1.2 above. We saw in Section 5.7.1 that the performance of the three implementations varied greatly. The Uwin forking server disappointingly failed to handle the load of 20000 connections - I failed to establish what the problem was. Both the Cygwin and Interix versions were reliable, though the Cygwin `fork()` is clearly very slow.

The Interix development environment is currently far better suited to compiling C source than ANSI C++. The Interix frontend to the Microsoft Visual C++ compiler accepts C code only. An alternative compiler, an early version of gcc (2.7.2), is bundled with the development environment, but this version has poor ANSI C++ conformance. This limitation is unfortunate and is likely to be corrected in future releases. It should be possible, for example, to compile a later version of gcc using the bundled gcc 2.7.2.

The performance results of Table 5.3 show that there is in each case a performance penalty when using any of the three toolkits, but each appears to have different strengths and weaknesses. In the simple iterative server tests the Cygwin version performed almost as well as the native version but used 42% more CPU time for the select based server. Uwin, on the other hand, suffered a performance hit of 11% CPU utilization for the iterative tests, but also only 11% for the select based server. Interix performed slightly worse in the iterative

tests (14% extra CPU utilization penalty) but, as discussed in Section 6.1.2 above, has a relatively efficient `fork()` implementation.

In summary we can conclude that POSIX emulation can be done quite effectively on Windows NT derived systems, but expect some performance penalty when using Unix paradigms which are particularly foreign to the native system. If raw performance is not the primary design goal (as frequently it isn't) then this is a painless way of porting applications. Each of the three systems discussed have been used effectively to port large amounts of existing source code to Windows. Both Uwin and Cygwin have seen active development over the past few years, with frequent version updates. We can reasonably expect that their functionality and efficiency will continue to improve. Interix has been absorbed into Microsoft's Services For Unix product where it forms the heart of their Unix legacy applications ports (inetd, telnetd, nfsd etc).

6.2.2 Portability from the outset

Designing software with portability as a specific design goal is a different problem to porting existing code. For a given set of requirements, there may be many choices open to the developer. The choice of implementation language can play a significant role in the ease of development and effectiveness of a portable solution.

C style code

We saw in Chapter 3 that the BSD sockets API and indeed the rest of the system API of the platforms considered are C language API's. Whereas programming close to the system API may be desirable in terms of run-time efficiency it can also create the most significant maintenance problems. We saw in Section 3.3, that the preprocessor can be used to isolate platform specific versions of the code for conditional compilation. The physical complexity of such an approach frequently results in code which is prone to error, difficult to read and maintain. This physical complexity can be contained to a certain extent by concentrating the ``#ifdef ...#endif'' preprocessor directives into a single file. Jon Snader [53] gives an example of one such approach in his book on TCP/IP programming[2].

[2]The source code for his examples are available at http://pw1.netcom.com/ jsnader/etcpsrc.tgz.

We can reduce this physical complexity by abstracting through the use of library functions. We saw in Section 2.3.1 how the Tcl C library can be used as an example of such an abstraction layer. Whereas the Tcl library is not a purpose written communication library, it is widely ported and provides a ready made channel abstraction which removes the necessity of programming directly to the sockets layer. The Tcl Notifier provides a convenient and portable event mechanism which makes it a particulary suitable candidate for coding event driven servers. Tables 5.3 and 5.4 show that the CPU utilization cost of a Tcl library based server over a native coded `select()` based server is remarkably low. Less than 10% on Windows and only 1% on Linux.

C++ code

The object oriented language features of C++, particularly in the implementation of Wrapper Facades, can be effectively used to reduce software complexity and hence increase readability and robustness. The Ace toolkit (described in Chapter 2) makes use of Wrapper Facades and other patterns to provide communication software frameworks which combine flexibility, reusability and portability in an extremely efficient way. In Chapter 5 we saw that portable iterative, event driven and multi-threaded servers could be implemented which showed neglible performance penalties compared to their native coded equivalents. Indeed in section 5.2 we saw three cases on Linux where there was no way to statistically differentiate their performance from their native equivalents.

These results provide the strongest support for my hypothesis that portability does not necessarily imply a significant penalty in efficiency. In the three cases referred to above there is no CPU utilization penalty at all[3].

Virtual machines

Java is a very popular choice for cross platform development. We saw in Chapter 2 that Java compilers produce byte code targetted at the Java Virtual Machine. The portability problem is completely delegated to the virtual machine layer. One of the limitations of Java has been the lack of support for non-blocking I/O. This has meant that, though Java could be used to implement all of the multithreaded architectures of Section 4.3, it was not possible

[3]Note that we are still ignoring the question of memory efficiency here.

to implement the event driven model of Section 4.2. The new Sun Jdk v1.4 has introduced support for non-blocking I/O, but I am unaware of any independent performance studies done on this at the time of writing.

The Sun Jdk v1.3 was used on both Windows and Linux for the experiments described in Chapter 5. The server optimized version of the virtual machine was used on Linux, whereas only the client virtual machine was available to me for the Windows experiments. The results (Tables 5.3 and 5.4) show that, for the simple iterative servers, Java performs well in terms of connections handled per second, but there is a significant cost in CPU utilization on both platforms. The multithreaded servers show a similar penalty in CPU utilization, but also show a drop-off in the number of connections handled per second on both platforms. This seems to indicate that the Java thread wrappers introduce some additional latency to the underlying PThreads and Win32 threads API.

6.3 Future work

A large part of the experimental aspect of this work involved looking at overall CPU utilization as a measure of efficiency. Whereas there are hazards associated with interpreting such a measure too simplisticaly, it is significant that the experiments revealed few surprizes. The figures, by and large, revealed simple and consistent characteristics. Future work is required to establish if there are useful predictive extrapolations to be made between this measure of efficiency and more established external performance measures such as latency and throughput under various test load conditions.

The Orchestrator distributed test suite used in the experiments presented in this work, was designed with flexibility and extensibility as primary design goals. The scope of this work required that only a single, simple load client be implemented. More work needs to be done to extend the range of load generating and monitoring services within the framework.

