University of Newcastle upon Tyne

Dept. of Computing Science

UNIX Communication Facilities

Gerhard Müller

2nd April 1996

Individual Project – CSC 399

Supervisor: Dr. N.A. Speirs

I wish to thank especially my Supervisor, Dr. A. Speirs, and the Free Software Foundation; without them my work would have been much more difficult.

Abstract:

This document investigates the different mechanisms offered for interprocess and interprocessor communications by the UNIX operating system.

Some important background information is given to allow a better understanding of the UNIX communication facilities. This includes the history of the UNIX operating system, some important standards, and the idea of the client/server model.

Some communication facilities are only available between processes which are executed on the same computer. The usage of signals, pipes, FIFOs, stream pipes, message queues, semaphores, and shared memory is described.

Communication between processes on different computers includes knowledge about networking. Therefore the principles of networking are summarized and one protocol suite, TCP/IP, examined. With this knowledge the UNIX communication facilities for processes on different computers are described: the BSD socket interface and the System V Transport Layer Interface.

The knowledge gained is then used to measure the performance of all described communication facilities. Measurements on three different computer systems are done to allow general conclusions. It is found that buffer sizes over 8 Kb do not normally result in faster data exchange. Shared memory was found to be the fastest local interprocess communication facility, while the User Datagram Protocol (UDP) provides the fastest information exchange between processes on different computers.

While doing the performance measurements the unreliability of the UDP protocol was noticed. The overhead of making communication reliable in spite of an unreliable communication protocol was therefore investigated and a possible solution is described.

This document is also available on-line through the World Wide Web under the URL `http://www.ncl.ac.uk/~n5820119/project/ip/ip.html`.

Contents

Abstract iii

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Outline . . . 2

2 Analysis & Design 4

2.1 Analysis . . . 4

2.2 Design . . . 9

2.3 Summary . . . 16

3 Local IPC 17

3.1 Signals . . . 18

3.2 Pipes . . . 21

3.3 FIFOs (named pipes) . . . 23

3.4 Stream Pipes . . . 23

3.5 System V IPC . . . 24

3.6 Summary . . . 28

4 Networking Principles 29

4.1 Switching Techniques . . . 29

4.2 Message Passing Concepts . . . 31

4.3 Addressing Issues . . . 31

4.4 Data Representation . . . 33

4.5 Protocols . . . 35

4.6 The OSI Seven Layer Reference Model . . . 36

4.7 Protocol Suites . . . 39

4.8 Summary . . . 40

5 Networking with TCP/IP 41

5.1 Overview . . . 41

5.2 TCP/IP compared to the OSI protocol stack . . . 42

5.3 History and Goals of TCP/IP . . . 44

5.4 Internet Addresses . . . 44

5.5 Port Numbers . . . 46

5.6 Multiplexing/Demultiplexing 46
5.7 Internet Protocol 47
5.8 User Datagram Protocol 49
5.9 Transmission Control Protocol 49
5.10 Reliability 51
5.11 Application Level Services 52
5.12 UNIX Network Tools 53
5.13 Summary 53

6 Distributed IPC **55**

6.1 Sockets 55
6.2 Transport Layer Interface 63
6.3 Remote Procedure Calls 66
6.4 Summary 68

7 Performance **70**

7.1 Measuring Time 72
7.2 Principle Overhead of a System Call 73
7.3 Signals 74
7.4 Pipes 74
7.5 FIFOs (named pipes) 76
7.6 Message Queues 76
7.7 Semaphores 77
7.8 Shared Memory 78
7.9 Sockets 80
7.10 Transport Layer Interface 85
7.11 Conclusion 85
7.12 Summary 88

8 Making Unreliable Communication Reliable **90**

8.1 Lost Messages 91
8.2 Corrupt Messages 92
8.3 Duplicate Messages 92
8.4 Messages out of Order 92
8.5 Congestion Control 92
8.6 Fundamental Problem of uncertain Communication 93
8.7 Implementation 93
8.8 Summary 93

9 Conclusion **94**

9.1 Results 94

9.2 Additional IPC Facilities 94

9.3 Possible Improvements 96

9.4 The Future of UNIX Communication Facilities 97

9.5 A Final Word 99

10 References **100**

A Complete Source code **105**

A.1 Makefiles 107

A.2 Commonly used Subroutines 111

A.3 Local IPC 167

A.4 Distributed IPC 190

A.5 Performance 234

A.6 Program Environment 286

A.7 Remote Procedure Calls 306

B Additional Information Resources **309**

B.1 Internet Request For Comments 309

B.2 Periodicals 309

B.3 Additional Web Pages 309

C Glossary **312**

D Used UNIX System & Library Calls **321**

E About this Document **326**

List of Figures

1 UNIX History & Directions . . . 7
2 Principle Methods of Communication . . . 11
3 Client/Server Model . . . 13
4 Typical Scenario of Client/Server Interaction . . . 14
5 The Principle of most local IPC Facilities . . . 18
6 Execution of a Signal Handler . . . 19
7 The Result of the `pipe()` System Call . . . 21
8 A Pipe after a `fork()` System Call . . . 22
9 Resulting Pipe . . . 22
10 The Principle of Message Switching . . . 30
11 Layered Naming Contexts . . . 32
12 Little vs. Big Endian with 0xABCD as Example Number . . . 34
13 A simple Protocol Stack . . . 36
14 Adding Protocol Control Information in each Level . . . 37
15 The ISO-OSI 7 Layer Reference Model . . . 37
16 Communication between OSI Layers . . . 39
17 The Components of the Internet Protocol Suite . . . 42
18 Encapsulation of Data as it goes down the Protocol Stack . . . 43
19 OSI Model versus TCP/IP Protocol . . . 43
20 Internet Address Classes . . . 45
21 The UNIX Port Model . . . 47
22 The Demultiplexing of a received Ethernet Frame . . . 48
23 IP Datagram, showing the Fields in the IP Header . . . 48
24 UDP Datagram, showing the Fields in the UDP Header . . . 49
25 TCP Segment, showing the Fields in the TCP Header . . . 51
26 Socket System Calls, Connection-Oriented Protocol . . . 59
27 Socket System Calls, Connectionless Protocol . . . 60
28 How Ports, Addresses, and Sockets fit together . . . 64
29 TLI Function Calls, Connection-Oriented Protocol . . . 65
30 TLI Function Calls, Connectionless Protocol . . . 66
31 Calls and Messages in an RPC. Each ellipse represents a single process, with the shaded portion being the stub. . . . 67
32 Throughput of Pipes . . . 75
33 Time per `read()` System Call for Pipes . . . 75
34 Throughput of FIFOs . . . 76

35 Time per `read()` System Call for FIFOs 77

36 Throughput of Messages 77

37 Time per `msgsnd()` System Call 78

38 Time needed to create, attach, detach and remove a Shared Memory Region 79

39 Throughput of local Sockets created by the `socketpair()` System Call 80

40 Time for `read()` System Calls for local Sockets 81

41 Throughput of UDP on one computer 82

42 UDP Throughput between different Computers 82

43 Time per `sendto()` and `recvfrom()` System Calls (both executed in client and server process) on one Computer 83

44 Time per `sendto()` and `recvfrom()` System Calls (both executed in client and server process) between different Computers 83

45 Throughput of a local TCP Connection 84

46 Throughput of a TCP Connection between different Computers 85

47 Time for `read()` System Calls of a TCP Connection on one Computer 86

48 Time for `read()` System Calls of a TCP Connection between different Computers 86

49 A STREAM with one processing Module 96

50 The basic IPng Header 98

List of Tables

1 POSIX Standards 8
2 Summary of the major Differences between connection-oriented and connectionless Service 12
3 POSIX Signals 20
4 Signal Functions 20
5 Summary of System V IPC System Calls 24
6 System Calls for Handling Message Queues 26
7 Maximum Number of Message Queues per Process 26
8 System Calls for Handling Semaphore Sets 27
9 Maximum Number of Semaphore Sets per Process 27
10 System Calls for Handling Shared Memory Regions 27
11 Maximum Number of Shared Memory Regions per Process 28
12 The Ranges of IP Address Classes 45
13 UNIX Network Tools 54
14 Comparison of Protocol Features for IP, UDP, and TCP 54
15 Library Calls for Database Services 62
16 Library Calls for converting from/to Network Byte Order 62
17 Internet Address Manipulation Library Routines 63
18 System Calls which fill in Parts of an Association (Internet domain 5-tuple assumed) 63
19 Comparison of Socket, TLI, Message Queues, and FIFO Function Calls 69
20 Principle Overhead of a System Call 73
21 Performance of Signals on different UNIX Systems 74
22 Semaphore Operations 78
23 Time needed to write 1 Mb 79
24 Average Overhead of Establishing a TCP Connection on one Computer 84
25 IPC Facility to use for different Amount of Data 88
26 IPC Facilities for different Types of Connections 88
27 System Calls for Memory Mapped I/O 95
28 System Calls for STREAMs 95
29 Overview of Source Code Directories 105
30 Makefiles 107
31 Purpose of Files in the `common` Directory 112
32 Purpose of Files in the `local` Directory 167
33 Purpose of Files in the `distSOCKET` Directory 190

34 Purpose of Files in the `distTLI` Directory 190

35 Purpose of Files in the `performance` Directory 235

36 Purpose of Files in the `environment` Directory 286

37 Purpose of Files in the `rpc` Directory 306

38 UNIX Signal Handling Functions 321

39 UNIX System Calls for Local IPC 322

40 UNIX System V IPC Specific System Calls 322

41 UNIX System & Library Calls for BSD's Socket API 323

42 UNIX Library Calls for the System V TLI 324

43 UNIX Timing System & Library Calls 324

44 UNIX String Library Calls 325

45 UNIX Process Related Library and System Calls 325

46 Diverse UNIX System and Library Calls 325

1 Introduction

This chapter

- **describes the goals of this Individual Project**
- **gives a brief background to the subject area**
- **states the contents of the following chapters**

The UNIX operating system is one of the older of the widely used operating systems, with a history of more than 30 years. UNIX has changed over the time, and many of its most distinctive and useful facilities are relatively recent developments [Leff88]. The design of the UNIX operating system had a major impact on nearly all operating systems developed later.

UNIX supports multiple processes to be (virtually) executed at the same time. Processes "live" and sometimes need to communicate to exchange data or to synchronize their work. Originally UNIX had very poor "interprocess" communication facilities. Today many different approaches are available. Some are restricted to work only for processes which are executed on the same computer, and some make it possible for communication between processes on different computers via a network.

The aim of this Individual Project is to

- investigate the different interprocess communication facilities at the system programming level
- explain networking to the degree which is necessary to understand the interprocess-communication facilities between different computers
- measure and compare the performance of the different interprocess communication facilities
- discuss the overheads that are needed to make communication reliable if only an unreliable communication service is available.

It is assumed that the reader has a good knowledge of C/C++, as no programming language features are described. If this is not the case, [Strou91] is a good starting point for learning C++.

The reader should also be familiar with the UNIX operating system and normal UNIX programming. This does not include network programming. This document does not describe how to invoke a C/C++ compiler, how to use '`make`' or other UNIX standard commands.

1.1 Outline

There now follows an outline of the purpose of subsequent chapters.

Chapter 2 summarizes what was done to achieve the aims stated above, discusses some design decisions (with reasons why certain decisions were taken) and gives important background information. This includes: the history of the UNIX operating system; important UNIX standards; communication principles; the principles of client/server communication and the aim of interprocess communication.

Chapter 3 describes the interprocess communication (IPC) facilities which are only available if the processes which want to communicate run on the same computer.

Communication between processes on different computers adds additional complexity. Therefore the principles of networking in general are described in Chapter 4.

As theory is nice, but useless without practice, the successful TCP/IP protocol suite is summarized in Chapter 5 as an example of an existing network protocol.

This information is then used in Chapter 6 to cover two different Application Protocol Interfaces (APIs) for networking: BSD sockets and the System V Transport Layer Interface (TLI). These two interfaces are used to demonstrate how to write client and server applications that use the TCP/IP protocol suite.

In Chapters 3 and 6 example programs demonstrate how to use the described interprocess communication facilities.

To choose the right IPC facility for a specific purpose the performance is often important. Therefore the performance of all described IPC facilities are measured, compared and discussed in Chapter 7.

The overheads of making communication reliable in spite of using an unreliable communication service are discussed in Chapter 8. Functions are developed which use an unreliable protocol (e.g. UDP), but enable reliable communication between two or more processes.

Chapter 9 covers the conclusions of this Individual Project, describes shortcomings, possible improvements and gives a short overview about new developments.

The annotated references chapter Chapter 10 contains the books and other information sources that were used to write this document.

The Appendix contains

- the complete source code for all programming examples
- the programs to measure the performance of the different IPC facilities (the WWW-version of this document also contains the results of all performance measurements)
- a collection of additional information resources
- a glossary of used terms
- a short description of all used UNIX system and library calls
- a list of all programs used to create this document.

2 Analysis & Design

This chapter

- **describes how the problem was analysed**
- **describes which fundamental approaches were considered**
- **states the limitations of this project**
- **covers principal design issues**
- **gives overview of important issues:**
  1. **History of UNIX**
  2. **Communication Principles**
  3. **the Client/Server Model**
  4. **Interprocess Communication**

2.1 Analysis

UNIX programming and networking is a very wide field. Therefore before it was decided what to cover in this document a lot of books and articles were read. It turned out that only certain aspects of the UNIX communication facilities could be covered due to the limited amount of time and space.

This individual project therefore concentrates on analysing the traditional UNIX system programming level. Higher level applications and protocols are only mentioned (if at all), but not covered in detail.

The different versions of UNIX complicated the programming. The availability of certain UNIX features differed quite a lot. To understand the differences and future directions of the UNIX operating system, an overview about existing and future standards, together with an overview of UNIX history are given later in this chapter.

The description of local IPC features was the easiest part. As networking adds a lot of complexity it was decided to cover the principles of networking. Without this knowledge it is not possible to understand how "distributed" IPC facilities fit into the overall picture. With the OSI protocol implementations declining in popularity ([Salus95]) it was decided to investigate only the TCP/IP protocol suite. This suite offers a working solution for today's most important networking demands. TCP/IP is used in the

Internet, and, particularly, in the UNIX environment it is *the* standard.

The performance measurements were done on three different computer systems. This was done to be able to make more general assumptions about the performance of certain IPC facilities.

Some techniques and feature developments deserve to be mentioned in a document that has the title "UNIX Communication Facilities". Therefore IPng and OSF DCE are summarized in the conclusion chapter.

The chosen programming language for all example programs was C/C++. UNIX has a strong connection to C: UNIX is written in C, to be precise, C was developed to fill the needs for a powerful low-level general purpose language for making the UNIX operating system portable. C is therefore the 'natural' language to program UNIX systems.

Today C++ is the industry choice to write large software systems. C++ is a hybrid language, it combines object-orientation and declarative programming so that C code is still valid C++ code. All examples therefore use the benefits of C++, but use the C interface to make UNIX system calls. No attempt was made to hide the use of the UNIX system call interface, as it was one goal of the individual Project to explore it. In the real world certain software classes would be used to hide some of the operating system details from the programmer. Two of them are mentioned in Chapter 9.4.2.

This document only addresses certain aspects of "UNIX Communication Facilities". Some subjects had to be omitted due to space or time (or both), although they are important and interesting areas. These are some limitations of the project:

- it is not a introduction to UNIX, UNIX programming, C or C++
- the example programs are important to understand, but can't be covered in detail. Please refer to [Stev90], [Stev92], [GNU C] or other similar documents if questions arise why something was programmed in a particular way.
- The network description is only an overview, for more detail about networking in general please refer to [Tan88] or [Com95] and [Com94] for the TCP/IP protocol suite.
- security aspects are omitted completely

- the physical transportation of data (how Ethernet works etc.) and the transport of data via other computers is not covered.
- This project concentrates on the view that applications have on the IPC facilities. The implementation of these facilities is not investigated.

2.1.1 History of UNIX

The UNIX interprocess facilities can only be understood with the historical background.

UNIX was originally developed by Ken Thomson (and later Dennis Ritchie) at Bell Labs (AT&T) to get a "leaner" MULTIX operating system. Originally UNIX was written in assembly language on a PDP-7 minicomputer. Later (1973) UNIX was rewritten in C, which was developed to make UNIX highly portable. Design principles were that it should be as simple and elegant as possible (no different file types etc.). Therefore only pipes were used/developed as interprocess communication facilities. UNIX was first used for word processing in AT&T Bell Laboratories. Bell Labs were asked for a copy of UNIX from many Universities. As AT&T was a regulated monopoly at this time, it was not permitted to be in the computer business, and therefore Bell Labs gave the source code away for only a small fee.

The University of California in Berkeley became a major center of UNIX development. The Computer Systems Research Group (CSRG) at the University of California in Berkeley was partly sponsored by DARPA, the Defense Advanced Research Projects Agency. Berkeley's first release of an enhanced UNIX was 1BSD (First Berkeley Software Distribution). 3BSD added virtual memory, while 4.2BSD added interprocess communication facilities (sockets) that made networking possible. BSD used TCP/IP as network protocol, which became a *de facto* standard. The 4.4BSD release (1993) was the last release from CSRG for reasons stated in [Bost93].

AT&T was split up in several companies in 1984. The new AT&T was allowed to enter the computer market, and it did. Shortly thereafter, AT&T released its commercial version of UNIX, System III, which was replaced by System V a year later. The original System V was further developed into releases 2, 3, and 4, which got more and more powerful, complicated and bigger. System V added message queues, semaphores, and shared memory to the interprocess communication facilities. AT&T recently sold UNIX System Laboratories (USL), the subsidiary which developed the AT&T version of UNIX, together with all interests in UNIX to Novell.

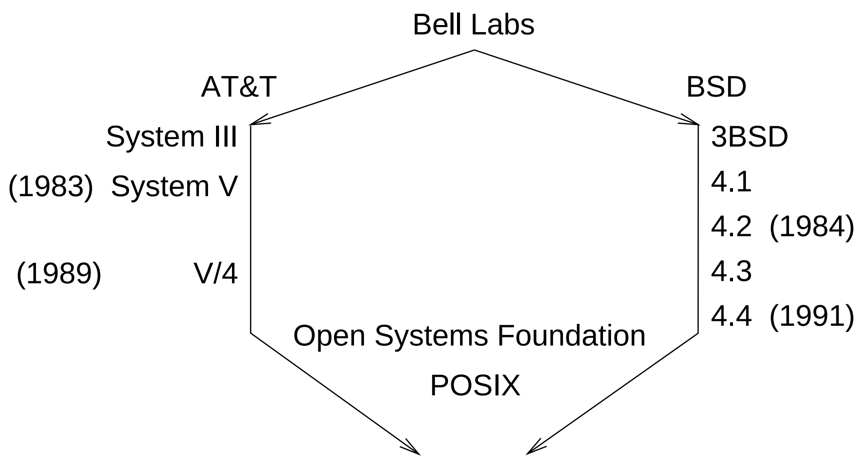

Figure 1: UNIX History & Directions

Most UNIX systems are based either on the AT&T version of UNIX (e.g. HP-UX from Hewlett-Packard, AIX from IBM), or Berkeley UNIX (e.g. Solaris (former SunOS) from SUN or Ultrix from DEC). Both derivatives have added the more important features of the other UNIX direction. Figure 1 shows the development of the two major UNIX directions.