POSIX Asynchronous I/O (aio) and Windows overlapped I/O have similar semantics, yet none of the POSIX emulation tools on Windows support this I/O mode. The increasing number of UNIX-like systems providing better aio support coupled with the dominance of overlapped I/O as the preferred I/O mode on Windows, raises the challenge of an aio port to Windows. The open source Cygwin environment may well be the best avenue to pursue

this work.

The Microsoft .NET framework has been released since this work was started. FreeBSD and Linux ports of the .NET virtual machine have already appeared. Though sharing the VM concept with Java, there are a number of substantial differences in the implementation and design rationale of the two [10]. Counting the CPU cycles used by the .NET VM under the same conditions presented here may well provide some useful insights into the efficiency of this new technology.

Appendix A

Orchestrator - a distributed test suite

Orchestrator is a system designed to facilitate the control and monitoring of server load test experiments in a heterogeneous distributed environment. The use of CORBA as a middleware abstraction layer allows a high degree of flexibility and configurability.

The system was designed to facilitate the gathering of the data presented in Chapter 5, but is sufficiently general that it could easily be adapted to meet different experimental requirements.

A.1 Overview

In order to conduct experiments such as those described in Chapter 5, we can identify a number of participants:

1. the server process under test;
2. one or more load generating clients;
3. a supervisory process to oversee and capture results.

To assist the supervisory process, it is also useful to provide a naming service. The supervisory process can query this service to determine where to find the server to test and where the load generating clients are.

I have implemented a number of CORBA objects which map onto these participants. Figure A.1 shows a schematic representation. The machine on which the server(s) are to be

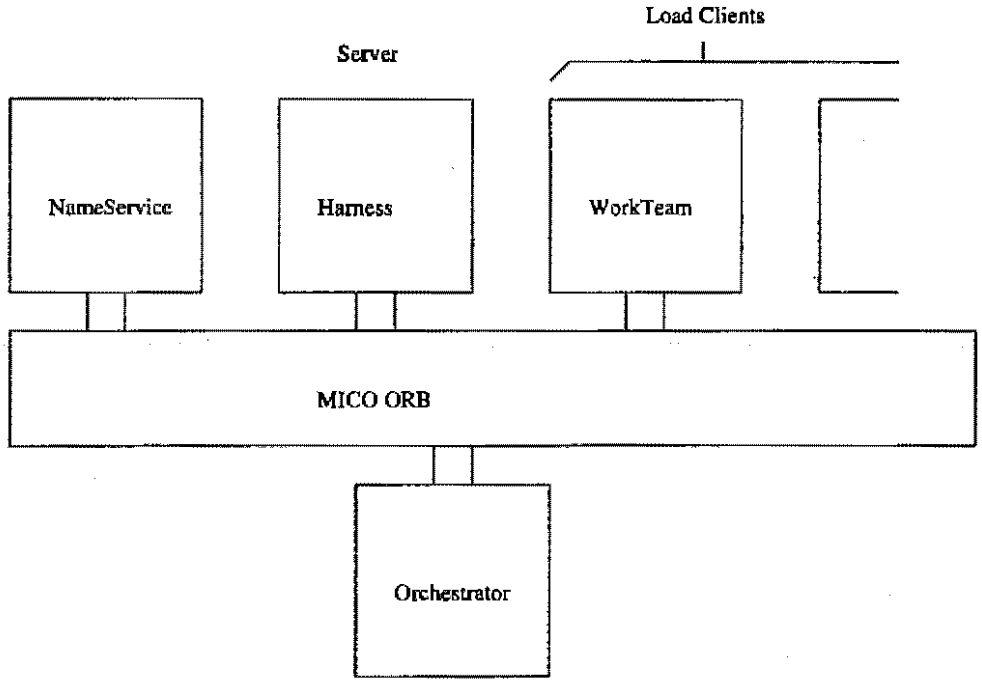


Figure A.1: Test Environment

tested exposes an object called Harness. Harness objects offer an interface for starting and stopping processes, as well as methods for querying the CPU utilization on the machine. A benefit of exposing such an interface is that the platform specific implementation of these methods is hidden from the user of the object.

Similarly, WorkTeam objects reside on load generating client machines. Users of a WorkTeam object can specify parameters to the load generator and start and stop load generating activity.

The Harness and all WorkTeam objects register themselves with the CORBA NameService on startup. The organization of the naming service is shown in the tree below:

```
\--orchestrator-- servers -- <hostname> -- Harness
                  |
                  +--- clients -- <hostname> -- WorkTeam
                                  |
                                  + -- <hostname> -- WorkTeam
                                  |
                                  ...
```

I call the supervisory process the *orchestrator*. The orchestrator runs in a single process on a machine separate from the server and load generators. By querying the name service the orchestrator can acquire a reference to the distributed Harness object. Using this reference it can start (and stop) server processes on the remote test machine. The same orchestrator process can also acquire references to one or more WorkTeam objects. Armed with these references it is possible to conduct an experiment by calling on the WorkTeam(s) to generate a load on the server. Once the server has handled the required number of connections (20000 connections were used in my experiments) the orchestrator queries the Harness object for the CPU utilization figures. At this point it can kill the running server process and repeat the cycle.

The CORBA IDL description of the harness object is shown in Figure A.2 below:

msrc/harness/harness.idl

```
1  // harness.idl
2  // IDL

3  struct usage {
4    double user;            // process user time in seconds
5    double system;          // process system time
6    double total_user;      // total user time in seconds
7    double total_system;    // total system time
8  };
9
10 interface harness
11 {
12   boolean start(in string cmd);  // Start process
13   boolean signal(in short signum); // Kill process
14   boolean getcurrent(out usage times); //Current usage
15 };
```

msrc/harness/harness.idl

Figure 1.2: Harness IDL

Harness objects are very simple. The start method can be used to start an arbitrary process on the machine by providing the name of the executable[1]. What distinguishes it

[1]There are no security safeguards currently built in - anyone with a handle on a Harness object can run arbitrary processes. Building in a security model would be possible, but was not deemed necessary in a controlled environment.

from a simple rsh type command which one would use on a UNIX-type system, is that the interface disguises the platform specific details of creating and monitoring the process on either UNIX or Windows systems.