2.1.2 UNIX Standards

As a result of the above there were two different and quite incompatible versions of UNIX around in the late 1980s. Some companies added additional features to their version of UNIX. This produced a need for standards. All initial attempts failed. AT&T issued the *System V Interface Definition* (SVID) (various versions for different UNIX versions, e.g. SVID Issue 3 in 1989 for System V Release 4 (SVR4)) as a standard to keep all the System V vendors "in line", but this standard had no effect on vendors which based their version of UNIX on BSD's UNIX. This split of UNIX is still noticeable and makes it complicated to write UNIX programs which should be portable between these two UNIX directions.

A more serious attempt to reconcile the two UNIX directions was initiated by the "neutral" IEEE Standard Board. The *POSIX* project with its standards is highly accepted in the industry. A well known standard is 1003.1 (sometimes also called POSIX.1), which is only one part of the POSIX standards (see Table 1).

POSIX is still working on standardizing UNIX, but its work has also influenced other

Standard	Description
1003.0	Guide and overview
1003.1	Library Functions (i.e., system calls)
1003.2	Shell and utilities
1003.3	Test methods and conformance
1003.4	Real time extensions
1003.5	Ada language bindings
1003.6	Security extensions
1003.7	System administration
1003.8	Transparent file access
1003.9	Fortran 77 bindings
1003.10	Supercomputing

Table 1: POSIX Standards

operating systems. Most modern operating systems try to be POSIX compliant.

Another big problem was a split of vendors. A group of vendors including IBM, DEC, and Hewlett-Packard did not like the idea that AT&T had control over the "official" UNIX. They set up *Open Software Foundation* (OSF), which has the aim to produce a system that conforms to all IEEE standards, but also contains a large number of enhancements like a window system (X11), a standardized graphical user interface (MOTIF). OSF also develops DCE (Distributed Computing Environment), which "provides services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment" [OSF DCE, page 1]. The reaction of AT&T was to set up *UNIX International* (UI) as a vendor-independent controlling body for the future of UNIX, which had similar aims. For years there has been a fight between UI and OSF. As stated above AT&T sold UNIX to Novell recently. With the threat of Windows NT from Microsoft the time for fights between OSF and UI is over; today UI has no significant industry relevance any more.

X/Open (a consortium of industry representatives with a common charter for developing open system specifications) has three goals:

1. support portability of applications at the source code level

2. enhance application connectivity via a set of portable transport-independent network services, plus support for common protocol stacks

3. define and promote consistent user interfaces

X/Open concentrates on the needs of end users. This makes X/Open more vendor neutral than most other organizations.

The Object Management Group (OMG) has developed a set of standards for the use of objects. The most known is the Common Object Request Broker Architecture (COBRA).

Another standard influenced UNIX: the standardization of C, known as ANSI C.

A detailed description of different flavours of the UNIX Operating System can be found in [Pab93, page 3-13]. A more detailed description of the UNIX history can be found in [Tan92]. A more detailed description of standards and industry organizations can be found in [Tan92] and [Elb94]. Some networking standards are mentioned in Chapter 4.7.

2.2 Design

Chapters 3 (Local IPC) and 6 (Distributed IPC) contain example programs for each IPC facility. The programs were designed to be reasonably short so that they can be understood without referring to other functions. Due to space limitations the source code itself is not included or described in the chapters which give the overview of the different IPC facilities. Nevertheless the examples are an important part of understanding a particular IPC facility and therefore this Individual Project.

The principles of communication and the client/server model are important for understanding interprocess communication facilities. Therefore they are described in detail later in this chapter. Most examples in this document use the client/server concept.

The programs which measure the performance of the different IPC facilities in Chapter 7 are normally very similar to the ones used as an example. This aids understanding of the performance measurement programs.

All programs try to conform to the POSIX.1 standard as far as possible. This is, unfortunately, not easy and sometimes even impossible. This is due to the fact that networking and some forms of local IPC are not covered by POSIX standards (yet) and the systems available may not offer specific features.

All developed functions, classes, and programs were intensively tested. Developed functions and classes were first tested individually, before they were used by more complex programs where possible. Debug code that prints internal states of interesting variables, while the program was being executed, was also used. Both the test functions and the debug code were removed to keep the program source shorter and to make them easier to read. These test programs are also not included in the appendix due to limited space. The test code for one function is preserved to demonstrate how test code looked like (see `local/namedstreampipe.c` (page 175)).

2.2.1 Communication Principles

The principle methods of communication between different *endpoints of communication* are

1. *One-to-One* communication, also called *point-to-point* communication. Data is exchanged between two endpoints.

2. *One-to-Many* communication, also called *broadcast*. One endpoint sends data to more than one other endpoint.

3. *Many-to-One* communication, also called *gathering*. One endpoint gets data from more than one endpoint.

4. *Many-to-Many* communication. Multiple endpoints exchange data with multiple endpoints.

Only the first form of communication is described in this document. The second one is normally only supported by local network types like Ethernet and therefore not discussed further. If methods 2 to 4 are needed they normally have to be implemented in software using point-to-point communication. Some problems in implementing these methods are to ensure that the communication is reliable (all endpoints get the information or none of them), and that the data is delivered to all receivers in identical order. These problems are beyond the scope of this document.

One-to-one communication can be classified further by the type of connection. Communication is termed *connection-oriented* if both communication endpoints establish a (logical, virtual) connection before the actual data exchange can take place. A connection-oriented exchange involves three basic steps: connection establishment; data transfer; and connection termination. An example for connection-oriented communication is the use of telephones, where one has to dial before it is possible to speak

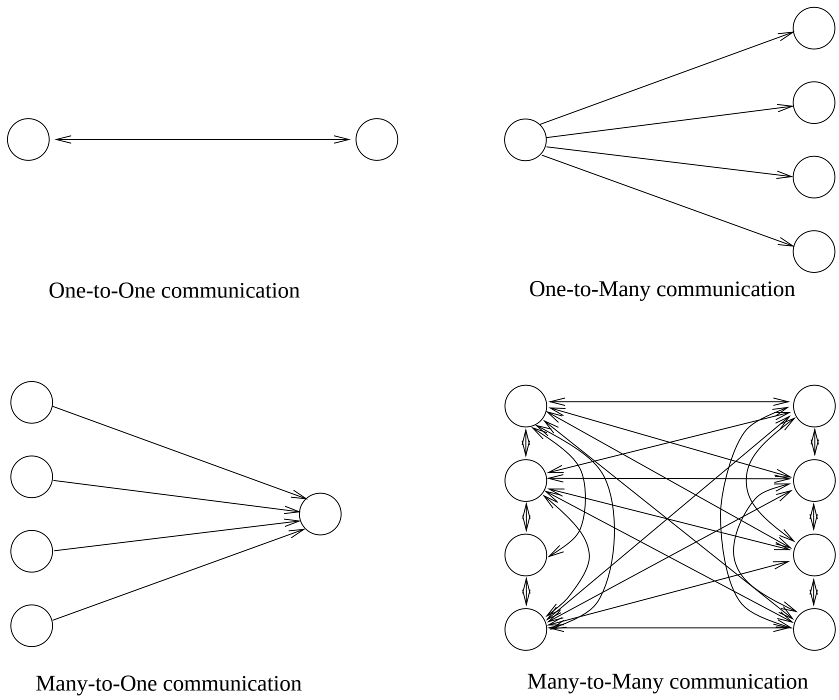

Figure 2: Principle Methods of Communication

with the person on the other end.

Connectionless communication uses messages (often called *datagrams*) to transmit information. These datagrams are transmitted independently. Therefore each datagram has to contain all information to reach the destination. Datagrams normally have a fixed length, can arrive in a different order than they were sent, and can get lost. Communicating with someone else using letters is a good example for connectionless communication. Table 2 [Tan88] summarizes the major differences between connection-oriented and connectionless services.

Connection-oriented communication can be classified further by how information can "flow". With the *simplex communication* method data can only flow from one endpoint to another endpoint in one direction. If the direction of data can be changed, but only one endpoint can send data at a time, communication is called *half duplex*. *Full duplex* or *bi-directional* communication allows flow of data in both directions at the same time.

Communication can be *blocking* or *nonblocking*. A send or receive operation is called blocking if the operation does not return till the operation is completed. Nonblocking operations return immediately.

Issue	**Connection-oriented**	**Connectionless**
Initial Setup	Required	Not possible
Destination address	only needed during setup	Needed on every packet
Packet sequencing	Guaranteed	Not guaranteed
Error Control	Done by network layer	Done by transport layer
Flow Control	Provided by network layer	Not provided by network layer
Are connection identifiers used?	Yes	No

Table 2: Summary of the major Differences between connection-oriented and connectionless Service

2.2.2 The Client/Server Model

Comprehension of the client/server model is very important to help understand UNIX communication facilities. Therefore the idea of the client/server model, and how this model is normally used in UNIX, is now covered.

In the client/server model two or more processes, eventually running on different computers, work together to provide a specific service. The communication scheme is normally like this: one process (called the client) sends a *request* (it requests a service) to other processes (called *servers*). These servers send a *reply* back to the client. These interactions are illustrated by Figure 3.

Two different kinds of servers are distinguished: so-called *concurrent* servers that can handle multiple requests at the same time, and so-called *iterative* servers which handle one request after the other. A typical example of an iterative server is a server that replies with the time on the computer where the server is running.

Concurrent servers are normally used if an answer can not be sent back immediately, or if there is additional communication with the client. Often concurrent servers use connection-oriented protocols. Concurrent servers in UNIX often spawn other processes via `fork()` to handle a request. Example: most servers that deal with the transfer of files are implemented in the concurrent fashion, as the amount of processing required depends on the file size.

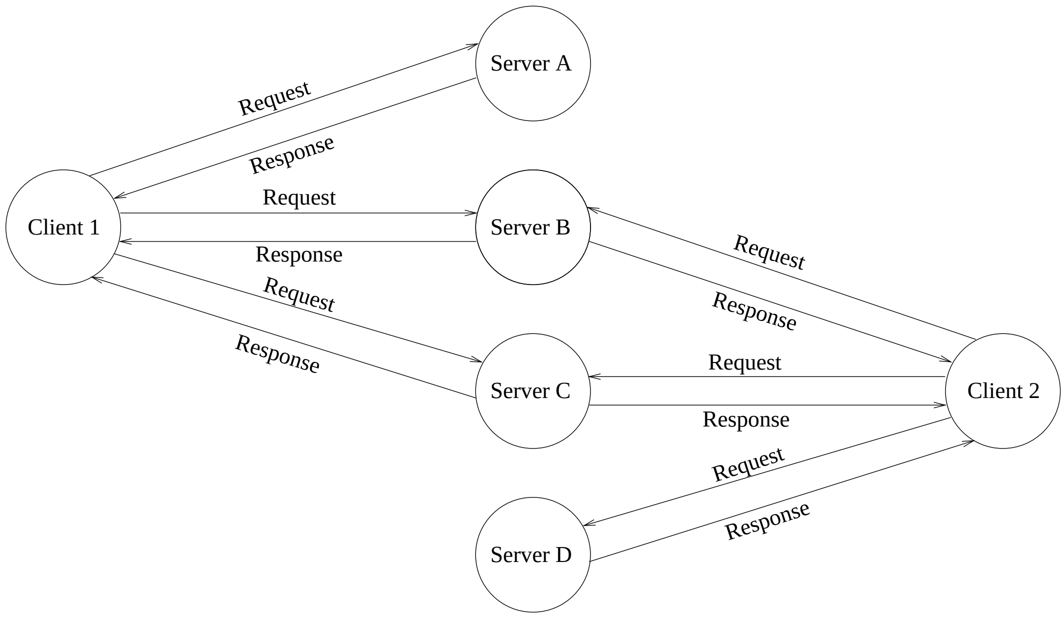

Figure 3: Client/Server Model

Iterative servers are ideal if requests can be handled very quickly and the replies are short. Normally connectionless protocols are used for iterative servers.

A client in the UNIX environment is normally the part of an application that is run by a user. The servers are normally *daemons* (servers are normally called this in the UNIX world) that are either started automatically on system boot, or when a request for this service arrives (by the `inetd` daemon). Once started, daemons normally stay in memory till the computer is rebooted (they do not die). Note that most daemons spend most of their time waiting for a request to handle.

It is possible that the client and server both run on one computer, but this is not necessary if they use network communication facilities to communicate.

Other definitions of client/server computing can be found in [Stall95, page 510]

Most examples in this project try to implement the client/server model, using different communication facilities.

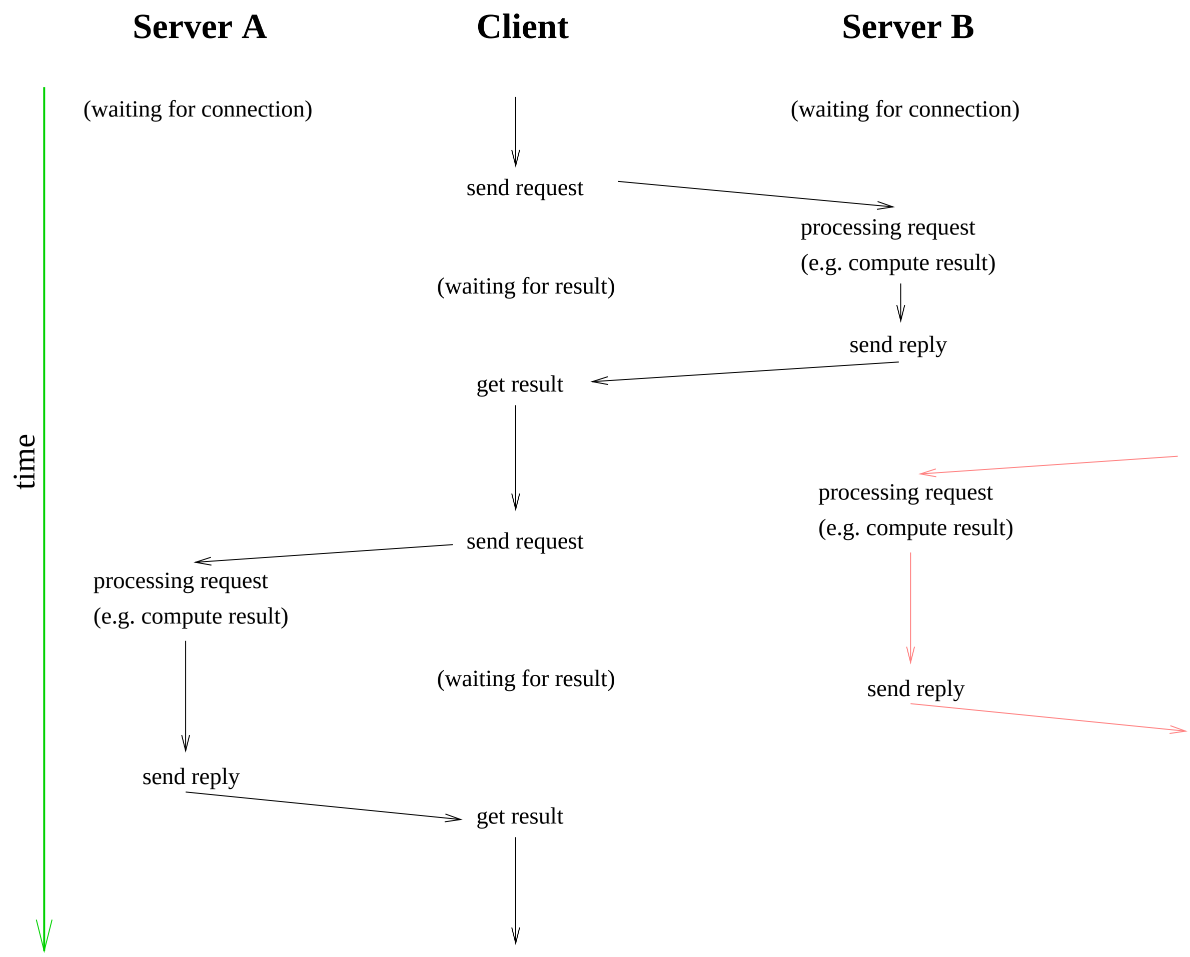

Figure 4: Typical Scenario of Client/Server Interaction

2.2.3 Interprocess Communication

Interprocess communication allows different processes to communicate via a small and comprehensive set of interaction possibilities between process. If interprocess communication mechanisms are used exclusively for the interaction between different processes, they force a clean and simple interface between them [Mull93].

Interprocess communication mechanisms serve four important functions [Mull93]:

1. they allow communication between separate processes (eventually over a network)
2. they provide *firewalls* against failures, and provide the means to cross protection boundaries
3. they enforce clean and simple interfaces, thus providing a natural aid for modular structuring of large (distributed) applications, and
4. they hide the distinction between *local* and *remote* communication, thus allowing static or dynamic reconfiguration.

The main aim of interprocess communication facilities is to deliver information with minimum latency and maximum throughput. For the delivering of continuous media like video it is also important to keep the irregularity of the latency to a minimum. If communication takes place over networks, interprocess communication facilities should hide as many failures as possible.

2.2.4 Some Definitions

Some terms are defined to avoid confusion in the following chapters.

A *program* is an executable file. It is usually created by a compiler, assembler and a linker. A *process* is a program that is being executed by the computer's operating system. If it is said that two computers communicate it is meant that two processes on these computers communicate (after [Stev90]).

UNIX processes normally use *descriptors* to reference I/O streams. A descriptor is a small unsigned number that refers to a particular stream. Read or write system calls can be applied to a descriptor to transfer data. The `close()` system call can be used to deallocate any descriptor (after [Leff88]).

The core UNIX operating system running on a computer is called *kernel*. The kernel provides basic services like memory management, CPU scheduling, file systems, and device I/O.

A computer *network* is a communication system for connecting end-systems (computers which run applications on behalf of users). End-systems are often referred as *hosts* if they are connected to a network. Where a large number of hosts are to be connected, so that any host may exchange data with any other, intermediate computers are usually used, which are referred to as network *nodes*.

Occasional references are made to the UNIX manual pages for additional details on a certain feature. The 8-section format used by both BSD and System V is assumed. References of the form `foo`(2) refer to the entry for *foo* in Section 2 of the UNIX manual (after [Stev90]).