Figure A.3 shows the IDL for the WorkTeam object:

msrc/wteam5/load.idl

```
24 interface WorkTeam {
25   // The init method must be called prior to connections being made
26   // Specify the ip address in dot format and the listening port
27   //    of the server under test
28   boolean init(in string host,
29        in unsigned short port,
30        in unsigned long numbytes);

31   // dowork is a blocking call
32   // it does not return until all the work is done or if there has
33   // been an error - in which case it returns false
34   // Parameters are:
35   //       numpararallel - the number of concurrent worker threads to deploy
36   //       numbytes - the number of bytes to request for each connection
37   //       num_iter - the total number of connections to make
38   //       time - total real time in seconds for all threads to complete
39   // either num_iter or time must be specified as non-zero to define the
40   // amount of work to be done.  On return both will contain valid data
41   // Returns no of failed attempts

42   long dowork( inout long num_iter,
43   inout double time,
44   in long numparallel);
45 };
```

/msrc/wteam5/load.idl

Figure 1.3: WorkTeam IDL

The CORBA implementation used was the excellent open source Mico ORB from the University of Frankfurt.

A.2 Scripting the orchestrator

Mico is a C++ CORBA implementation. The Harness and WorkTeam classes are also implemented in C++ because they need to access low level system API's on the target

platforms. The CORBA IDL is, however, language independent - we can implement the orchestrator in any language which has CORBA IDL bindings. Using C++ (or Java) for this task is neither the simplest nor the most desirable in this context. Ideally we would like to be able to flexibly configure different experiments using simple configuration scripts.

Tcl is a language ideally suited to this problem. Automated software testing is one of its traditional strengths [21] and creating Tcl bindings to the CORBA IDL is a simple process. Better still, an existing Tcl-IDL binding could be used. I made use of of the Combat (originally Tcl Mico) Tcl extension [34] for this purpose.

A simple orchestrator script might then look like the following:

```
set wt1name "/orchestrator/clients/starship/WorkTeam"
set harnessname "/orchestrator/servers/bobsdell/Harness"

# look up handles from nameservice
set harnesshndl [get_handle $harnessname]
set wt1hndl [get_handle $wt1name]

#initialise the workteam:
$wt1hndl init 192.168.0.10 60002 4000

# ask harness to run "myserver"
$harnesshndl start "myserver"
puts stderr "Server starting ..."
set wt1hndl [get_handle $wt1name]

# pause a second to let things settle
after 1000

# spawn 5 workers to make 20000 connections
$wt1hndl dowork 20000 300 5

$harnesshndl getcurrent R
puts stderr "$server Usage: $R"

# send SIGINT to "myserver"
$harnesshndl signal 2
```

The actual scripts used are more complex in that they must be capable of running batch jobs to test lists of servers robustly. A combination of awk and tcl scripts are used on the

orchestrator output to calculate statistical properties and prepare the data for plotting with gnuplot.

A.3 Test environment

Figure A.4 shows the physical test environment used. The test servers were run on `bobsdell` which dual-boots Linux and Windows 2000. The orchestrator process was run on `starship`. WorkTeam loadservers were run on `bobsdell` and `boys`.

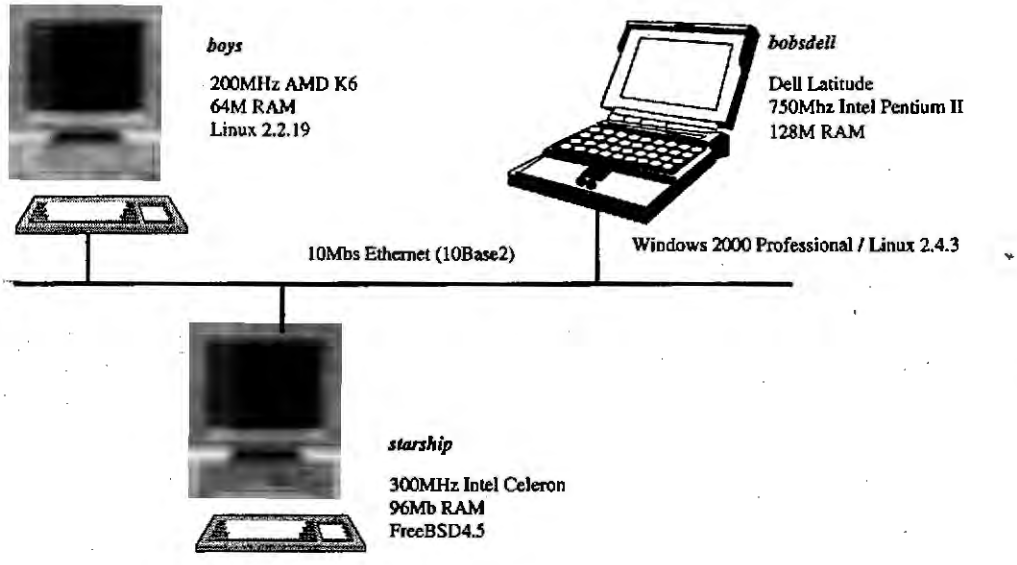


Figure A.4: Test Environment

The 10Mbs ethernet network was the bottleneck which determined the maximum throughput. This effectively ensured that the server would never be saturated - typically it was running at around 30% CPU utilization.

This was not a limitation for the experiments carried out in this work, as I was primarily concerned with the total number of cycles used to process a fixed number of requests.

A.4 Related work

There a number of other examples of using CORBA in a testing and monitoring environment. The Open Group proposed a Universal Measurement Architecture [32] which suggests

CORBA as a good candidate for distributed measurement.

Geihs and Gebauer [9] developed a general load monitoring tool using CORBA and Tcl. Whereas their tool is more a more complete monitoring system than mine, it does not not include the load generation aspect.

A weakness of my orchestrator system is that it does not take dynamic real-time measurements. The system must reach a steady state (ie complete a fixed number of tasks) before the measurement is taken. Real-time extensions [60] to the CORBA standard can be used to implement a more dynamic flexible measurement system. Harrison et al [12] implemented a Real-time CORBA Event Service which could meet the Quality of Service requirements for low-latency, predictable real-time measurement.

Appendix B

Software versions

The software versions used in this work are as follows:

	Windows	**Linux**
OS version	Win2000 Professional SP3	2.4.3 (Mandrake 8.0)
ACE	5.2	5.2
Tcl	8.0p2	8.3.2
Cygwin	1.3.10	-
Uwin	2.9	-
Java	Sun Hotspot VM 1.3 (Client)	Sun Hotspot VM 1.3 (Server)

Table B.1: Software versions

Appendix C

Cost of Function calls

The ACE toolkit includes a number of sample applications which measure various aspects of the system making use of high resolution timers. I include the results here of the function call tests on Linux and Windows systems. The Linux system is using gcc2.96 and the Windows sytem, Visual C++ 6.0.

C.1 Linux

```
bobsdell.cs.up.ac.za (i686), Linux 2.4.3-20mdk at  18:00:37.457912
10000000 iterations
An empty iteration costs    0.002 microseconds.

operation                                  time, microseconds
=========                                  ==================
global function calls:
  inline function call                              0.000
  non-inline function call                          0.005

member function calls:
  inline member function call                       0.000
  non-inline member function call                   0.008

member function calls, class has a virtual function:
  inline member function with virtual call          0.000
  non-inline member function w/virtual call         0.008

virtual member function calls:
```

```
  virtual member function call, optimizable     0.009
  virtual member function call                  0.009
```

C.2 Windows 2000

```
BOBSDELL (Intel Pentium Pro), Win32 Windows NT 5.0 at  12:59:27.430000
10000000 iterations
An empty iteration costs    0.031 microseconds.

operation                                 time, microseconds
=========                                 ==================
global function calls:
  inline function call                            0.000
  non-inline function call                        0.009

member function calls:
  inline member function call                     0.000
  non-inline member function call                 0.008

member function calls, class has a virtual function:
  inline member function with virtual call        0.000
  non-inline member function w/virtual call       0.007

virtual member function calls:
  virtual member function call, optimizable       0.007
  virtual member function call                    0.008
```

Appendix D

TCPdump profiles

In each of the following two tcp dumps, the client (starship) is a FreeBSD4.5 machine. The server (bobsdell) is running a simple iterative service.

Windows 2000 on bobsdell

```
00:13:37.518468 starship.30384 > bobsdell.60002: S 460880889:460880889(0) win 65535
                <mss 1460,nop,wscale 1,nop,nop,timestamp 4052004 0>
00:13:37.519470 bobsdell.60002 > starship.30384: S 2710004082:2710004082(0) ack 460880890 win 17520
                <mss 1460,nop,wscale 0,nop,nop,timestamp 0 0>
00:13:37.519569 starship.30384 > bobsdell.60002: . ack 1 win 33304 <nop,nop,timestamp 4052004 0>
00:13:37.520394 bobsdell.60002 > starship.30384: P 1:8(7) ack 1 win 17520 <nop,nop,timestamp 437493 4052004>
00:13:37.521205 starship.30384 > bobsdell.60002: P 1:5(4) ack 8 win 33304 <nop,nop,timestamp 4052005 437493>
00:13:37.524723 bobsdell.60002 > starship.30384: . 8:1456(1448) ack 5 win 17516 <nop,nop,timestamp 437493 4052005>
00:13:37.527066 bobsdell.60002 > starship.30384: . 1456:2904(1448) ack 5 win 17516 <nop,nop,timestamp 437493 4052005>
00:13:37.527821 bobsdell.60002 > starship.30384: P 2904:4008(1104) ack 5 win 17516 <nop,nop,timestamp 437493 4052005>
00:13:37.527999 starship.30384 > bobsdell.60002: . ack 2904 win 32560 <nop,nop,timestamp 4052005 437493>
00:13:37.528615 starship.30384 > bobsdell.60002: F 5:5(0) ack 4008 win 33304 <nop,nop,timestamp 4052005 437493>
00:13:37.529050 bobsdell.60002 > starship.30384: . ack 6 win 17516 <nop,nop,timestamp 437493 4052005>
00:13:37.529185 bobsdell.60002 > starship.30384: F 4008:4008(0) ack 6 win 17516 <nop,nop,timestamp 437493 4052005>
00:13:37.529299 starship.30384 > bobsdell.60002: . ack 4009 win 33304 <nop,nop,timestamp 4052005 437493>
```

Linux 2.4.3 on bobsdell

```
00:57:04.536136 starship.30404 > bobsdell.60002: S 2811444960:2811444960(0) win 65535
                <mss 1460,nop,wscale 1,nop,nop,timestamp 4312702 0>
00:57:04.536880 bobsdell.60002 > starship.30404: S 1799787272:1799787272(0) ack 2811444961 win 5792
                <mss 1460,nop,nop,timestamp 59771 4312702,nop,wscale 0>
00:57:04.537065 starship.30404 > bobsdell.60002: . ack 1 win 33304 <nop,nop,timestamp 4312702 59771>
00:57:04.537977 bobsdell.60002 > starship.30404: P 1:8(7) ack 1 win 5792 <nop,nop,timestamp 59771 4312702>
00:57:04.538358 starship.30404 > bobsdell.60002: P 1:5(4) ack 8 win 33304 <nop,nop,timestamp 4312702 59771>
00:57:04.538810 bobsdell.60002 > starship.30404: . ack 5 win 5792 <nop,nop,timestamp 59771 4312702>
00:57:04.542002 bobsdell.60002 > starship.30404: . 8:1456(1448) ack 5 win 5792 <nop,nop,timestamp 59771 4312702>
00:57:04.543210 bobsdell.60002 > starship.30404: . 1456:2904(1448) ack 5 win 5792 <nop,nop,timestamp 59771 4312702>
00:57:04.543940 bobsdell.60002 > starship.30404: P 2904:4008(1104) ack 5 win 5792 <nop,nop,timestamp 59771 4312702>
00:57:04.544078 starship.30404 > bobsdell.60002: . ack 2904 win 32560 <nop,nop,timestamp 4312703 59771>
00:57:04.544797 starship.30404 > bobsdell.60002: F 5:5(0) ack 4008 win 33304 <nop,nop,timestamp 4312703 59771>
00:57:04.545243 bobsdell.60002 > starship.30404: F 4008:4008(0) ack 6 win 5792 <nop,nop,timestamp 59772 4312703>
00:57:04.545403 starship.30404 > bobsdell.60002: . ack 4009 win 33304 <nop,nop,timestamp 4312703 59772>
```

Appendix E

Static Reflector Pattern

This pattern was presented at the PLOP 2001 conference, Allerton, Illinois. It is structured according to the canonical form under the headings Name, Problem, Context, Forces, Solution, Resulting Context, Rationale, Examples and Related Patterns.