2.3 Summary

The design and analysis for this document are sketched. Some helpful background information is given. Historically interprocess communication was a weak area of UNIX, but now a wide range of options is available. The two most important UNIX directions are System V from UNIX System Laboratories and BSD by the Computer Systems Research Group at the University of California at Berkeley. A lot of UNIX standards have been developed. Principles and aims of interprocess communication were stated. Some important terms are explained. The client/server model is important in the UNIX world. Most example programs in the following chapters try to use this model.

3 Local IPC

This chapter gives on overview of the most common local UNIX interprocess communication facilities, including

1. **Signals**
2. **Pipes (named and unnamed)**
3. **Stream Pipes**
4. **Messages**
5. **Semaphores**
6. **Shared Memory**

This chapter describes these interprocess communication facilities that are restricted to the use between processes that are executed on one computer.

All "local" interprocess communication facilities rely on memory for the exchange of information. Processes in the UNIX model can normally not see/access the memory of another process (except for System V shared memory regions, described later in this chapter). Therefore the kernel (which has access to all memory) has to copy the data first from the user address space into some internal kernel buffers (caused by a "write" system call), and from there to the address space of another process (caused by a "read" system call from that process). Figure 5 illustrates this process.

Semaphores and other related IPC facilities do not copy information, but modify memory that is controlled by the kernel.

How do processes get "in touch" with one another? Two basic approaches exist for this purpose:

- an object is stored in file system and accessed via pathnames (e.g. FIFOs and sockets of the UNIX domain use this method)
- the kernel handles some form of resource identifiers (e.g. *keys* for System V IPC facilities or descriptors for pipes).

The socket interface which provides distributed IPC facilities, but also IPC facilities that are only available between processes that run the same computer, is introduced in Chapter 6.1.

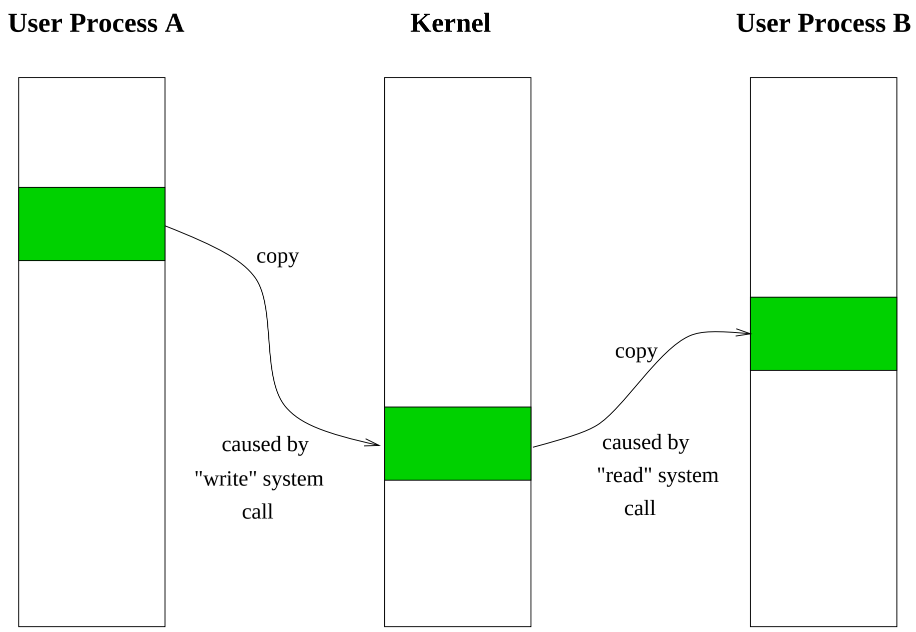

Figure 5: The Principle of most local IPC Facilities

3.1 Signals

Most UNIX applications have to deal with signals. A good introduction in the terminology of signals is given in [Zlo91]:

> [Signals] are a mechanism by which a process may be notified of, or affected by, an event occurring in the system. When the event that causes the signal occurs, the signal is said to be *generated.* When the appropriate action for the process in response to the signal is taken, the signal is said to be *delivered.* In the interim, the signal is said to be *pending.*

Signals are sometimes called "software interrupts", as they behave very similarly to hardware interrupts. Signals usually occur asynchronously (Figure 6). Signals can be sent from one process to another process (or to itself), or are generated by the kernel.

Signals that are not asynchronous are normally generated if an error occurs like an erroneous arithmetic operation.

A process can choose how to handle a specific signal. A signal can be treated in the following ways:

- it can be ignored (except `SIGKILL` and `SIGSTOP`),

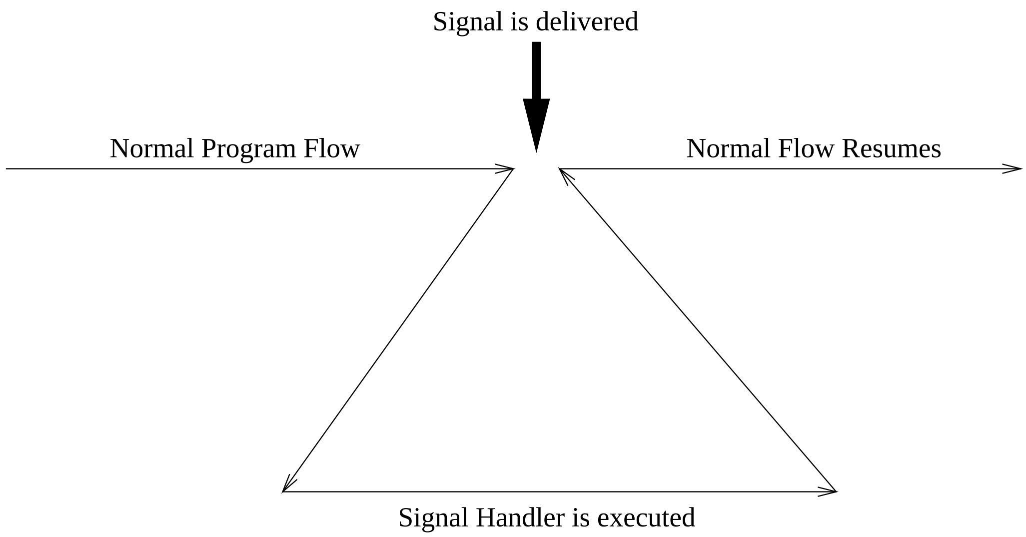

Figure 6: Execution of a Signal Handler

- or the "default action" can be executed (which usually causes termination with or without a core dump),
- or a specific signal-catching function can be invoked.

Originally the `signal()` system call was used to influence the behaviour of signals. The problem with the original implementation of signals was that they were *unreliable*, signals could be lost. A process also had to reinstall a signal handler every time it was invoked.

This was changed quite early by the AT&T and BSD version of UNIX, but unfortunately the changes are not compatible. The POSIX.1 standard, which defines a standard for reliable signals, is widely accepted in the UNIX world. Therefore only the signal-related functions of this standard are described here.

Every signal has a specific name. Table 3 states the signals which every POSIX-compliant system has to support. The system calls in Table 4 are useful for dealing with signals.

Some "slow" system calls like `read()`s from pipes are interrupted by a signal. They return the error code `EINTR`, and the program has the responsibility to issue the system call again if needed. As this is not always a desired behaviour, some systems allow an automatic restart of these slow system calls on a per-signal basis (option `SA_RESTART`, a Berkeley extension, with `sigaction()`).

Signal	Default Action	Description
SIGABRT	terminate w/core	Abnormal termination, normally caused by abort().
SIGALRM	terminate	Timeout, caused e.g. by alarm().
SIGFPE	terminate w/core	Erroneous arithmetic operation (e.g. divide by zero).
SIGHUP	terminate	Hangup on controlling terminal.
SIGILL	terminate w/core	Invalid hardware instruction.
SIGINT	terminate	Interactive attention signal (interrupt).
SIGKILL	terminate	Termination. Cannot be caught or ignored.
SIGPIPE	terminate	Write on a pipe that is not open for reading by any process.
SIGQUIT	terminate w/core	Interactive termination signal (quit).
SIGSEGV	terminate w/core	Invalid memory reference.
SIGTERM	terminate	Termination signal.
SIGUSR1	terminate	Application-defined signal 1.
SIGUSR2	terminate	Application-defined signal 2.

Table 3: POSIX Signals

Function	Description
kill()	sends a signal to a process
raise()	sends a signal
alarm()	schedules an alarm
pause()	suspends process execution
sigaction()	examines and changes signal action
sigprocmask()	examines and changes blocked signals
sigpending()	examines pending signals
sigsuspend()	waits for a signal

Table 4: Signal Functions

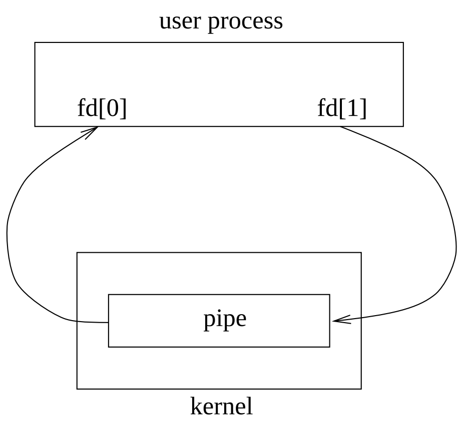

Figure 7: The Result of the `pipe()` System Call

Signal handling is useful in many contexts. An example is to signal another process that it is ready for a particular task or to do some clean-up operations like removing temporary files if a `SIGINT` signal is caught. The program `local/signal.c` (page 186) illustrates the use of the described signal functions. A more detailed description of POSIX.1 signal handling can be found in [Zlo91, Chapter 5], [POSIX.1, Chapter 6], and [Stev92, Chapter 10].

In terms of interprocess communication signals can not transport large amounts of information from one process to another process. The advantage of signals is that they normally occur completely asynchronously. Signals were used in some of the performance measurement programs in Chapter 7 to synchronize different processes.

3.2 Pipes

Pipes are the oldest IPC mechanism in UNIX, therefore they are supported by all UNIX systems. Pipes are created with the `pipe(int fd[2])` system call, which creates two new descriptors. These descriptors are connected, the first one is used for reading and the second one for writing. The result of a `pipe` system call can be seen in Figure 7.

Pipes are normally useless in a single process. They become useful when a process wants to communicate with its children or vice versa. Children are created with the `fork()` system call. The result of a fork on a pipe is shown in Figure 8.

If the data should flow from the child to the parent, the child closes the read end of

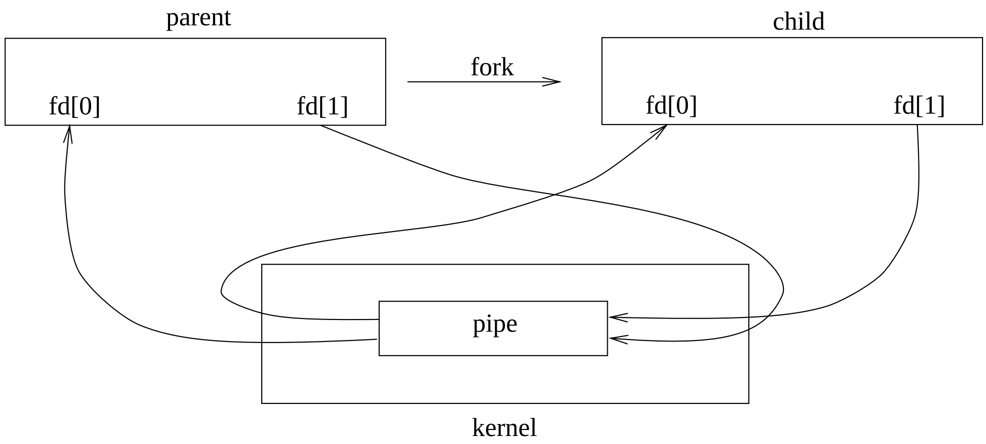

Figure 8: A Pipe after a `fork()` System Call

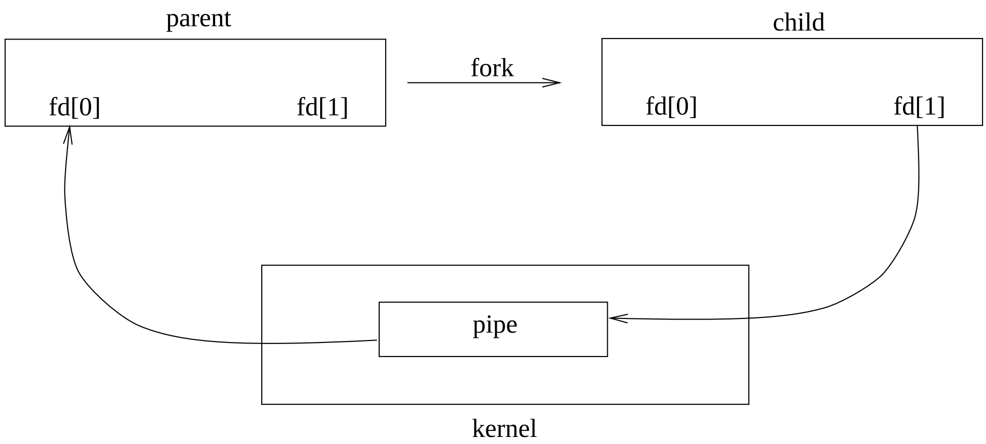

Figure 9: Resulting Pipe

the pipe, and the parent process closes the write end. This results in Figure 9.

The UNIX system guarantees that a certain number of bytes are written atomically to a pipe. This number of bytes varies, but is normally 4096 or 8192 bytes (the system constant `PIPE_BUF` should be used for portability). This can be used if several processes have to communicate with one process.

The use of pipes has two major drawbacks:

1. the processes which want to communicate must have a common ancestor

2. the data can only flow in one direction.

FIFOs (described next) can solve the first problem, while stream pipes (Chapter 3.4) solve the second one. Program `local/pipe.c` (page 177) demonstrates the use of pipes.

3.3 FIFOs (named pipes)

FIFOs (for First In, First Out), sometimes also called "named pipes", solve the problem that processes have to be related when they wish to communicate. A FIFO is created with the system call `mkfifo(const char *pathname, mode_t mode)`, or the shell command `mkfifo`.

FIFOs use the *pathname* to establish a connection. A special file in the file system is created with the permission bits *mode* which can be seen by all processes. A FIFO can then be used with the normal I/O functions like `open()`, `read()`, `write()` and `close()`. One or more processes may open a FIFO for reading and writing.

FIFOs are often created by shells and used by shell commands to pass data from one process to another. The use of FIFOs removes the necessity for temporary files. Another use for FIFOs is the exchange of data in a client/server application where the processes are not related (e.g. they are created dynamically from different processes). The programs `local/fifoclient.c` (page 167) and `local/fifoserver.c` (page 169) demonstrate the use of FIFOs.

FIFOs have the drawback that communication is still only possible in one direction. Either two FIFOs have to be used for communication in both directions, or stream pipes (described next). Another problem is the use of file names, as it may not be clear what happens if a program crashes and leaving a FIFO in the file system, without the other processes notifying the crash of the communication partner.

3.4 Stream Pipes

A so-called "stream pipe" is a bi-directional pipe. To obtain bi-directional data flow between parent and child, only a single stream pipe is required [Stev92]. Stream pipes can be named or unnamed.

In SVR4 every pipe is already a bi-directional pipe, so the normal `pipe()` system call creates always an unnamed stream pipe.

	Message Queue	Semaphore	Shared Memory
system call to create or open	`msgget()`	`semget()`	`shmget()`
system call for control operations	`msgctl()`	`semctl()`	`shmctl()`
system calls for IPC operations	`msgsnd()`	`semop()`	`shmat()`
	`msgrcv()`		`shmdt()`

Table 5: Summary of System V IPC System Calls

In BSD an unnamed stream pipe can be obtained via the `int socketpair(int` *`domain`*`, int` *`type`*`, int` *`protocol`*`, int` *`fd`*`[2])` system call like this:

```
int fd[2];
int success;
success = socketpair(AF_UNIX, SOCK_STREAM, 0, fd);
```

For a description of socket parameters please refer to Chapter 6.1.

Named stream pipes are more difficult to establish. The function `namedstreampipe()` in the file `local/namedstreampipe.c` (page 175) implements named stream pipes for BSD systems with the use of sockets. Named stream pipes are possible within SVR4, but have to be created by the superuser and are therefore not discussed further.

3.5 System V IPC

The System V IPC facilities were first added to System V. They include three different types of IPC with a very similar interface (see Table 5 for summary of system calls):

- message queues
- semaphores
- shared memory

A System V IPC object (a message queue, a set of semaphores, or a region of shared memory) has an unique identifier, a *key*. Keys are comparable to the file descriptors used for normal I/O, but they exist system wide, they are not related to processes. To access a certain System V IPC object the key has to be known. Key identifiers have to be selected carefully, otherwise conflicts between unrelated applications might arise. Message queues, semaphores and shared memory are using different name spaces, e.g. a key for a shared memory region is meaningless for semaphores and messages. To get unique keys the library function `key_t ftok(char *pathname, char proj)` can be used. This function converts the pathname of an existing accessible file and a "project identifier" (an arbitrary 8-bit value) into a (hopefully unique) key suitable for System V IPC.

Each IPC object has a set of IDs and permissions (reading and writing) like files in a file system. This allows control over the access to a specific IPC object. The owner ID and group ID specifies processes with owner or group access privileges.

All three forms of System V IPC have built-in limits. These may be changeable on some systems by reconfiguring the kernel, others have fixed limits. The maximum number of message queues, shared memory regions and semaphore sets that a process can have on the three different systems that are used (and described) for performance measurements in Chapter 7 was derived with the program `environment/sysvipclimits.c` (page 299) (and adjusted according to the `ipcs` command). A word of caution: on some systems the number of available IPC objects per process is equal to the number of system wide available IPC objects. On other systems the system wide number of IPC objects is higher than the number of IPC objects a process can obtain.

3.5.1 Message Queues

Message queues are a linked list of messages stored within the kernel and identified by a message queue key [Stev92]. Message queues are used to send data from one process to another process. Every message can have a priority, and the receiver can specify the priority of the messages it is willing to receive. This can also be used to allow bi- or more-directional flow of data if every sender identifies its data with a different priority. The size of individual messages is often very limited, normally to about 2 to 4 Kb. Table 6 gives an overview of the system calls that can be used to handle message queues.

The program `local/message.c` (page 171) demonstrates the use of message queues by implementing the classical producer/consumer problem. Table 7 shows the maximum number of message queues a process may have at best on a variety of systems.