Somewhere near the bottom of the food chain of object oriented programming, the developer frequently encounters the rock face of a non object oriented API. This paper describes a specialisation of the Wrapper Facade [49] [47] pattern. Wrapper Facades encapsulate functions and data provided by existing non-object oriented API's. The Static Reflector addresses the particular problem of building wrappers which contain functions which take C function pointers as parameters. The pattern makes use of a static reflection method to facilitate the construction of cohesive, reusable framework classes which make use of such C functions. I show that the application of this pattern is surprisingly wide. Though concerned primarily with the interface between C and C++, the pattern has implications and applications to other languages as diverse as Java and [incr Tcl].

E.1 Name

Static Reflector

E.2 Problem

Many non-object oriented API's contain functions which arrange for another function (the target function) to be dispatched, perhaps in the context of a new thread or in the future, in response to an I/O, timer or user interface event. Building object oriented components on top of such API's is complicated by the fact that the target function must be statically declared. There is generally no way to directly specify a member function of an object instance to be the target of such an API function.

As a motivating problem, consider the problem of implementing a Java Thread class in a Java Virtual Machine written in C++ using the POSIX threads API. Java threads have a `start()` method which causes a new thread to be spawned to run, with its `run()` hook method as the thread entry point. In Java we would create and despatch the thread like this:

```
Thread t = new Thread();
t.start();
```

A starting point might be to collect together the POSIX threads functions (`pthread_create()` and family) into a cohesive Wrapper Facade[47]. It would then be convenient if we could build a C++ implementation as follows:

```
class Thread {
public:
   Thread() {}
   int  start()
     {
       // incorrect - run is not static
       return pthread_create(&tid, NULL,
                             run, NULL);
     }
protected:
   virtual void* run()
     {/* thread function */}

   pthread_t tid; //thread id
   /* ... other member data for
      the thread object */
};
```

The resulting `run()` method of Thread instances would have access to the member data of the instances. We could create new thread classes by inheriting from the base Thread and simply providing an overloaded `run()` method. Unfortunately, the call to `pthread_create()` is illegal as the third parameter refers to our `run()` method, which is not a valid static C function.

The above code can be modified to compile correctly by simply declaring the `run()` method as `static`, but this has a serious drawback. Static class functions have no direct access to the instance data of the object, nor can they benefit from inheritance and polymorphism.

E.3 Context

This problem recurs frequently in the context of building C++ classes around C functions which take C-style function pointers as parameters. Such functions are usually scheduling functions of some sort i.e. they request that another function be dispatched after some event occurs or in the context of a new thread.

Examples include:

The POSIX threads library The function `pthread_create()`[1], which is used for creating new threads, has an argument which specifies the entry function for the new thread. The function prototype is as follows:

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

The start_routine argument specifies the entry function.

The Win32 API This interface is rich in its use of function callbacks. The `SetWaitableTimer()` function, for example, has an optional argument to cause an Asynchronous Procedure Call (APC) to be queued when the timer expires. The argument specifies a pointer

[1]The Windows `beginthreadex()` function has a similar form and is also a candidate for static reflection

to a C function. An application places itself in an alertable state, such as in a call to `SleepEx()`, to receive notification of the event and dequeue and dispatch the APC.

The Tcl C library This library provides a number of useful functions for creating event-driven applications based around the Tcl Notifier[33]. A commonly used one is `Tcl_CreateFi` which has the form:

```
Tcl_CreateFileHandler(int fd, int mask,
            Tcl_FileProc proc,
            ClientData clientdata)
```

The proc argument is a pointer to a C function. Typically, the application waits for events in an infinite loop, blocking in calls to `Tcl_DoOneEvent()`.

Each of the above is characterized by having at least two arguments; one being a pointer to a C function and the other a general purpose argument which is passed through to the target function (the *void* arg* in pthread_create, the *ClientData clientdata* in `Tcl_CreateFileHandler`). The purpose of this argument is to pass data to the target function.

Such functions are commonly found in system APIs as well as in legacy C libraries. These functions are important in the implementation of components for use in extensible object oriented frameworks. Template and Hook methods[38] commonly form the metapatterns for such components, with the initiating method which makes the call to the C API scheduling function, being the template method. Hook methods are the application specific "hotspots" ie. the methods which are dispatched as a consequence of invoking the template method. They provide the application specific behavior. The problem-solution pair description below illustrates how the impedance mismatch between C and C++ frequently dictates the use of a third participant in this collaboration, the static reflector function.

E.4 Forces

- The developer needs to interact with a non object-oriented API for reasons of efficiency or fine grained functionality. The creation of Wrapper Facades i.e. clustering cohesive groups of functions into classes, is a proven good strategy[49] for dealing with

this interactiion. Functions which arrange the dispatch of other functions, such as those described above, present an implementation problem in creating classes based on Wrapper Facades because the target functions, in each case, must fall outside of the Wrapper Facade (or any other) class. The `run()` method of a thread class, for example, should form part of the cohesive cluster of functions which operate on thread objects.

- An important benefit of building object-oriented infrastructure on top of a non object-oriented API is the encapsulation of data with the methods which operate on that data. We would like the `run()` method in our thread example to have access to the thread instance variables such as `tid` in the example[2]. This behaviour would in fact be mandatory in an implementation of a Java Thread.