3.5.2 Semaphores

Semaphores are used for managing a fixed amount of shared resources. The UNIX System V IPC semaphores have more capabilities than normal abstract semaphores [Stev92]:

System Call	Description
`msgget()`	open an existing message queue or create a new queue
`msgctl()`	performs various operations on a message queue, e.g. removing a message queue
`msgsnd()`	place data onto a message queue
`msgrcv()`	retrieve a message from a message queue

Table 6: System Calls for Handling Message Queues

System	Message Queues
SparcCenter 2000/Solaris 2.5	50
HP 9000/710/HP-UX 9.01	50
i586/Linux 1.2.9	128

Table 7: Maximum Number of Message Queues per Process

- they are correctly called *semaphore sets*, as a System V semaphore can consist of a set of several independent semaphore values. The number of used semaphore values has to be specified at the creation of a semaphore set.
- all System V IPC objects are system wide. This means that a semaphore stays in existence even if no process is using it. This may cause problems if a process quits without releasing a semaphore. To avoid this problem, a special "undo" feature is available. If a program specifies the flag `SEM_UNDO` for a semaphore operation, the kernel remembers how many resources were allocated for that particular semaphore. When the process terminates (on purpose or due an error) the kernel applies the correct adjustment for the corresponding semaphores automatically.

Table 8 gives an overview of the system calls that can be used to handle semaphore sets.

A problem with System V IPC semaphores is that the creation of semaphores is independent of their initialization. This can lead to race conditions, an atomic system call that handles this problem would be desirable.

The program `local/semaphore.c` (page 179) shows how semaphores can be used to coordinate two different processes. Table 9 shows the maximum number of of semaphore sets a process may have at best on a variety of systems.

System Call	Description
`semget()`	obtain a semaphore key
`semctl()`	performs various operations on a semaphore set, e.g. removing a semaphore set
`semop()`	perform atomically an array of operations on a semaphore set

Table 8: System Calls for Handling Semaphore Sets

System	Semaphore Sets
SparcCenter 2000/Solaris 2.5	10
HP 9000/710/HP-UX 9.01	64
i586/Linux 1.2.9	128

Table 9: Maximum Number of Semaphore Sets per Process

System Call	Description
`shmget()`	obtain a shared memory key
`shmctl()`	performs various operations on a semaphore set, e.g. removing a shared memory region
`shmat()`	attach a shared memory region to the address space
`shmdt()`	detach a shared memory region from the address space

Table 10: System Calls for Handling Shared Memory Regions

3.5.3 Shared Memory

Shared memory overcomes the normal isolation between process address spaces. It allows windows of memory to be shared among a number of processes (possibly at different address spaces). The access to the shared memory is not interlocked, therefore the processes using shared memory normally have to coordinate the access themselves for example with semaphores. Shared memory is accessed and manipulated using normal instructions. Table 10 gives an overview of the system calls that can be used to handle shared memory regions.

The maximum number of shared memory regions a process can have on each of the

System	Shared Memory Regions
SparcCenter 2000/Solaris 2.5	6
HP 9000/710/HP-UX 9.01	120
i586/Linux 1.2.9	128

Table 11: Maximum Number of Shared Memory Regions per Process

computer systems used varies greatly. (see Table 11). The program `local/shared.c` (page 184) demonstrates the use of shared memory.

3.5.4 System V IPC Problems

System V IPC facilities often create problems for system maintenance, as the IPC structures are system wide without a reference count. If a program crashes before the IPC structures are freed, or the program which uses these IPC facilities has a bug, they remain in the system till it is rebooted. This causes a problem, as the maximum number of semaphore sets etc. is limited.

For this reason two programs can be used to get information about System V IPC in a system or to delete them:

- `ipcs`(1) — provide information about System V IPC facilities
- `ipcrm`(1) — remove System V IPC resources

3.6 Summary

A large number of different local IPC facilities are available in UNIX. Signals, pipes, FIFOs, stream pipes, message queues, semaphores, and shared memory were described. Signals allow asynchronous reaction to special circumstances. Semaphores are used to synchronize access to shared limited resources. The use of pipes, FIFOs, stream pipes, and message queues, used to exchange data, is very similar. Example programs for all facilities demonstrate the use of these IPC facilities. Some IPC facilities have built-in limits, some of them are stated.

4 Networking Principles

This chapter describes

- **the principles of networking**
- **its underlying concepts**
- **the OSI seven layer model**

Processes running on different computers can only communicate if there is some form of connection between the computers. This chapter describes the basic principles of networking. The knowledge of this chapter is needed in the following chapter to understand the potential and limitations offered by the TCP/IP protocol suite, which is used by all UNIX interprocessor communication facilities.

4.1 Switching Techniques

There are three basic methods for transmitting data from one endpoint to another endpoint via electronic networks:

- Circuit Switching
- Message Switching
- Packet Switching

In *circuit switching*, the oldest method, a link has to be set up along all the nodes forming a path between the two hosts. This link has to be maintained for the whole duration of the transmission. If one node between the two hosts crashes or is removed, the connection is broken. A typical use of this type of switching is using the phone. After dialing, one cable is reserved. It does not matter if one speaks or not. This type of communication is therefore expensive, as resources are often not used effectively. In the days of multimedia (video, audio, etc.) with the need for a constant data stream to be transmitted in a certain time this technique is getting more attention again. When circuit switching methods are used data can be transferred without additional information like destination address, control information etc. (one does not have to do anything to maintain a phone call).

As circuit switching is very expensive new methods for transmitting data were needed. *Message switching* takes into account that computers do not exchange data all the time. With message switching no real connection is established between the two hosts. All messages are transmitted independently. Therefore all messages need the full information about the destination, and other control information. Messages can have various sizes up to a fixed limit. All nodes along the path must be able to store the entire incoming message and have control facilities for forwarding the message to a node in the right direction. This technique is therefore called *store-and-forward*. If no suitable node is available, and a fixed number of retries are unsuccessful, an error message is normally generated and sent back to the origin. As multiple paths are often available for routing, the failure of one node does not normally matter. If too many hosts try to exchange data serious congestion can result. Message switched data normally needs much longer to be transferred to the endpoint, as messages are stored for variable times in the nodes between the endpoints. Message switching makes much better use of available resources than circuit switching. Figure 10 illustrates message switching.

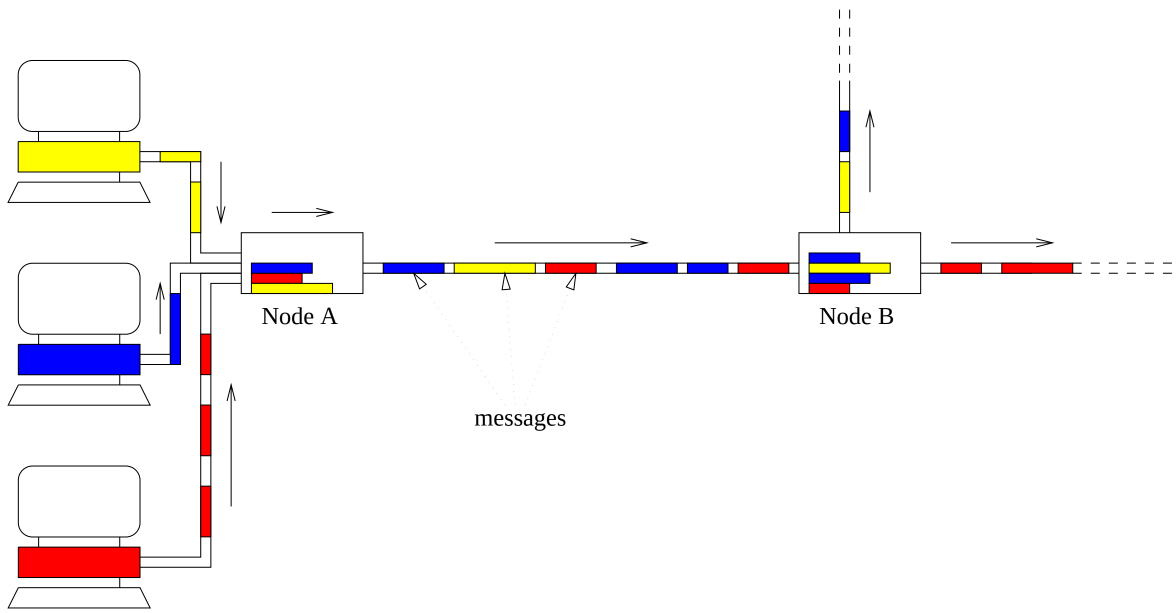

Figure 10: The Principle of Message Switching. Only one direction of data flow is shown.

A special combined form of message and circuit switching is called *packed switching*. Here the original data is split into very small fixed length *packets*. These packets again need some information about the destination. The advantage of these small packets is that they can be forwarded very fast. The amount of storage needed is decreased. This allows much higher throughput of data than message passing. At the endpoint the packets are reassembled. A very popular method, ATM (Asynchronous Transfer Mode), uses this technique. ATM is described in detail in [CCR95/2].

The most used technique with computer networks is message switching. The other techniques are not discussed further.

One might think that connection-oriented communication is equivalent to using circuit switching and connectionless communication equivalent to message switching. The first is not really the case, although they are closely connected. Connection-oriented protocols like TCP (described in Chapter 5.9) use messages to transfer data. TCP establishes a *virtual* connection, which has no guarantee about throughput, but emulates all the other properties of a circuit switched connection.

4.2 Message Passing Concepts

Message passing techniques differ mainly in the way they deal with the following issues:

1. Is the connection between sender and receiver connection-oriented (virtual circuit switched) or connectionless?
2. How are messages addressed?
3. How are sending and receiving activities synchronized (are the send and receive operations blocking? If yes, how long?)

Issues 1 and 3 were addressed in Chapter 2.2.1. Therefore they are not discussed further in this chapter.

4.3 Addressing Issues

Normally there are multiple ways to designate with whom to communicate:

- by name (e.g. object A)
- by address (e.g. object at location B)
- by content (e.g. object with value C)
- by route (e.g. object found at end of path D)
- by source (e.g. all objects owned by E)
- broadcast identifier (e.g. all objects of type F)
- group identifier (e.g. all objects related to G)

In all cases it is necessary that the name ("A" to "G") has to be mapped to the real object or destination. This can only be done within a specific *context* (also called *space*). In a network for example the name "A" of a computer can be resolved by its address "B", which in turn can be used in the next layer and so on. Figure 11 illustrates this.

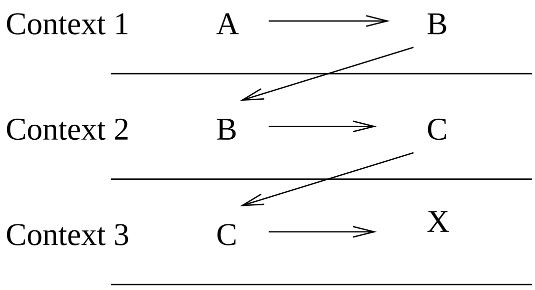

Figure 11: Layered Naming Contexts

4.3.1 Static vs. Dynamic Binding

On a UNIX system the simplest way to address a message is to use the process identifier of the receiving address. However, if processes communicate via a network, a process number is not unique enough. A more precise destination identifier, e.g. the tuple {*host address, process id*} has be used. To use the direct naming scheme, the client must know the physical network address of the node where the server process runs on. If the network card of the server is replaced this information becomes invalid. Therefore this *static binding* technique is seldom used. A *global name space* of addresses has to be used. A global name is then an identifier that is bound *dynamically* to the server's network address at boot time (or changed even later).

4.3.2 Global Name Space

To have the ability to address anyone in a given context the name has to be unique. Logically names can only conflict if their contents are equal. An example is the process ids in a network: their naming contexts conflict. A solution is to use *federated namespaces* which are constructed out of several distinct and independent name contexts. Often hierarchical concatenation can be used to build a global name space. An example is how to make an address of a person unique in the the universe:

milky way/solar system/planet/country/city/street name/house number/flat number/name

This system can also be used for networks, where the result is to identify every process on all computers unequivocally. Here is an example for a network with subnetworks:

network/subnetwork/host name/process id

Another way for all participants to agree on a partition of ranges, e.g. computer A only has process ids from 0 - 999, computer B from 1000 - 1999 and so on.

Both techniques can be combined to achieve a global name space.

4.3.3 Forms of Dynamic Binding

To archive dynamic binding multiple possibilities exist. To map global names to network addresses a file could be used. The use of a file, however, gets impractical if the network is big. Also this file has to be consistent and transmitted to every host. Even if the file is accessible via a network file system it has still to be updated every time a new computer is connected to the network or removed from it.

Broadcasting of all addresses might be useful in small networks, e.g. in token rings or Ethernets where all messages are seen by all computers anyway. In large networks broadcasting is often too expensive.

A method often used is the use of *name servers*, which are used to resolve global names or at least parts of a global name. A client asks a name server for the network address of a global name. The client can then communicate directly with the intended server. The only problem is: how to contact a name server? Name servers can either have a fixed well-known address, or a client can broadcast a message on the local network with the name server responding with its address. Often more than one name server is used to avoid problems when the only name server is down or congested with too many requests.

4.4 Data Representation

Different types of computers with different kinds of CPUs exist. The same information is therefore not always represented in the same way. Problems especially occur due to:

- the size of a byte
- the order of bytes in words, longwords etc.

- the representation of floating point numbers
- the size of pointers
- the character set for text
- the aligning of data

Nowadays the size of a byte is quite stable: it consists of 8 bits on nearly all computers. However this was not always the case. Therefore in networking the term *octet* is often used to specify that exactly 8 bits are meant. In this document the term octet is not used to avoid confusion.

The order of bytes in data sizes longer than 8 bits is still a major problem. Two different orders of bits are used: the *little endian* order where the most significant byte comes first, and the *big endian* order where the most significant byte is the last byte. Figure 12 illustrates this.

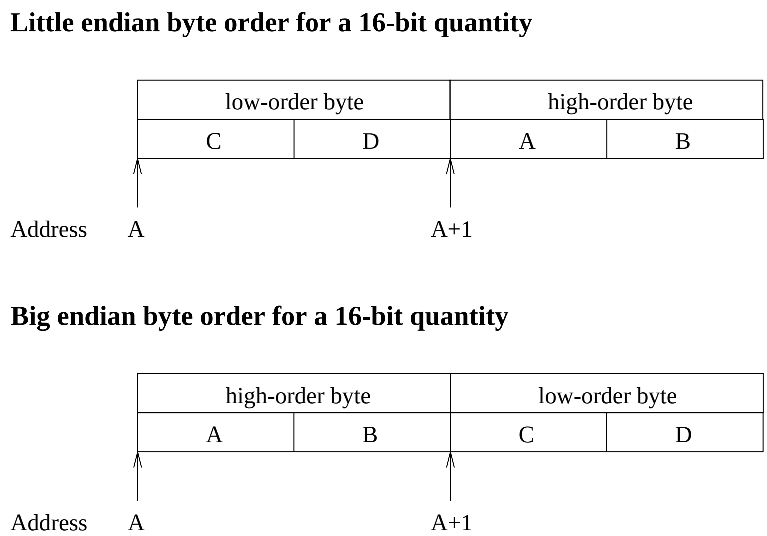

Figure 12: Little vs. Big Endian with 0xABCD as Example Number

The Motorola 680x0 series, Sun SPARCstations and IBM mainframes use the big endian byte order, while DEC VAXes and Intel 80x86 for example use the little endian byte order.

The representation of floating point numbers also differs. Most modern computers use the IEEE 754 standard to represent single-precision floating point numbers.

The size of pointers is not really an issue, as pointers are normally meaningless on other computers anyway. If space for pointers has to be reserved, one should consider that the number of 64-bit computers is increasing rapidly.

The ASCII (American Standard of Information Interchange) character set is used on most common computers, although some IBM mainframes still use EBCDIC to encode characters.

If different computers have to exchange data structures one has to consider the alignment of data. Some computers align longwords on 16-bit boundaries, others on 64-bit boundaries. Therefore one has to be careful when using the same data structures to exchange data between different computer types. Even compilers sometimes align data on the same system differently.

As one can see the need for data conversion is inherent in networking. The problem of byte ordering is often solved in declaring a *network byte order* which everyone has to use if data is exchanged. Most text is exchanged using the ASCII character set. The other issues are often handled as application specific.

4.5 Protocols

A protocol is a set of rules and conventions between the communicating participants [Stev90]. They can be very complex. One engineering technique that deals with this complexity is *layering*. Therefore protocols are normally designed in *layers*. Every layer N provides a service to layer N+1, and uses the service of the layer N-1 below. Layer N on one computer communicates virtually directly with the same layer on the other computer, even if the data flows down the layers on the first one and up on the second one. Figure 13 illustrates this. Between the different layers different protocols are used. The result is called *protocol stack*. A *protocol suite* is the combination of different protocols at various levels.

4.5.1 Advantages of Protocol Layering

A layered approach is useful for several reasons [Elb94]:

1. The services provided by a given layer are specific and well-defined. Additionally, they are made available consistently to the end user at the upper interface. The value in this consistency is that protocol stacks can be constructed using a

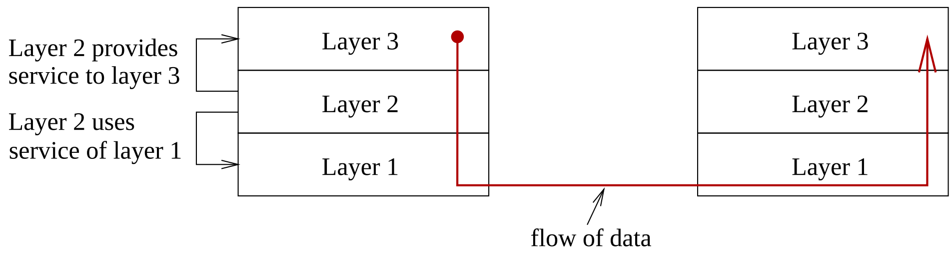

Figure 13: A simple Protocol Stack

building block approach, leveraging the advantages of such software engineering techniques as abstraction and encapsulation.

2. Successively more valuable functionality can be added to a protocol stack by adding layers and applications, without regard for how the underlying services are implemented

3. For the reasons given above, a layer that provides a well-defined service to its upper layer users, and utilizes the well-defined services of its lower layer supplier, can employ a substitute protocol without changing either the upper or lower layer protocols. This is useful in taking advantage of newer, faster, and more efficient protocols without disturbing the entire protocol stack implementation.

4.5.2 Disadvantages of Protocol Layering

All data has to flow through all layers. If individual layers do not add much functionality, this approach is inefficient and leads to poor performance.

4.5.3 Protocol Control Information

An encapsulation technique is normally used to exchange protocol information. Layer N takes the data from the layer above, adds its own protocol control information and handles this to the next layer down. This is comparable to putting something in an envelope, putting this together with other information into a bigger envelope, and so on. Figure 14 illustrates this process.

4.6 The OSI Seven Layer Reference Model

The Open System Interconnection (OSI) reference model, as established in ISO 7498, is often used to compare different networking architectures. It is a formal description of a protocol stack architecture. The OSI reference model was positioned as a framework

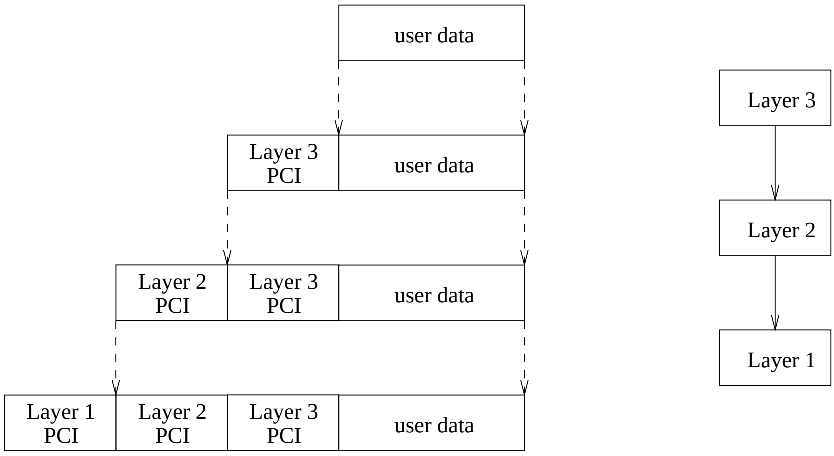

Figure 14: Adding Protocol Control Information in each Level

7	Application Layer
6	Presentation Layer
5	Session Layer Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer

Figure 15: The ISO-OSI 7 Layer Reference Model

for the development of an extensible and scalable protocol suite for communication among open systems [Elb94]. The OSI reference model and implementations of it have to be distinguished. The OSI reference model consists of seven layers (Figure 15). The principle is that each layer can be specified and implemented more or less independently of those above and below as long the interfaces are consistent.

The layer definitions of the OSI reference model are commonly used and generally accepted, although not all layers and services are present in most common networks.

4.6.1 Application Layer

The application layer contains all the utilities that support application programs. This layer does not include applications itself. Services like file transfer, e-mail, virtual terminals and others are part of the application layer.

4.6.2 Presentation Layer

The presentation layer is responsible for presenting the data in a way that is meaningful for an application. This includes details about the CPU and operating system dependent encoding of data as described above. This allows the application layer to communicate without regard to the actual presentation of data on a specific system.

4.6.3 Session Layer

The session layer manages the data exchange between open systems in an orderly fashion [Elb94]. It creates, manages, and terminates the dialogues between the users.

4.6.4 Transport Layer

The transport layer takes packets of data from the Network layer and assembles them into messages. Two different kinds of services are supported: connection-oriented and connectionless services. The first one is used to provide end-to-end, error-free service to the session layer. The support of connectionless transport services is minimal; these are used if the need for the services above is not critical.

4.6.5 Network Layer

The network layer is responsible for routing from end to end within a network [Elb94]. The services provided include network addressing, blocking and segmenting message units, switching and routing, and controlling congestion. As in the transport layer connectionless and connection-oriented services exist.

4.6.6 Data Link Layer

The data link layer is responsible for the error-free transport of data between adjacent systems. It shields the upper layers from details concerning the physical transmission. The data link layer contains services like media management, framing, and flow control. Three types of service are provided: connection-oriented, acknowledged connectionless and unacknowledged connectionless.

4.6.7 Physical Layer

The physical layer is responsible for the actual transport of data. Here functional, electrical and procedural characteristics are specified: signal timing, the voltage levels, connector types, etc.

The flow of data between endpoint in the OSI model is summarized in Figure 16.

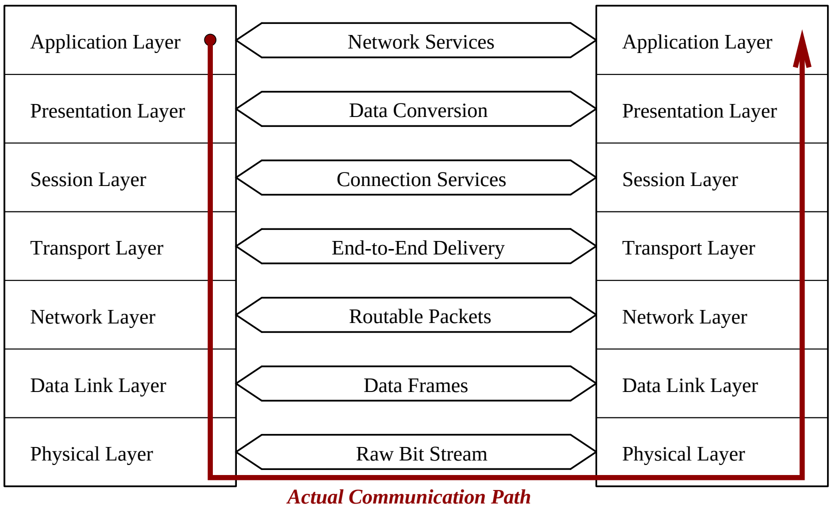

Figure 16: Communication between OSI Layers

4.6.8 Criticism of the OSI Reference Model

The OSI Reference Model can be criticized in many areas. Some of its problems are:

- Originally it completely ignored connectionless services and connectionless protocols, even though that is the way most local area networks work [Tan88].
- It uses too many layers, therefore it is unnecessarily expensive.
- The model is extraordinarily complex [Tan88]
- The levels of the OSI protocols are not equally important.
- Some functions like addressing, flow control, error control are repeated in all levels

Further problems of the OSI Reference Model are discussed in [Day95] and [Tan88, page 30ff, 274ff]

4.7 Protocol Suites

A huge number of different network protocol suites have been developed. Some of the more commonly used ones are:

- Open Systems Network Architecture – OSI (open, complicated, not fully specified, pressure to use it from governments)

- The Internet suite of protocols: Transmission Control Protocol/Internet Protocol – TCP/IP (open, free, linked with UNIX, used in the Internet, supported by Microsoft)
- IPX/SPX from Novell (mainly used for LANs & PCs)
- System Network Architecture – SNA from IBM (first protocol suite, widely used in IBM networks)

TCP/IP is used mostly to connect different systems, as it is supported by nearly all vendors/products, especially in the UNIX environment. BSD 4.2 was the first operating system which used TCP/IP. As the source code is free, this led to widespread use and implementation of this protocol.

4.8 Summary

Networking adds a lot of complexity. The layering technique is used in multiple levels to deal with this increased complexity. The OSI Reference Model is often used to describe networking in theory, even if this model has shortcomings. The most used protocol suite in the UNIX world is TCP/IP and is discussed in the following chapter.

5 Networking with TCP/IP

This chapter covers

- **the history and goals of TCP/IP**
- **its underlying concepts**
- **a comparison with the OSI model**
- **a detailed description of checksums and timeouts**
- **some basic UNIX network tools**

TCP/IP is *the* protocol suite for UNIX interprocessor communication. Therefore some knowledge about it is needed. This chapter gives an overview of some aspects of the TCP/IP protocol suite. Chapter 6 describes how this protocol suite can be used from applications for interprocess communication.

5.1 Overview

The TCP/IP or Internet Protocol suite consists of several protocols. The more common ones are:

TCP The *Transmission Control Protocol* is a connection-oriented protocol which provides a reliable, full-duplex byte stream. It uses the Internet Protocol to transfer data. TCP is the most used protocol for user processes. This is the reason why the Internet Protocol suite is often called TCP/IP.

UDP The *User Datagram Protocol* is an unreliable connectionless protocol for user processes.

ICMP The *Internet Control Message Protocol* is used for handling error and control information. It uses the Internet Protocol to exchange data. User processes normally do not need access to ICMP, as ICMP messages are generated and processed by the TCP/IP networking software.

IP The *Internet Protocol* is the basic protocol that provides a best effort packet delivery service for UDP, TCP, and ICMP. User processes normally use only TCP or UDP, the direct use of IP is seldom.

ARP The *Address Resolution Protocol* maps an Internet address to a hardware address. Only some networks need it.

RARP The *Reverse Address Resolution Protocol* maps a hardware address into an Internet address.

Figure 17 illustrates the relationship of these protocols. The following text concentrates mainly on UDP and TCP, as these are directly accessible from user processes.

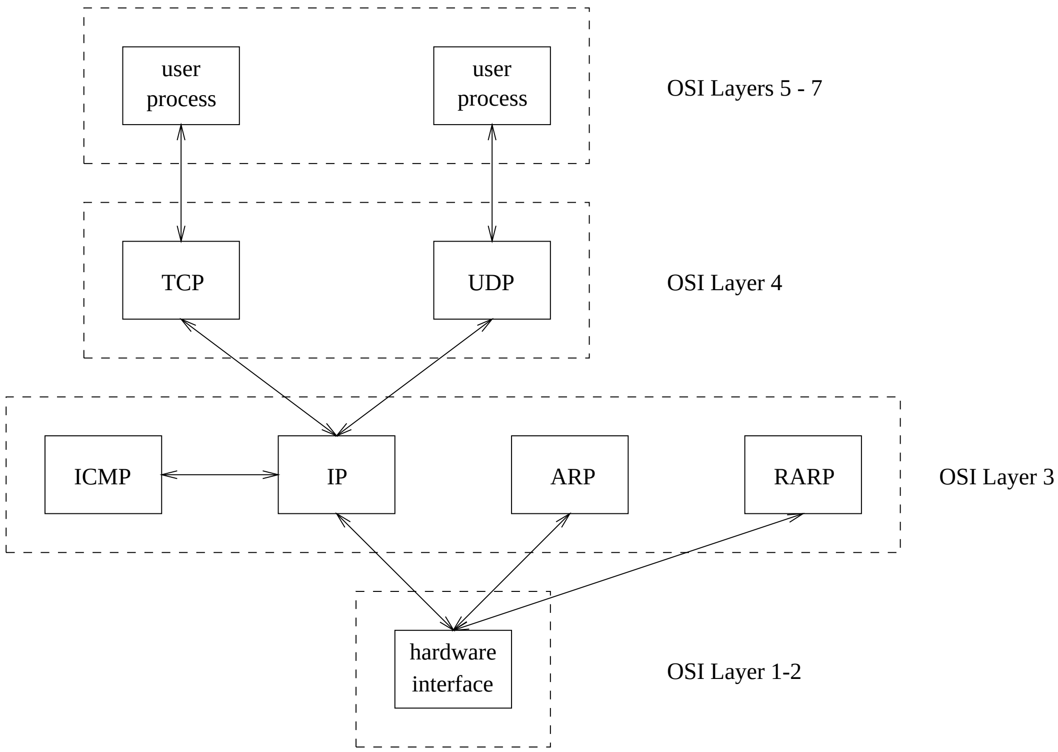

Figure 17: The Components of the Internet Protocol Suite

The network byte order in the TCP/IP protocol suite is the big endian byte order. This byte order is required for all integer values which are not part of the transmitted data.

As with all layered protocol suites, every protocol layer encapsulates the data from the previous level. This is shown in Figure 18 for data flowing down the protocol stack to the Ethernet level.

5.2 TCP/IP compared to the OSI protocol stack

The TCP/IP protocol suite can be compared with the OSI reference model quite easily. Typical TCP/IP applications like ftp, telnet or SMTP cover layers 5 to 7. UDP and

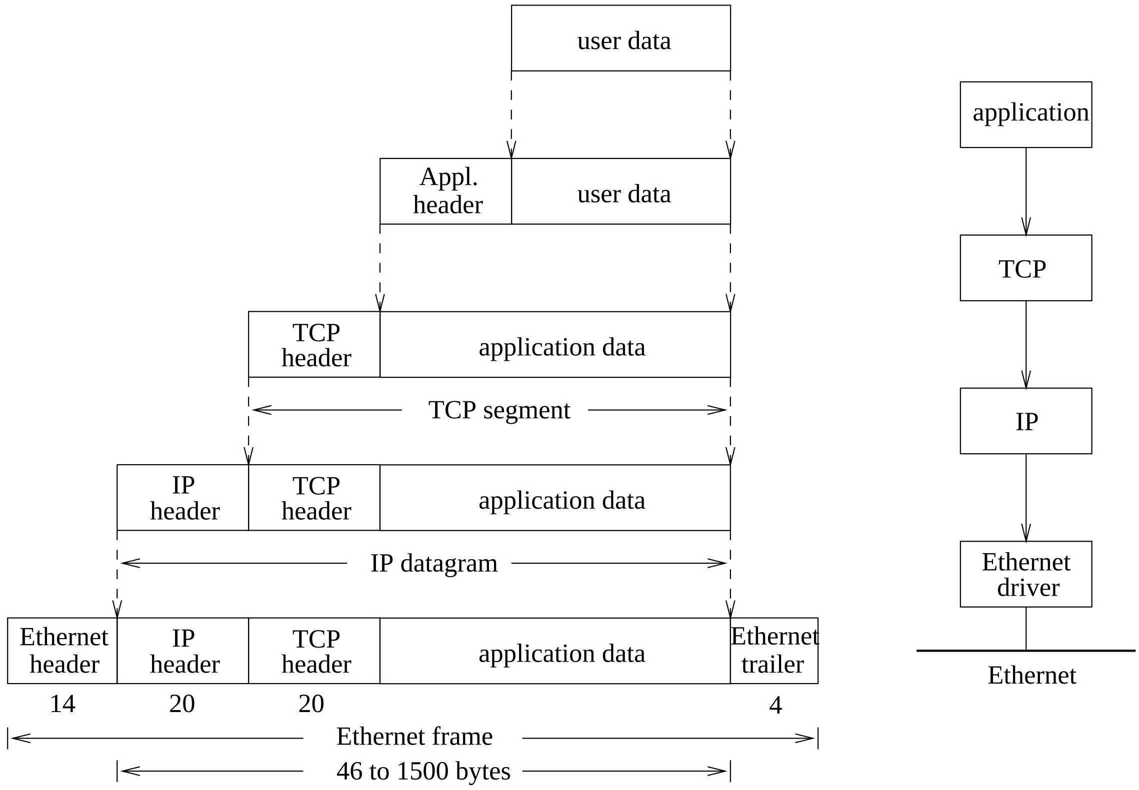

Figure 18: Encapsulation of Data as it goes down the Protocol Stack

TCP are comparable to the transport layer, layer 4. IP and some other protocols cover layer 3, the network layer. Layers 1 and 2 (data link and physical layer) are not defined by the TCP/IP protocol suite. Here Ethernet or other suitable protocols are used. Figures 17 and 19 illustrate the relationship.

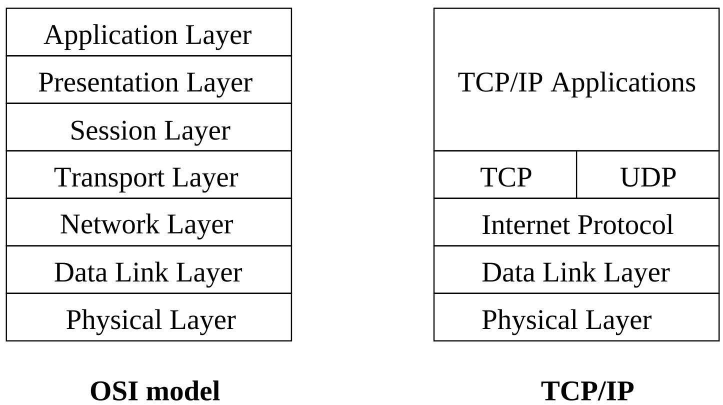

Figure 19: OSI Model versus TCP/IP Protocol

5.3 History and Goals of TCP/IP

The Internet protocol suite, TCP/IP, was developed by the Defense Advanced Research Projects Agency (DARPA) in the mid 1970s, with the architecture and protocols taking their current form around 1977-79.

The technique selected for multiplexing was packet switching, as the applications which should be supported were naturally served by the packet switching paradigm, and the networks which were to be integrated together were packet switched networks.

The top level goal for the DARPA Internet Architecture was to "develop an effective technique for multiplexed utilization of interconnected networks" [Clark88]. This goal was reached, and now thousands of networks (connected or not connected to the famous Internet) use the TCP/IP protocol suite.

Second level goals were (in decreasing order of importance): communication must continue despite loss of service; support for multiple types of service; the Internet architecture must accommodate a variety of networks; permit distributed management of resources; be cost effective; permit host attachment with a low-level of effort; and resources used must be accountable.

Especially one goal, the continuing of communication despite loss of service, had a major impact on the design. To achieve this goal, the state information which describes the on-going conversation must be protected. Therefore this information is gathered at the endpoints of communication, no intermediate packet switching nodes need to have any state information.

A detailed review of the design philosophy of the DARPA Internet protocols is given in [Clark88].

5.4 Internet Addresses

Each host on a TCP/IP internet is assigned a unique 32-bit internet or IP address that is used in all communication with that host [Com95]. This address encodes the identification of the network to which a host is attached to (netid) as well as the identification of a host on that network (hostid). Therefore IP addresses do not specify an individual computer, but rather a connection to a network. Five different sized classes A to F of IP addresses exist (see Figure 20). Classes A to C are used to identify hosts.

Class	Lowest Address	Highest Address
A	0.1.0.0	126.0.0.0
B	128.0.0.0	191.255.0.0
C	192.0.1.0	223.255.255.0
D	224.0.0.0	239.255.255.255
E	240.0.0.0	247.255.255.255

Table 12: The Ranges of IP Address Classes

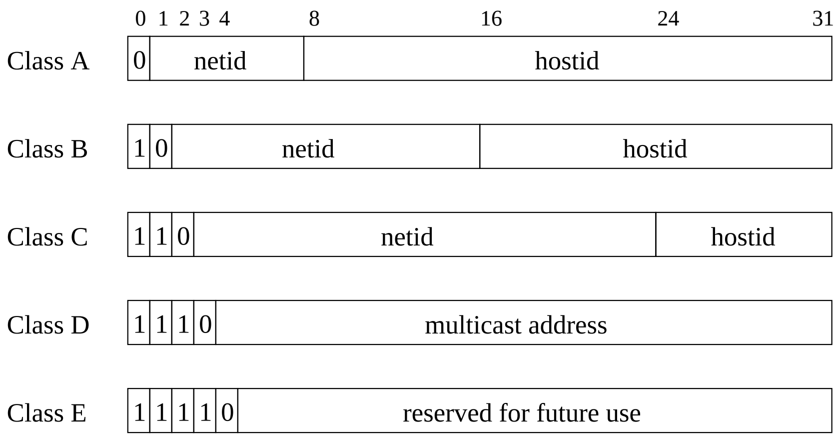

Figure 20: Internet Address Classes

IP addresses are normally written as four decimal integers separated by decimal points (*dotted decimal notation*), where each integer corresponds to 8 bit of the 32 bit address. An IP address (binary)

```
10000000 10000001 10000010 10000011
```

is therefore written as

```
128.129.130.131
```
.