  Similarly, callback functions, such as file[3] and timer event handlers must frequently implement complex state machines. Having the callback as a member function hook with access to instance data has desirable consequences. It is these hooks which must act upon and alter the state data. Encapsulating the data with the methods which operate on it imposes some control and order on the resulting design.

E.5 Solution

The Static Reflection pattern resolves these forces by providing a mechanism for causing the dispatch of an object member function. It does this by introducing a static method to the collaboration, which is the intermediate target of the scheduler function. It makes use of the generic void* type argument provided by the scheduler functions to send, not explicit data to the target function, but a *reference* back to the originator of the scheduling call (a `this` pointer in C++). This collaboration is illustrated in Figure E.1.

Applying this pattern to our Java Thread class implementation yields a solution which resolves the problems encountered earlier:

[2]The advantages and disadvantages of implementing thread specific storage in this way compared to using the traditional thread specific storage interfaces have been described in (cite schmidt plopd 4)

[3]I use file in the very generic Unix sense of a file descriptor, which may actually refer to sockets, pipes or fifos

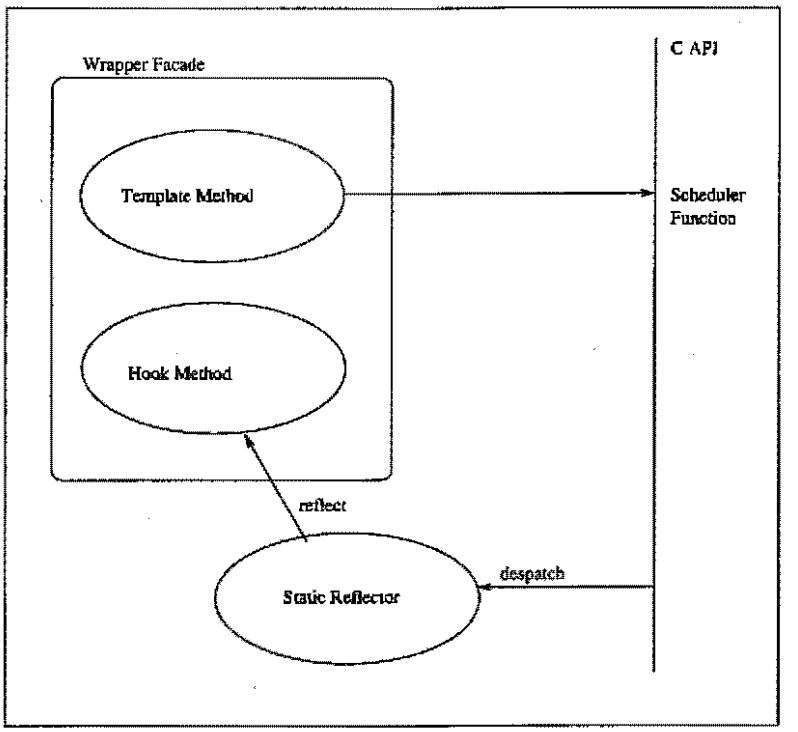


Figure E.1: Collaborations

```
class Thread {
public:
   Thread() {}
   int  start()
     {
       return pthread_create(&tid, NULL,
                             reflect, this);
     }
protected:
   inline static void reflect(void* id)
     {
        (Thread*)(id)->run();
     }

   virtual void* run()
     {/* thread function */}

   pthread_t tid; //thread id
   /* ... other member data for
      the thread object */
```

```
};
```

E.6 Resulting Context

An important consequence of the Static Reflection pattern is the ability to build framework objects through inheritance. By making the hook method virtual in the base class, derived classes need simply to provide an implementation of the hook method. The reflector in the base class will ensure that the hook is dynamically bound and despatched.

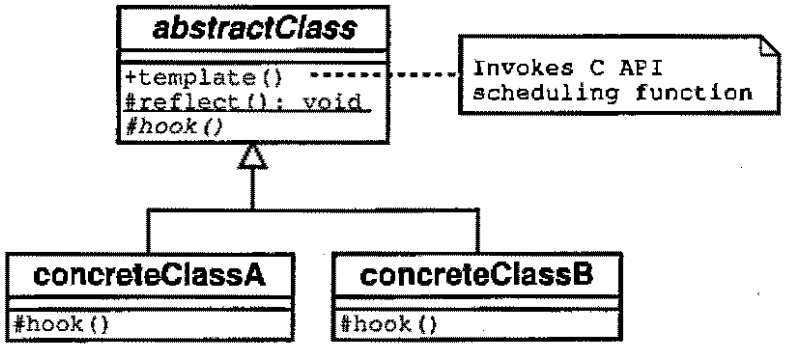


Figure E.2: Inheritance

A desirable consequence of placing the static reflector within the namespace of the class is that scoping can be used to make the hook method protected. Placing the reflector outside of the class would require either the hook method to be declared public or the reflector to be a friend.

A possible negative consequence of this pattern is the extra overhead involved with the double dispatch. This overhead is minimized by inlining the reflector method. Combined with the effects of compiler optimization the overhead should be negligible.

E.7 Rationale

The task of the static reflector method is to simply delegate to the hook method of the object which originated the message. The mechanism is similar to the double dispatch of

the Visitor Pattern of [8][4] By providing a reflector method to reflect the message back to the originating object the pattern solution effectively resolve the forces in the given context:

- Cohesion is achieved, because we can include our target function among the other Wrapper Facade functions which act upon the object.
- Access to encapsulated data is achieved, because the target function is a non-static member function of the class.

E.8 Examples

The Java thread example in the previous section is an example of a synchronous application of the Static Reflector pattern. The pattern is also used in this way in [47] to implement a threaded TCP service handler. Another example from the POSIX threads API is in the installation of thread exit handlers with `pthread_cleanup_push()`.

The pattern is more commonly seen in the context of asynchronous, event-driven scenarios. In this section I describe two such cases: one using C++ and the Win32 API and the other using IncrTcl and the underlying Tcl Notifier. Both cases describe the implementation of timer handlers. I/O event handlers and GUI event handlers can be constructed using the same pattern, but the code for timers is shorter.