Table 12 shows the ranges of IP addresses for the different classes. Addresses not shown are reserved for special purposes, e.g. 127.0.0.1 is a loop-back address for the local host. This feature was used to measure the performance of distributed IPC facilities between processes on the same computer for Chapter 7.

For humans numbers are still difficult to remember. Therefore each host can have a symbolic name which can be resolved via local databases or name servers.

Problems with IP addressing are:

- If a host moves from one network to another, its IP address has to change.
- A host has to have a different IP address for every network it is connected to.
- The number of IP addresses is limited. It is assumed that the Internet will run out of addresses in the year 2005 (+/- 5 years), therefore a new addressing scheme together with a new version of the Internet Protocol is developed (see Chapter 9.4.1).

5.5 Port Numbers

IP datagrams only specify a host as destination. But how does a datagram reach the right application? TCP and UDP use 16-bit *port numbers* for this purpose. Port numbers are more flexible as process ids, as one process may have several distinct connections with another process at the same time using several ports. One port can also be used to receive messages from several sources. Figure 21 illustrates the port model. Only one direction of data flow is shown in Figure 21 to simplify the figure, therefore client ports are omitted.

Servers normally listen for client requests on a *well-known* port number. Well-known port numbers are defined in the file `/etc/services`. The World Wide Web HTTP (HyperText Transfer Protocol) as an example normally uses TCP and UDP port number 80. The *Internet Assigned Numbers Authority* (IANA) manages the well-known port numbers between 1 and 1023. On UNIX systems the well-known ports are *reserved ports*. Only processes with superuser privileges can assign themselves a reserved port number. This increases security for `rlogin` and other areas.

Client processes normally do not care what port number they use, as long as the used port number is unique on their host. Normally the operating system assigns an *ephemeral* (short lived) port for them. This is due to the fact client processes are started and terminated when the user decides, whereas server processes typically run as long as the host is up.

5.6 Multiplexing/Demultiplexing

Communication protocols use techniques of *multiplexing* and *demultiplexing* throughout the layered hierarchy. When sending a message, the different protocol layers of

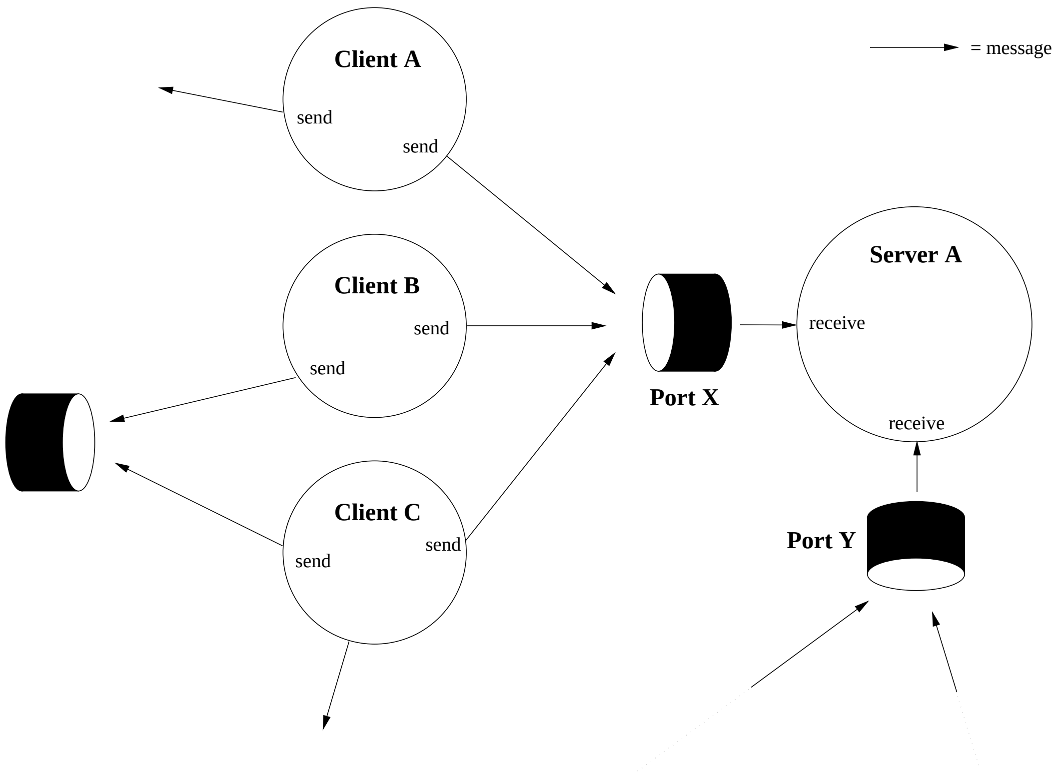

Figure 21: The UNIX Port Model

the source computer include extra bits that encode the message type, protocol used, etc. The destination computer uses this extra information to guide the processing. Figure 22 illustrates the process of demultiplexing an Ethernet frame up to the application level.

5.7 Internet Protocol

The Internet Protocol (IP) is used by all high-level protocols, like TCP and UDP, in the Internet Protocol suite to transfer data. IP provides an unreliable, connectionless, best effort datagram delivery service.

Unreliable means that IP datagrams can get lost, duplicated, or arrive in a different order than they were sent. No connection has to be established before a datagram can be sent to a destination.

The header of every IP datagram, shown in Figure 23, consists of at least 20 bytes.

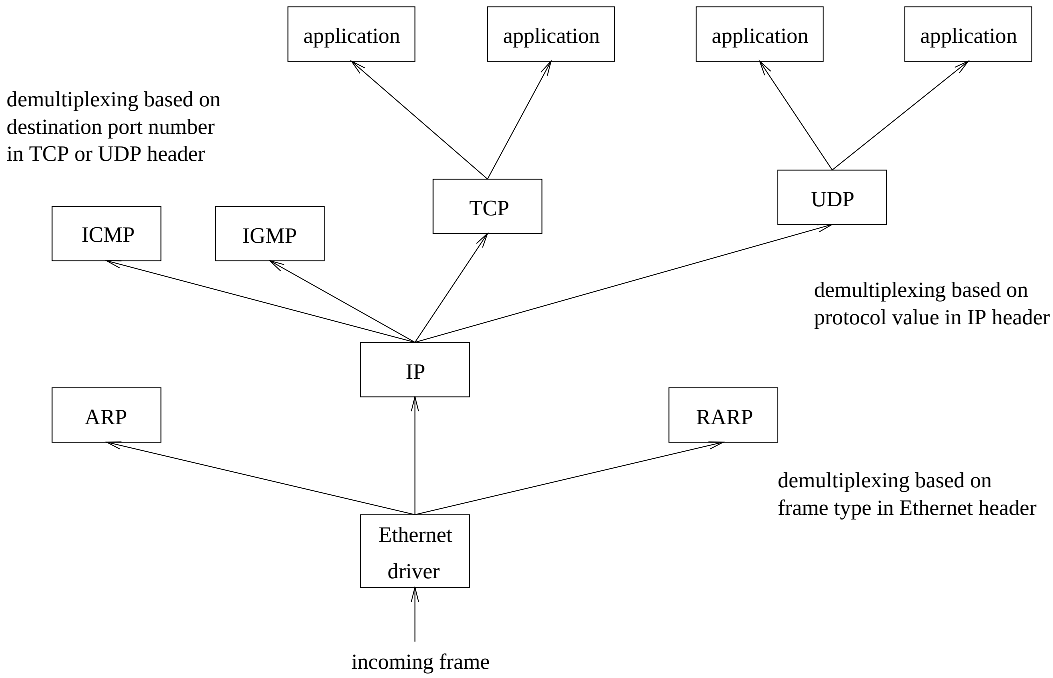

Figure 22: The Demultiplexing of a received Ethernet Frame

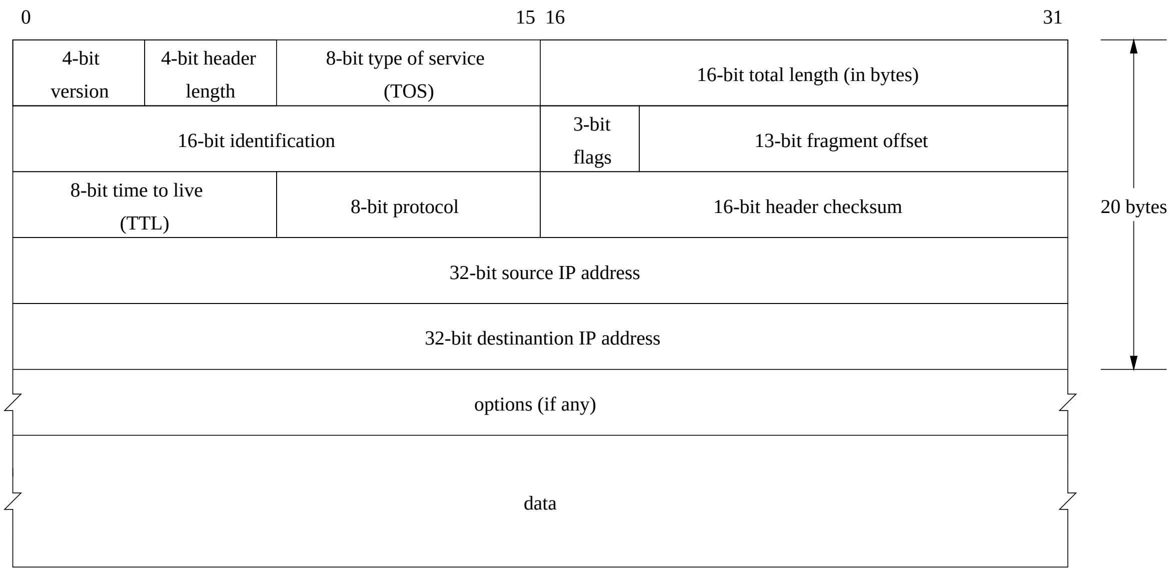

Figure 23: IP Datagram, showing the Fields in the IP Header [Stev94]

The current IP *version* is 4, version 6 is under development. Changes between version 4 and 6 are summarized in Chapter 9.4.1. The field *header length* gives the length of the IP header in 32-bit words. The *type of service* field indicates whether this datagram requests special handling or not. The field *total length* states the length of the IP datagram including the header. Therefore the maximum data size of an IP datagram is 65515 bytes. Most link layers can not handle such big packets. IP automatically fragments datagrams if necessary. The *identification* field uniquely identifies each datagram sent by a host. The *flags* and fragment offset field are used for fragmentation of large datagrams. The *time to live* field limits the number of routers through which a datagram can pass. The *protocol* field specifies which type of data the datagram carries (e.g. UDP or TCP), this is needed for demultiplexing IP datagrams (see Chapter 5.6). The *header checksum* is a checksum over the IP header. *Source IP address* states the origin of an IP datagram, *destination IP address* states where it should be sent to. Various *options* can be specified.

The internet protocol is specified in [RFC 791].

5.8 User Datagram Protocol

The User Datagram Protocol (UDP), Figure 24, enhances IP datagrams in such a way that they become usable for processes: *port numbers* are included which allows identification of the sending and the receiving process. The UDP optional checksum covers the UDP header, some fields of the IP header, and the data. The *length* field states the number of bytes in the UDP datagram including the header.

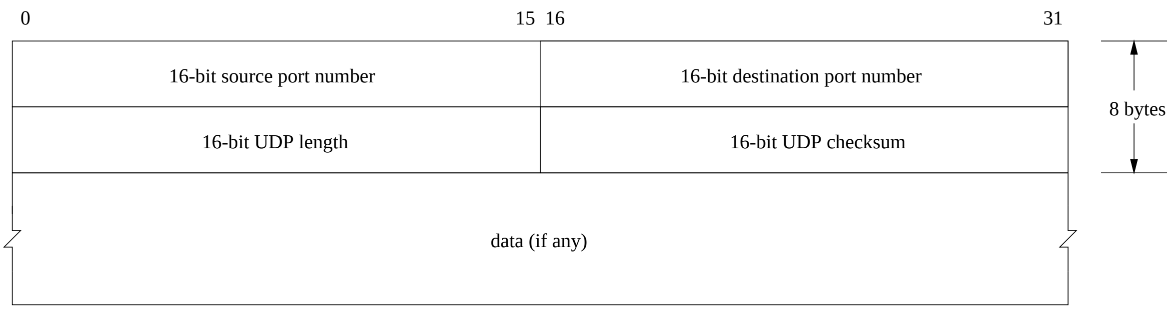

Figure 24: UDP Datagram, showing the Fields in the UDP Header [Stev94]

UDP is just an interface to make IP datagrams usable from different processes. Therefore UDP is as unreliable as IP, there is no guarantee that UDP datagrams reach their destination. UDP is specified in [RFC 768].

5.9 Transmission Control Protocol

The Transmission Control Protocol, or short TCP, provides a connection-oriented, full-duplex, reliable, byte stream service. TCP uses IP as data exchange protocol. Before two applications can transfer data via TCP, a connection has to be established. With TCP message boundaries not being visible to the application, applications use TCP data streams like normal UNIX I/O streams. TCP provides reliability by doing the following:

- Data from the applications is broken down by TCP into different *segments* of data which are sent.
- TCP maintains a timer for sent segments. If an acknowledgment is not received in time, the segment is retransmitted. Chapter 5.10.2 investigates the algorithm TCP uses for timeout and retransmission.
- Incoming data is acknowledged.
- A checksum ensures that no corrupted data is delivered to the application. Corrupted segments are discarded, the sender is expected to timeout and retransmit the segment in question.
- As TCP uses IP datagrams to exchange data, and IP datagrams can arrive out of order, TCP resequences the data if necessary.
- TCP detects and discards data caused by duplicated IP datagrams.
- Flow control is handled by TCP: each communication endpoint has a finite amount of data which it is allowed to send till further permission of the other side (a so-called "window").

The TCP header, shown in Figure 25, consists of at least 20 bytes. Like UDP datagrams, TCP segments contain source and destination *port numbers* to identify the sending and receiving process. Together with the source and destination IP address they uniquely specify each connection. The *sequence number* is used to identify the position of the data in the segment. The *acknowledgment number* contains the next sequence number that the sender of the acknowledgment expects to receive. The *header length* field gives the length of the header in 32-bit words. Various flags are used for connection establishment and control status information. The *window size* field is used to announce the number of bytes the sender of the segment is willing to accept. The *checksum* covers the TCP header, some parts of the IP header, and the TCP data. The *urgent pointer* together with one flag can be used to advertise data that should be delivered to the receiving process as fast as possible. Some *options* may be used.

TCP is specified in [RFC 793].

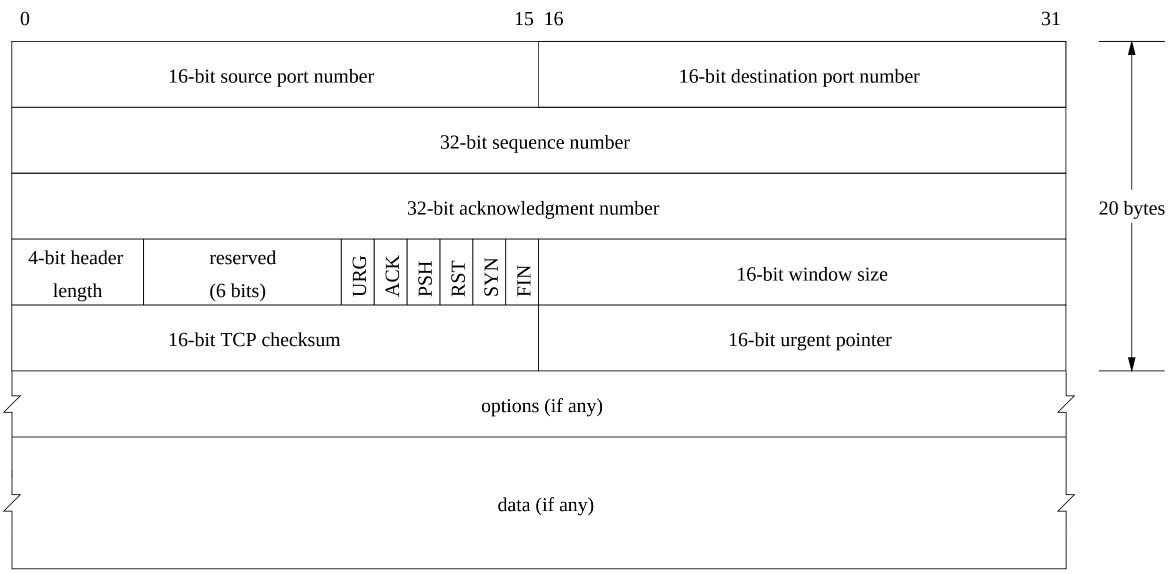

Figure 25: TCP Segment, showing the Fields in the TCP Header [Stev94]

5.10 Reliability

Two aspects of how TCP ensures reliability are examined more closely: the TCP checksum and timeout algorithm.

5.10.1 Checksums

For the purpose of data communication, the goal of a checksum algorithm is to balance the effectiveness at detecting errors with the cost of computing the check values [Part95].

IP, TCP, and UDP use a checksum that detects corrupted messages. The checksum covers the header, and with TCP and UDP also the transferred data. The so-called TCP checksum is a 16-bit ones complement sum of the data. This sum will catch any 1 bit error in the data, and over uniformly distributed values of data it is expected to detect other types of errors at a rate proportional to 1 in 2^{16} [RFC 1071].

When the checksum is calculated for TCP and UDP a so-called *pseudo header* (various parts from the IP header) is included in the calculation.

The TCP checksum can be computed very quickly. Tests have shown that it is good enough for most purposes, although the checksum has a major limitation: regardless of the order in which the 16-bit values appear, the sum stays the same.

Details about the TCP checksum can be found in [RFC 1071].

5.10.2 Handling of Timeouts

Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections the retransmission timeout must be dynamically determined [RFC 793]. There also is a large variability of delays in different routers. Therefore an *adaptive retransmission algorithm* is used.

TCP monitors the performance of every connection, and tries to deduce reasonable values for timeouts. TCP records the time for every outgoing segment and the time when an acknowledgment for that segment arrives. From these two time stamps the *round trip time* RTT (the difference between the two times) is calculated. This value is then used to slowly adjust an estimated *smoothed round trip time*, SRTT, a weighted average:

$$\mathrm{SRTT}_{\mathrm{new}} = (\alpha \times \mathrm{SRTT}_{\mathrm{old}}) + ((1 - \alpha) \times \mathrm{RTT})$$

If the chosen α is close to 1, the weighted average is nearly immune to changes that last a short time like a single segment that encounters long delay. If the value for α is chosen close to 0, the weighted average responds quickly to delay changes.

The *retransmission timeout* RTO is then calculated as a function of the current round trip estimate. In [RFC 793] the function

$$\mathrm{RTO} = min(UBOUND, max(LBOUND, (\beta \times \mathrm{SRTT})))$$

is suggested where *UBOUND* is an upper bound on the timeout (e.g., 1 minute), *LBOUND* a lower bound on the timeout (e.g., 1 second), and β a delay variance factor (e.g., 1.3 to 2.0). Practice has shown that this function does not always have the desired result. Therefore more complicated algorithms and functions were developed that behave better in most situations. These algorithms are, unfortunately, too long to be covered in this document. The interested reader might refer to [Com95, page 209-216].

5.11 Application Level Services

The TCP/IP protocol suite includes protocols for file transfer, remote login facilities and other purposes.

5.11.1 FTP

The File Transfer Protocol (FTP) is a standard protocol which allows computers connected by TCP/IP to exchange data. FTP offers some facilities beyond the transfer function itself:

- interactive access
- format (representation) specification
- authentication control

Details about the FTP protocol can be found in [Com95, Chapter 24].

5.11.2 TELNET

The TELNET protocol provides a simple remote terminal protocol. It allows a user to sit on one computer and work remotely on another computer. Details about the TELNET protocol can be found in [Com95, page 408ff].

5.11.3 Rlogin

`rlogin` is a UNIX remote login service (derived from BSD UNIX). It is more powerful than telnet, but is limited to UNIX. `rlogin` exports part of the users environment to the remote machine, including information like the terminal the user is using. Remote login sessions appear to a user almost exactly as a normal local login session.

5.12 UNIX Network Tools

There are some basic network utilities offering low-level network management within UNIX. The most common ones are described in Table 13.

5.13 Summary

This chapter has examined the protocol normally used for UNIX interprocessor communication. The next chapter can now describe how the TCP/IP protocol suite can be used within programs to exchange data.

The TCP/IP protocol suite is one of the most used protocol suites. IP is the basic protocol for data exchange. For applications UDP and TCP are of interest. UDP provides an unreliable, connectionless datagram service, while TCP provides a reliable, full-duplex, connection-oriented byte stream service. Table 14 summarizes some important issues of IP, UDP, and TCP.

Program	**Purpose**
`dnsdomainname`	show the system's domain name
`hostname`	show or set the system's host name
`netstat`	display active network connections
`nslookup`	query Internet name servers interactively
`ping`	checks connection to destination address by sending ICMP ECHO_REQUEST packets
`rdist`	remote file distribution client program
`traceroute`	print the route packets take to network host

Table 13: UNIX Network Tools

Issue	**IP**	**UDP**	**TCP**
connection-oriented	no	no	yes
message boundaries	yes	yes	no
port numbers	no	yes	yes
data checksum	no	opt.	yes
positive ack.	no	no	yes
timeout and retransmit	no	no	yes
duplicate detection	no	no	yes
sequencing	no	no	yes
flow control	no	no	yes

Table 14: Comparison of Protocol Features for IP, UDP, and TCP

6 Distributed IPC

This chapter covers the UNIX communication facilities for distributed interprocess communication. Two APIs are introduced:

1. **the Socket Interface**
2. **Transport Layer Interface**

Different computers normally make use of networks and protocols to exchange information, as they do not share memory. This chapter describes two application program interfaces (APIs) to communication protocols: the socket interface and the Transport Layer Interface. The TCP/IP protocol suite is used as an example where appropriate.

6.1 Sockets

The socket interface to networking was developed by the University of California in Berkeley. It was introduced with 4.2BSD. The basic idea of the socket interface was to make interprocess communication similar to file I/O.

However, network protocols are more complex than conventional I/O devices. Therefore interaction between user processes and network protocols has to be more complex than interaction between user processes and conventional I/O facilities. Normal I/O does not allow server code that awaits connections passively and client code that forms connections actively. Also there was no mechanism for sending datagrams to different destinations: the `open()` system call always bounds a fixed file name to a descriptor.

The basic building block for communication is the *socket*, an endpoint of communication. A name may be bound to a socket. Sockets can have different *types*. A socket can be associated with one or more processes. A socket exists within a *communication domain*, an abstraction to bundle properties of processes communicating through sockets. Data normally can only be exchanged with sockets in the same domain.

6.1.1 Supported Protocols

The socket interface supports different communication protocols. The *UNIX domain* protocol (used for interprocess communication between processes on the same computer) and the *Internet domain* protocol (used for interprocess communication between processes on either the same or different computers) are described.

6.1.2 Connections and Associations

The term *connection* is used in this chapter for a communication link between two processes. The term *association* is used for an n-tuple that completely specifies the two endpoints of communication that make up a connection. For connections which use the TCP/IP protocol suite this n-tuple is the 5-tuple
{protocol, local IP address, local port number, foreign IP address, foreign port number}
For the UNIX domain the following 3-tuple specifies an association:

{protocol, local path, foreign path}

All members of an n-tuple have to be specified by both processes wishing to communicate before data can be exchanged. Different system calls fill in different parts of the n-tuples.

6.1.3 Asymmetry

The typical client/server relationship, described in Chapter 2.2.2, is not symmetrical. Initiating a network connection requires that both programs know which role, client or server, they are to play. This is reflected in the design of the socket interface. Some system calls are only useful for the clients, while others are only useful for servers.

6.1.4 Socket Types

A socket has a specific type which influences its communication properties. Processes are presumed to communicate only between sockets of the same type. Four different types of sockets are available: stream sockets, datagram sockets, raw sockets, and sequenced packet sockets.

A *stream* socket provides bi-directional, reliable, sequenced, unduplicated flow of data. Message boundaries are not visible. A *datagram* socket supports bi-directional flow of data. Record boundaries are visible. Messages may be duplicated, lost, or arrive in a different order than they were sent. A *raw* socket allows user processes to access the underlying communication protocols. Raw sockets are not intended to be used by normal applications and therefore not described further. A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. Sequenced packet sockets are not supported by the Internet domain, therefore they are also not discussed further.

6.1.5 Basic Socket Function Calls

Socket Creation A socket is created with the

`int socket(int *domain*, int *type*, int *protocol*)`

system call. The *domain* parameter specifies the communication domain within which communication will take place; this selects the address or protocol family which should be used. For the UNIX domain `AF_UNIX` (`AF` for address family) has to be specified, for the Internet protocols `AF_INET`. The *type* indicates the socket type as stated above. For stream sockets `SOCK_STREAM` and for datagram sockets `SOCK_DGRAM` has to be specified. The *protocol* parameter can select a specific protocol. Normally `0` is used to selects the default protocol. With the internet protocols a stream socket uses TCP, and UDP is used for a datagram socket. The returned integer is a new socket descriptor which is needed for further access to the socket. UNIX allocates socket descriptors and normal file descriptors from the same set of integers. This is part of the goal to make networking as similar to standard file I/O as possible. Sockets perform exactly like UNIX files or devices where appropriate. The `socket()` system call specifies the *protocol* in the association n-tuple.

Sockets for the UNIX domain can also be created with the

`int socketpair(int *domain*, int *type*, int *protocol*, int *sd*[2])`

system call. It creates two sockets simultaneously and places the two socket descriptors in the two elements of *sd*. This call is the socket equivalent of the `pipe()` system call. The first three parameters are equal to the ones in the `socket()` system call.

Binding The local part of an association (*local address, local port number* for the Internet domain, and *local pathname* for the UNIX domain) has to be specified before a connection can be established. This is normally done by the

`int bind(int *sockfd*, struct sockaddr **localaddr*, int *addrlen*)`

system call. Under certain circumstances the `send()` and `connect()` system calls (described later) automatically bind a socket. The pointer `sockaddr` points to the needed data to fill in the local part of an association. For the Internet domain this is a pointer to a `struct sockaddr_in`, and for the UNIX domain to a `struct sockaddr_un`. As the different domains use structures of different sizes, the length of the used structure has to be specified in *addrlen*. The structure `sockaddr_in` and `sockaddr_un` is defined like this:

```
struct sockaddr_in {
  short int          sin_family;  /* Address family (AF_INET)            */
  unsigned short int sin_port;    /* Port number, network byte order     */
```

```
  struct in_addr     sin_addr;    /* Internet address, network byte order */
  char               sin_zero[8]; /* unused                               */
};

struct sockaddr_un {
  unsigned short sun_family;      /* Address family (AF_UNIX)             */
  char sun_path[108];             /* pathname                             */
};
```

There are still unspecified elements of the association: the *foreign address* and *foreign port number* for the Internet domain, and *foreign pathname* for the UNIX domain. These are filled in differently for connection-oriented and connectionless communication.

Socket Termination When a process finishes using a socket, it calls the normal `close()` system call. If the process was the last one using this socket, the socket is discarded.

6.1.6 Connection-oriented Communication

Figure 26 gives an overview of system calls used for client/server interaction using a connection-oriented protocol.

Connection Establishment is usually asymmetrical, with one process being the client and the other the server. The **server** first binds to its well-known port number as described above and then passively "listens" on its socket with the

`int listen(int sd, int backlog)`

system call. This puts the socket in a passive mode ready to accept incoming connections. An incoming connection is then accepted with the

`int accept(int sd, struct sockaddr *addr, int *addrlen)`

system call, which finally fills in the foreign parts of the n-tuple. The argument ***addr*** is a result parameter that is filled in with the address of the client. The ***addrlen*** is a value-result parameter; it should initially contain the amount of space pointed to by ***addr***. On return it will contain the actual length (in bytes) of the address returned. This information can be used to verify that the server is willing to serve the client. `accept()` creates a new socket with the same properties of ***sd*** and allocates a new socket descriptor for the socket. `accept()` blocks until a connection is present. The original socket ***sd*** remains open and may be used to accept the next connection. The

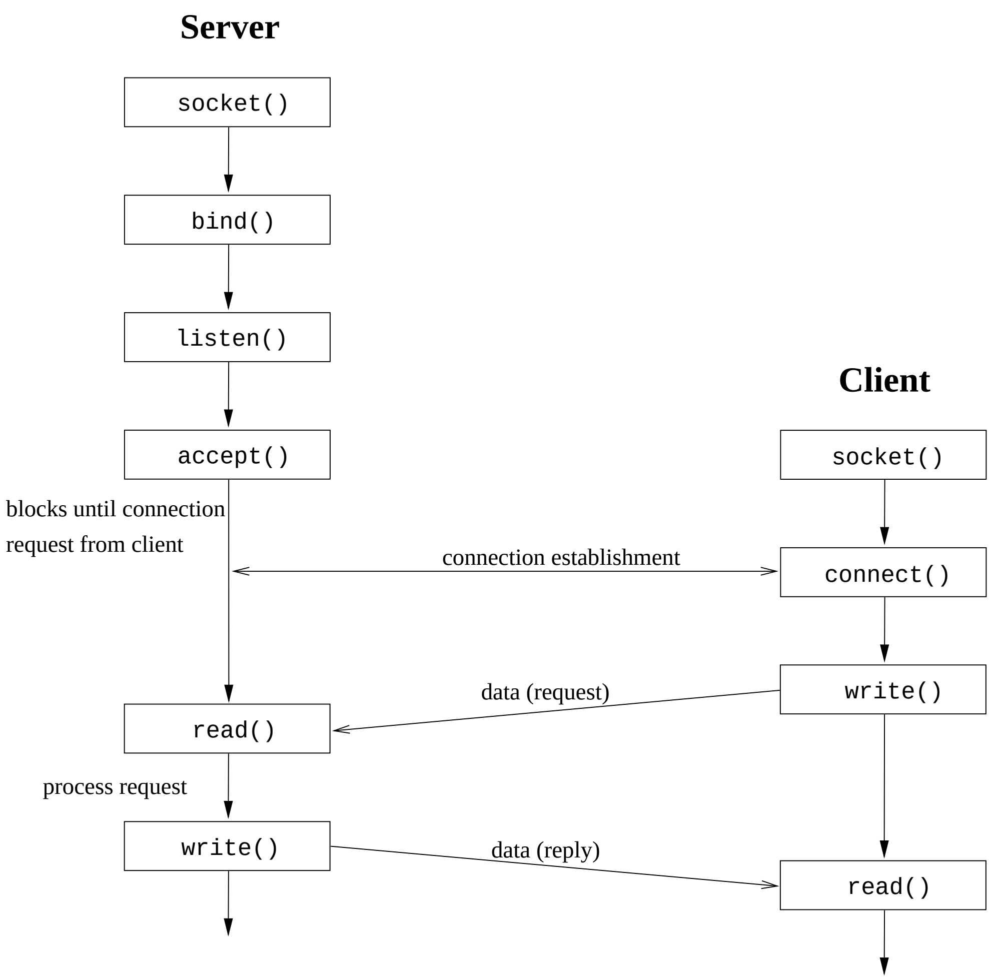

Figure 26: Socket System Calls, Connection-Oriented Protocol [Stev90]

newly created socket descriptor, which now can be used for data transfer, is returned.

The **client** calls

`int connect(int` *`sd`*`, struct sockaddr *`*`serv_addr`*`, int` *`addrlen`* `)`

after creating a socket to bind (to a temporary port number in the Internet domain) and to establish a connection with the server stated in *`serv_addr`*. All parts of the association n-tuple are now filled in and data exchange can start.

Data Exchange can be performed with the normal `read()`, `readv()`, `write()`, and `writev()` system calls. The new system calls `send()` and `recv()` may also be used. They are similar to `readv()`/`writev()`, but allow specification of a special flag. This can be used to send or receive *out of band* data, to look at data without reading it, or other purposes.

Examples The client/server pair of programs `distSOCKET/tcpclient.c` (page 197) and `distSOCKET/tcpserver.c` (page 200) demonstrate how to exchange data using a stream socket with TCP.

6.1.7 Connectionless Communication

Figure 27 gives an overview of system calls used for client/server interaction using a connectionless protocol.

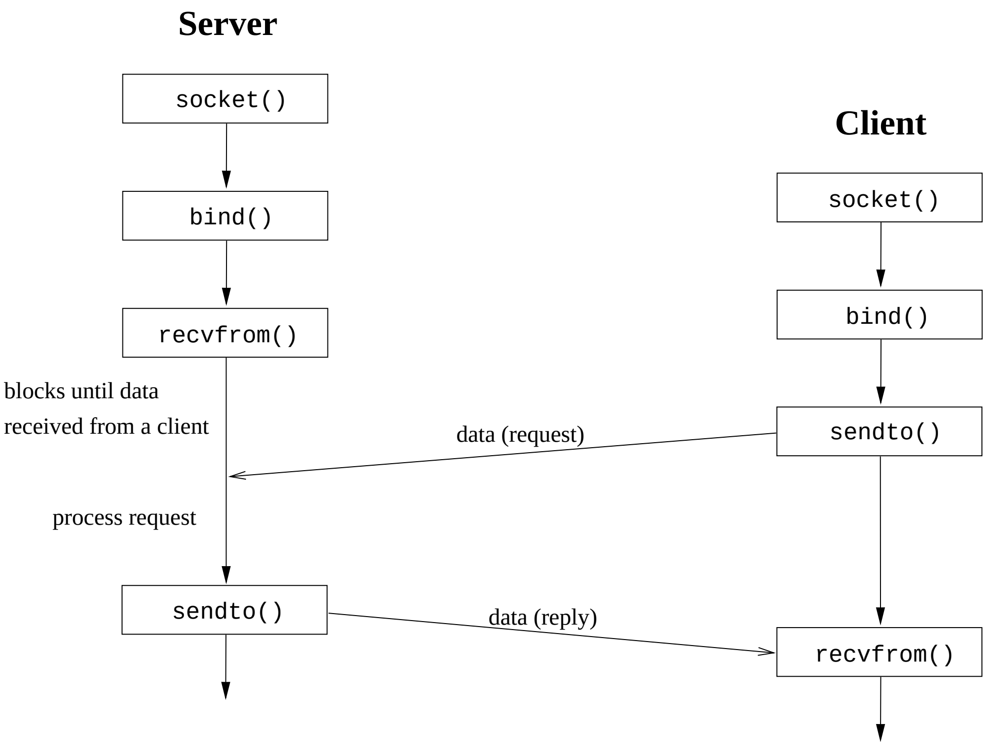

Figure 27: Socket System Calls, Connectionless Protocol [Stev90]

Connection Establishment is performed on a per datagram basis. A connection only exists for one datagram. Both server and client call `bind()` to set their local address in their association n-tuple after creating a socket. Note that this procedure is symmetrical.

Data Exchange Data can be exchanged in both directions with the new `sendto()` and `recvfrom()` system calls. For each datagram the destination has to be stated. If several messages are to be sent to the same destination, the `connect()` system call may be used to register the destination. Data is then sent to the registered destination with the normal `write()` and `writev()` system calls. Also, only datagrams from this address will be received by the socket.

Examples The client/server pair of programs `distSOCKET/udpclient.c` (page 191) and `distSOCKET/udpserver.c` (page 194) demonstrate how to exchange data using a datagram socket with UDP.

6.1.8 Options

For many applications the need arises to control the socket in more detail. The application might change a timeout parameter, or change the size of a buffer. The system call `getsockopt()` allows the application to request information about a specific socket, while the system call `setsockopt()` allows an application to set a socket option.

6.1.9 Obtaining Local and Remote Socket Addresses

A process might need to determine the local address of a socket or the destination address to which a socket connects. This can be performed with the system calls `getpeername()` and `getsocketname()`.

6.1.10 Obtaining and Setting Host Name and Domain

The `gethostname()` system call allows user processes to access the host name. The host name can be set by a privileged process with `sethostname()`. The host domain may be obtained with the `getdomainname()` and can be changed by privileged user processes with the `setdomainname()` system call.

6.1.11 Library Functions

A number of library functions are available that handle

- mapping host names to network addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers.

These functions are summarized in Table 15.

In addition to these general protocol independent library functions Internet domain specific functions are available. Table 16 summarizes functions which convert data from and to network byte order. Table 17 summarizes the internet address manipulation functions.

Library Call	**Purpose**
`gethostbyname()`	obtains network host entry for a host by its name
`gethostbyaddr()`	obtains network host entry for a host by its address
`getnetbyname()`	obtains network net entry by its name
`getnetbyaddr()`	obtains network net entry by its address
`getprotobyname()`	obtains protocol entry by its name
`getprotobynumber()`	obtains protocol entry by its number
`getservbyname()`	obtains service entry by its name
`getservbyproto()`	obtains service entry by its port number

Table 15: Library Calls for Database Services

Library Call	**Purpose**
`htonl(val)`	convert 32-bit quantity from host to network byte order
`htons(val)`	convert 16-bit quantity from host to network byte order
`ntohl(val)`	convert 32-bit quantity from network to host byte order
`ntohs(val)`	convert 16-bit quantity from network to host byte order

Table 16: Library Calls for converting from/to Network Byte Order

Details about these functions can be found in the corresponding manual pages.