E.8.1 Win32 APCs

The Win32 API provides a mechanism known as an Asynchronous Procedure Call (APC)[27]. Threads have an APC queue upon which APCs are queued when they are due to be scheduled. Threads need to be in an alertable state for the APCs to be dequeued and dispatched. APCs are typically used for timer and I/O event handlers and provide an alternative mechanism to the `WaitForMultipleObjects()` family of functions for demultiplexing and dispatching. APCs are a more explicit event delivery mechanism. `WaitForMultipleObjects()`, like the Unix `select()` and `poll` functions[55], lends itself more to a state-driven rather than event-driven design[4].

[4] In fact it is more like a triple dispatch, where the dispatching of the reflector is separated by space (a new thread context) or time (an event handler) from the originating call

In the example below, an APC is used to implement a timer handler. The template method in this class is the `start()` method and the hook method, `timedout()` is a pure virtual method. Note how the static reflector method, `reflect()`, and the `timedout()` hook are both protected. A concrete timer class, `myTimer`, is implemented by providing an implementation of the `timedout()` hook.

```
#include <windows.h>
#include <iostream>
#include <string>

class Timer {
public:
  Timer()
    {
      thndl = CreateWaitableTimer(NULL, FALSE, NULL);
    }
  void start(int fire, int repeat)
    {
      // scale everything up to milliseconds
      liDueTime.QuadPart=-fire*10000;
      interval = repeat;
      // arrange for Win32 APC to reflector
      SetWaitableTimer(thndl, &liDueTime, interval, \
                       Timer::reflect, this, FALSE);
    }
protected:
  // the static reflector function
  static VOID CALLBACK
  reflect(LPVOID self, \
          DWORD dwTimerLowValue, \
          DWORD dwTimerHighValue)
    {
      Timer* id = (Timer*)self;
      id->TimedOut();
    }

  virtual void TimedOut() = 0;
  HANDLE thndl;
  LARGE_INTEGER liDueTime;
  int interval;
};
```

```
class myTimer : public Timer {
public:
  myTimer(const string& name = "Anonymous")
    :myname(name) {}
  // the callback - with access to member data!
  void TimedOut()
    {
      cout << myname << " timed out" << endl;
    }
protected:
  string myname;
};

int main()
{
  myTimer T1("A Win32 alertable timer"),T2;
  cerr << "Starting timers ...\n";
  T1.start(3000, 3000);
  T2.start(4000, 3000);
  // A primitive event loop ...
  while(1) {
    SleepEx(INFINITE,TRUE);
  }
}
```

E.8.2 [incr Tcl]

Incremental Tcl [incr Tcl] is an object system for the Tcl language created by Michael J. McLennan of Lucent Technologies[25]. Being an interpreted language, the mechanics are considerably less sophisticated than C++. [incr Tcl] supports classes, scoping and inheritance, but has no notion of polymorphism and virtual methods. The underlying event demultiplexing and dispatching mechanism is based on the C language Tcl Notifier, which necessitates the application of the Static Reflector pattern to build notifiable, event driven objects. The form is slightly different from the previous examples, but the pattern is the same.

```
class Timer {
    # Note: the after command causes
```

```
    # the reflex scriptlet to be
    # evaluated at global scope after
    # the elapsed ms.
    # reflex thus plays the role of
    # the static reflector
    method schedule {ms} {
        set reflex "$this hook"
        after $ms $reflex
    }

    method hook {} {
        puts "Timer expired"
    }
}

class myTimer {
    inherit Timer

    method hook {} {
        # reschedule for 2 sec later
        schedule 2000
        puts "myTimer expired!!"
    }
}

myTimer t1
t1 schedule 2000

# wait forever in event loop
# t1's hook will be despatched
#      after 2 seconds
vwait 1
```

It may not be immediately clear how static reflection is being used here. The key point is that the semantics of the tcl *after* command determines that the argument script to *after* is evaluated at global scope. Notice how the template method (schedule) creates a string variable (reflex) which acts as the reflector to call back the hook method. In this case the hook method must be public because reflex is evaluated outside of the class namespace. We could have made the reflex script call back to a class wide procedure within Timer, which in turn called hook. This way the hook method could be declared protected, but at some cost.

There are many non object oriented APIs to which this this pattern can be meaningfully applied. One other such API the author is aware of is the Gtk toolkit, which is a C GUI framework used in the Gnome project[5]. Functions such as `gtk_signal_connect()` bind a C style function to a user-interface event. Static reflection is required to route such event handlers to object methods.

E.9 Exceptions and Variations

Not all scheduling type functions are candidates for static reflection. One notable exception is the installation and dispatch of signal handlers. The BSD `signal()` function and it's POSIX counterpart, `sigaction()`, specify a function to be dispatched in response to an operating system signal. Neither API function provides the facility for passing a *this* pointer, so static reflection cannot be used. [48] demonstrates how design patterns can be applied to the development of signal handling components.

The OpenGL GLUT library supports C-style callback functions for GUI events. These callbacks do not have the facility for passing a *this* pointer, so static reflection cannot be used. The author is aware of object oriented interfaces to OpenGL but not how they are implemented.

Variations on the static reflector pattern are commonly seen when the collaboration between template, reflector and hook methods transgress class boundaries. I have shown examples where all three are defined in a single class. There are cases where it may be desirable to more clearly separate the functionality of these three.

Creating pools of managed threads, for example, may suggest a design strategy where the template and reflector methods occur in a thread factory class, and the thread entry hook in a separate thread class. Similarly, one can separate an event handler class from the event dispatch and demultiplexing mechanism, as is done in the Reactor[51] pattern. The essence of the template-reflect-hook collaboration remains the same in each case.

[5]More information about gnome and gtk can be found at http://www.gnome.org/

E.10 Related Patterns

This pattern is closely related to the Wrapper Facade[49] pattern, which addresses the problem of building object oriented infrastructure on top of non object-oriented APIs. Whereas the Wrapper Facade deals with cohesive grouping of related existing API functions, the Static Reflector provides a mechanism for extending Wrapper Facades to include scheduled functions such as thread entry points and event callbacks.

In its application to the context of event callbacks, there is also some relationship with the Reactor pattern. The TkReactor implementation of the Reactor in the ACE toolkit makes use of static reflectors to dispatch timer and I/O handlers.

The Static Reflector makes use of the Hook/Template Method[38] pattern and something similar to a Double Dispatch[8].

Bibliography

[1] Christopher Alexander. *A Pattern Language: Towns/Buildings/Construction.* Oxford University Press, 1977.

[2] John H. Baldwin. Locking in the Multithreaded FreeBSD Kernel. In *BSDCon 2002, San Francisco, California*, pages 27–35, 2002.

[3] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *USENIX Annual Technical Conference*, November 1998.

[4] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, June 1999.

[5] Per Bothner. A gcc-based java implementation. In *IEEE Compcon 1997 Proceedings*, pages 174–178, February 1997.

[6] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993. ISBN 1-55615-481-X.

[7] Michael Franz. Emulating an Operating System on Top of Another. *Software - Practive and Experience*, 23(6):677–692, June 1993.

[8] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] Kurt Geihs and Christoph Gebauer. Load Monitor LM - ein CORBA-basiertes Werkzeug zur Lastbestimmung in heterogenen verteilten Systemen. In *MMB Freiberg*, pages 173–189, 1997.

[10] K John Gough. Stacking them up: a comparison of Virtual Machines. *Australian Computer Science Communications, IEEE Computer Society Press*, 23(4):55–61, January 2001.

[11] The WinSock Group. Windows sockets 2 specification, rev 2.2.2, August 1997. ftp://ftp.microsoft.com/bussys/winsock/winsock2/.

[12] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA '97*, pages 184–200, Atlanta, GA, October 1997. ACM.

[13] ISO/IEC. *International Standard: Programming Languages - C++, Number 14882:1998(E) in ASC X3*. American National Standards Institute, 1998.

[14] Douglas C. Schmidt James C. Hu, Irfan Pyarali. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *2nd Global Internet mini-conference*, November 1997.

[15] David A. Patterson John L. Hennessey. *Fundamentals of Computer Design*. Morgan Kaufmann Publishers (Inc), 1990.

[16] Bob Jolliffe. Static Reflection Pattern. In *PLOP2001 (Pattern Language of Programs), Illinois*, September 2001.

[17] Poul-Henning Kamp. Rethinking /dev and devices in the unix kernel. In *BSDCon 2002, San Francisco, California*, pages 77–88, 2002.

[18] David G. Korn. Porting UNIX to Windows NT. In *Proceedings of the 1997 USENIX Conference*, pages 43–57. USENIX, 1997.

[19] David G. Korn. UWIN — UNIX for Windows. In *Proceedings of the USENIX Windows NT Workshop*, pages 133–145. USENIX, 1997.

[20] Greg Lehey. *Porting UNIX Software*. O'Reilly and Associates,Inc, 1995.

[21] Don Libes. Automation and testing of character-graphic programs. *Software Practice and Experience*, 27(2):123–137, 1997.

[22] T Lucey. *Quantitative Techniques.* DP Publications Ltd, Aldine Place, 142-144 Uxbridge Rd, London W12 8AA, 4th edition, 1993.

[23] M. R. Macedonia and D. P. Brutzman. Mbone provides audio and video across the internet. *IEEE Computer*, 27(4):30–36, April 1994.

[24] McKusick, Bostic, Karels, and Quarterman. *The Design and Inmplementation of 4.4 BSD Operating System.* Addison Wesley, 1996.

[25] M. McLennan. [incr Tcl]: Object-Oriented Programming with Tcl, 1993.

[26] Microsoft Corporation. *Microsoft Developer Network (MSDN) Library*, October 1999.

[27] Microsoft Corporation. *Microsoft Developer Network (MSDN) Library*, April 2000.

[28] Microsoft Corporation. *.NET Framework Developer's Guide, Microsoft Developer Network (MSDN) Library*, July 2002.

[29] Jeffrey Mogul. Brittle metrics in operating systems research. In *7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 90–95, March 1999. Rio Rico, AZ.

[30] Patrick Naughton. *The Java Handbook.* McGraw-Hill, 1996.

[31] Geoffery J Noer. Cygwin32: A free win32 porting layer for unix applications. In *2nd Usenix Windows NT Symposium*, 1997.

[32] The Open Group (X/Open & OSF). *Systems Management: Universal Measurement Architecture. CAE Specification C427*, January 1997.

[33] John K. Ousterhoudt. *Tcl and The Tk Toolkit.* Addison-Wesley, 1994.

[34] Frank Pilhofer. Combat. a CORBA language mapping for Tcl. In *EuroTcl Conference*, 2000.

[35] J. Postel. User Datagram Protocol. RFC768, August 1980.

[36] J. Postel. Internet Protocol. RFC791, September 1981.

[37] J. Postel. Transmission Control Protocol. RFC793, September 1981.

[38] Wolfgang Pree. *Design Patterns for Object Oriented Software Development.* ACM Press, Addison Wesley, 1995.

[39] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. Technical Report CITI00-4, Center fo Information Technology Integration, University of Michigan, May 2000.

[40] Bob Quinn and Dave Shute. *Windows Sockets Network Programming.* Addison-Wesley, 1 edition, 1995.

[41] Dennis Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8 part 2), October 1984.

[42] D.M. Ritchie and K. Thomson. The unix time-sharing system. *Communications of the ACM*, 17(7), July 1974.

[43] Kay A Robbins and Steven Robbins. *Practical Unix Programming.* Prentice Hall, 1st edition, 1996.

[44] Herbert Schildt. *Windows 95 Programming in C and C++.* McGraw-Hill, first edition, 1995.

[45] Douglas Schmidt. Experience using design patterns to develop reuseable object-oriented communication software. *Communications of the ACM, Special Issue on Object-Oriented Experiences*, October 1995.

[46] Douglas Schmidt and Irfan Pyarali. Strategies for implementing condition variables on win32. *C++ Report, SIGS*, 10(5), June 1998.

[47] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

[48] Douglas C. Schmidt. "Applying Design Patterns to Simplify Signal Handling". *C++ Report, SIGS*, 9(6), June 1997.