6.1.12 Socket Summary

The BSD interprocess communication facilities have become the *de facto* standard for UNIX, as they are available in nearly all UNIX derivatives. The socket interface is nowadays also used on other operating systems, e.g Microsoft Windows.

Sockets were created to make interprocess communication look like standard UNIX file I/O. The socket function calls are based on the client/server model. Different "domains" exist which support local IPC or distributed IPC. The socket interface allows different types of sockets for different purposes.

Table 18 summarizes which system calls change fields in an association. Figure 28 summarizes how data flows from one process over a socket descriptor via TCP, IP, Ethernet to another process.

Library Call	Purpose
`inet_addr(cp)`	converts the Internet host address `cp` from dotted decimal notation into binary data in network byte order
`inet_network(cp)`	extract the network number in network byte order from the address `cp` in dotted decimal notation
`inet_ntoa(in)`	converts the Internet host address `in` given in network byte order to a string in dotted decimal notation
`inet_makeaddr(net, host)`	makes an Internet host address in network byte order by combining the network number `net` with the local address `host` in network `net`, both in local host byte order.
`inet_lnaof(in)`	returns the local host address part of the Internet address `in`. The local host address is returned in local host byte order
`inet_netof(in)`	returns the network number part of the Internet Address `in`. The network number is returned in local host byte order

Table 17: Internet Address Manipulation Library Routines

Function Call	Protocol	l. Addr.	l. Port-Nr.	f. Addr.	f. Port-Nr.
`socket()`	•				
`bind()`		•	•		
`accept()`				•	•
`recvfrom()`				•	•
`sendto()`				•	•

Table 18: System Calls which fill in Parts of an Association (Internet domain 5-tuple assumed)

6.2 Transport Layer Interface

The Transport Layer Interface (TLI) was introduced with Release 3 of System V in 1986. The TLI is designed after the ISO Transport Service Definition. It is based on System V STREAMS (the principles of System V STREAMS are summarized in

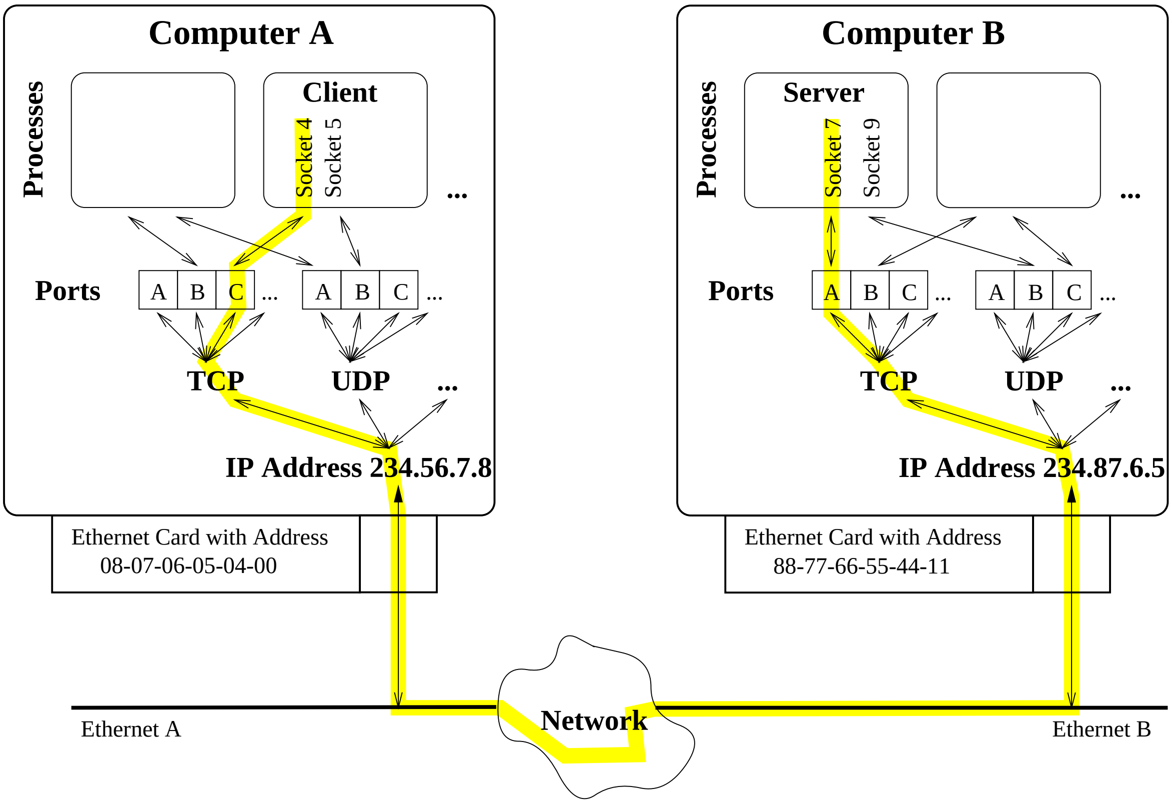

Figure 28: How Ports, Addresses, and Sockets fit together. The shaded path indicates the flow of data between the computers.

Chapter 9.2.2). Normal UNIX descriptors are used to refer to network data. As the TLI is similar to the socket interface, only the more important differences are stated.

The Transport Layer Interface is a collection of library routines, not system calls. Therefore if the TLI is used, a special library has to be linked to the program. The function names and functionality of the TLI are similar to the socket interface (with an additional `t_`). The number of options for TLI functions is much higher than in the socket interface. This allows more flexibility, but prevents a detailed description in this document.

Figures 29 and 30 give an overview of the function calls used for an iterative server.

Many of the data structures passed between the user code and the TLI functions are structures. Most of these structures contain one or more `netbuf` structures, each of which contains a pointer to a buffer used for a special purpose. The functions `t_alloc()` and `t_free()` simplify the dynamic allocation of these structures.

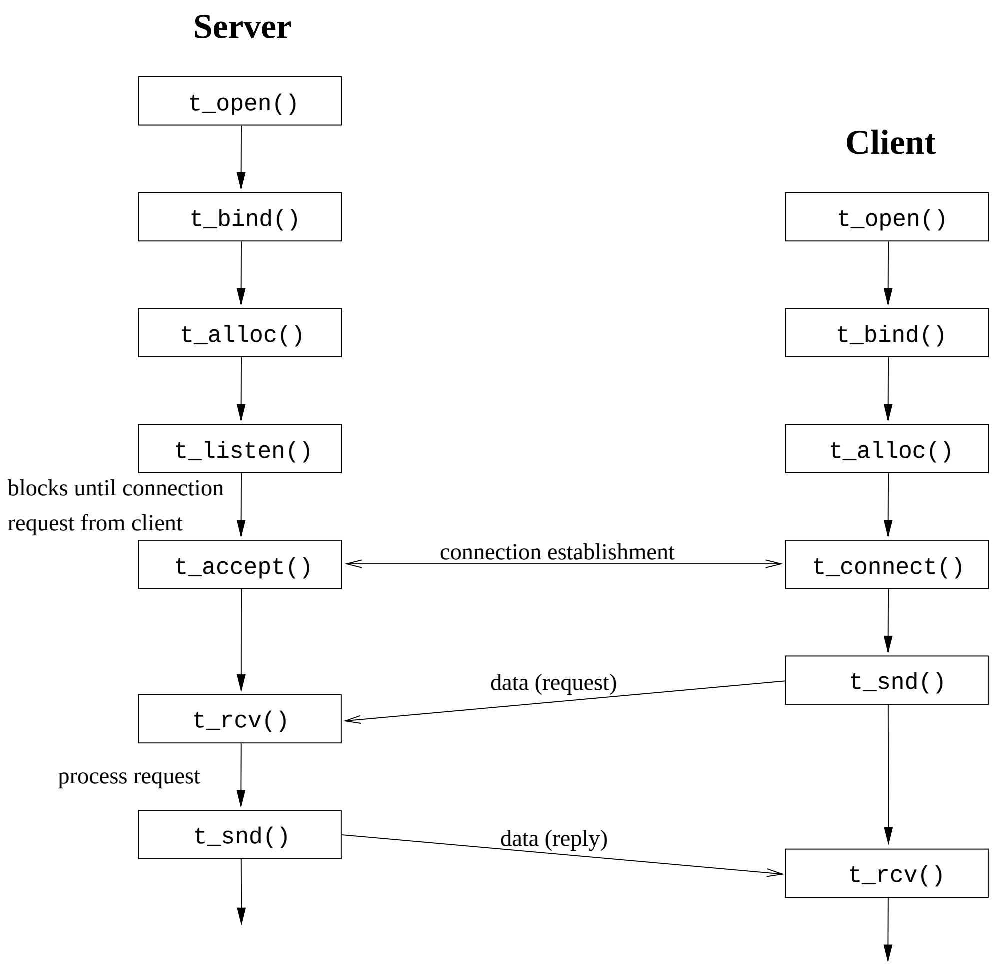

Figure 29: TLI Function Calls, Connection-Oriented Protocol [Stev90]

The `t_listen()` function does not accept an incoming connection. This has to be done with `t_accept()`. The `t_accept()` function call does not create a new *transport endpoint* (the equivalent to a socket). For a new file descriptor, needed by a concurrent server, the functions `t_open()`, and `t_bind()` have to be called.

The functions `t_sndudata()` and `t_rcvudata()` correspond to the socket system calls `sendtto()` and `recvfrom()`.

The example programs `distTLI/udpclient.c` (page 217) and `distTLI/udpserver.c` (page 220) demonstrate how to use TLI with UDP. The UDP server is implemented as an iterative server. The use of TCP with TLI is demonstrated with the example programs `distTLI/tcpclient.c` (page 224) and `distTLI/tcpserver.c` (page 227). As TCP connections might exist for a longer time the TCP server is implemented in a concurrent fashion.

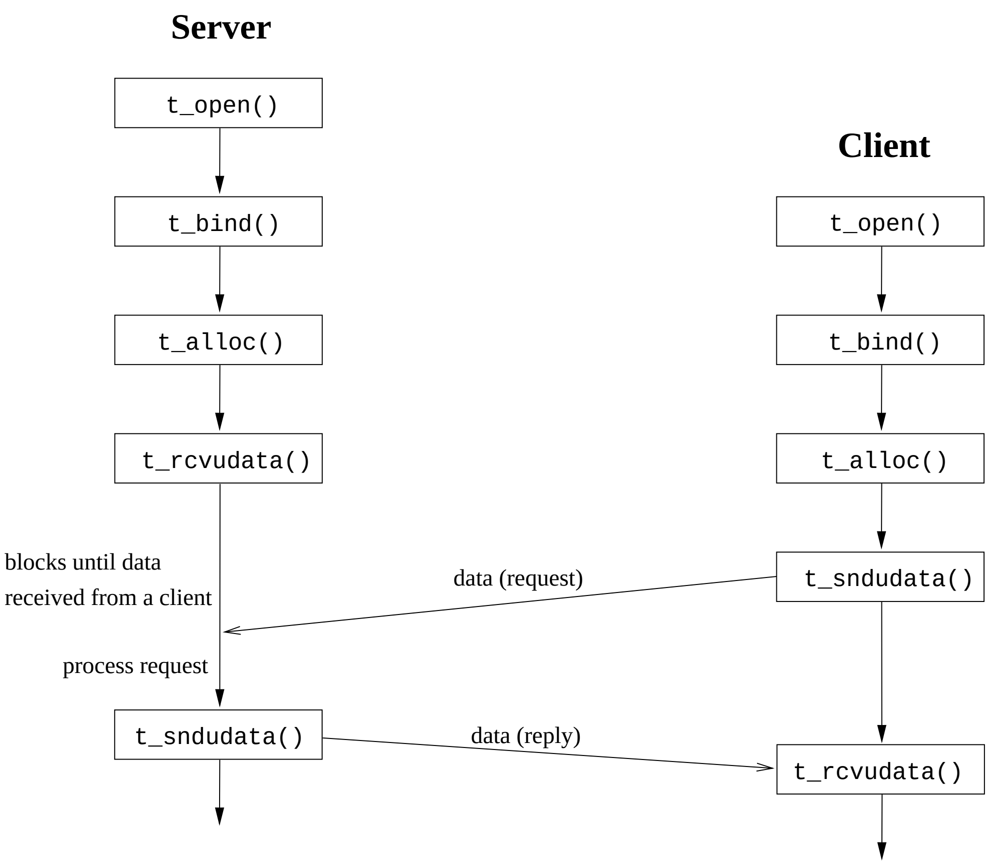

Figure 30: TLI Function Calls, Connectionless Protocol [Stev90]

6.3 Remote Procedure Calls

Remote Procedure Calls (RPCs) are a higher level communication abstraction which make use of the distributed UNIX communication facilities described above. As remote procedure calls are becoming more and more important, and RPCs are a basic functionality in most modern operating systems, the principles of RPCs are sketched.

The idea behind remote procedure calls is to hide the underlying I/O (message passing...) communication for invoking a service on a different computer. The call of a function or procedure on a different computer should be as easy and transparent as a local procedure call; the programmer should not be concerned *where* a function is executed.

To archive that goal, special programs are used to create code that handles the necessary message exchange: the client and the server *stub code*. The client stub code packs the arguments into a message (called *parameter marshalling*), sends the message, waits for the result message, unpacks the result and returns the result to the caller. The server stub code unpacks the parameters, calls the right function, packs the result into a message and sends a message back. Figure 31 (from [Tan92]) illustrates this.

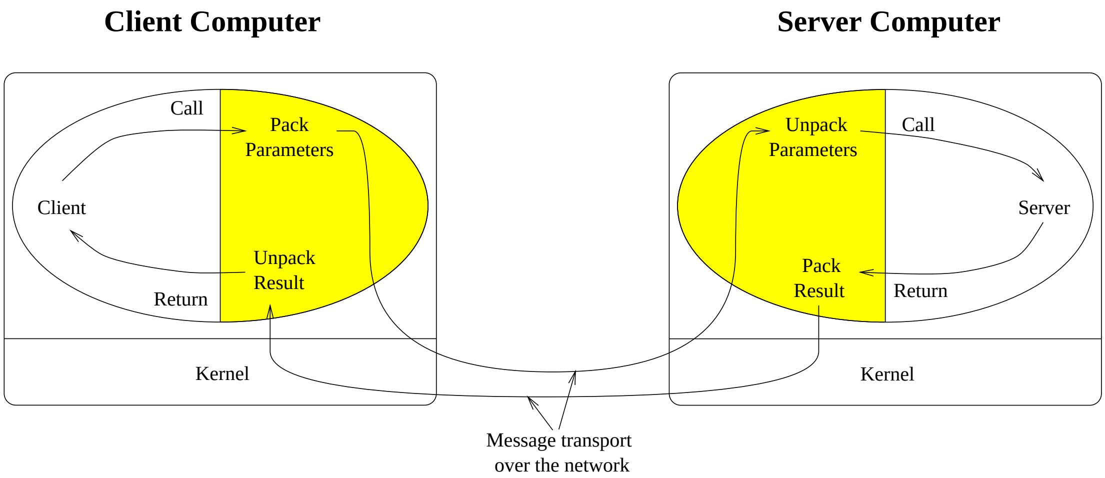

Figure 31: Calls and Messages in an RPC. Each ellipse represents a single process, with the shaded portion being the stub.

RPC should be as transparent as possible. To achieve this goal the following issues have to be considered:

- *Parameter passing:* only call-by-value parameters can be supported directly. Call-by-reference parameters (using pointers) introduces problems which can not be solved easily.
- *Binding:* how to find a remote host for a desired service, and how to find the correct server process on that host.
- *Transport Protocol:* which protocol should be used for connecting to the server.
- *Exception handling:* what should be done if something goes wrong in the remote procedure call.
- *Call semantics:* often it is difficult to ensure that a RPC is guaranteed to be executed only once (the message might be delivered more than once, what happens if the server crashes while the RPC is executed...). Therefore the exact semantics of an RPC are important.
- *Data representation:* the binary format of data might be different (floating point representation, big vs. small endian...). Therefore there might be a need for data conversion.

- *Performance:* what are the overheads of RPCs, and how can these overheads be kept to a minimum?
- *Security*

With remote procedure calls the development of distributed applications is much easier.

Sun RPC

The Sun RPC protocol, defined in [RFC 1057] is common in the UNIX environment. The program `rpcgen`(1) is used to generate the appropriate client and server C-stub code. XDR (eXternal Data Representation, defined in [RFC 1057]) is used for the representation of data. UDP and TCP can be used as network protocols. The `portmapper` daemon handles all binding issues.

The principle of remote procedure calls is described in more detail in [Tan92, Chapter 10.3], while existing implementations are covered in [Stev90, Chapter 18]. A modified RPC example from [Stev90] is included in Appendix A.7 (page 306). An even more powerful method for developing distributed applications is the use of the Arjuna system.

6.4 Summary

Distributed interprocess communication is far more complex than local interprocess communication. Details about the underlying network protocols have to be considered. The number of system and library calls that are needed for distributed IPC is much higher than for local IPC.

Two different APIs for distributed IPC have been described: the socket interface and Transport Layer Interface. Sockets are widely used, although TLI provides more flexibility.

A high-level use of networking facilities, remote procedure calls, can make the development of distributed applications easier.

Distributed IPC is more general than local IPC, as it can be used on one computer, but also between distinct computers. Table 19 compares the function calls for some local and distributed IPC facilities used for connection-oriented communication. Chapter 7 compares the performance of local IPC and distributed IPC facilities.

		Sockets	TLI	Messages	FIFOs
Server	allocate space		t_alloc()		
	create endpoint	socket()	t_open()	msgget()	mknod() open()
	bind address	bind()	t_bind()		
	specify queue	listen()			
	wait for connection	accept()	t_listen()		
	get new fd		t_open() t_bind() t_connect()		
Client	allocate space		t_alloc()		
	create endpoint	socket()	t_open()	msgget()	open()
	bind address	bind()	t_bind()		
	connect to server	connect()	t_connect()		
	transfer data	read() write() recv() send()	read() write() t_rcv() t_snd()	msgrcv() msgsnd()	 read() write()
	datagrams	recvfrom() sendto()	t_rcvudata() t_sndudata()		
	terminate	close() shutdown()	t_close() t_sndrel() t_snddis()	msgctl()	close() unlink()

Table 19: Comparison of Socket, TLI, Message Queues, and FIFO Function Calls [Stev90]

7 Performance

This chapter

- **describes how the performance of the different interprocess communication facilities can be measured**
- **compares the performance of the different approaches**
- **describes the usability of the different approaches**

So far many different interprocess communication facilities have been described. How fast are they? From the results of performance measurements derived for this chapter conclusions can be drawn which communication facilities to use in different situations.

A large number of programs were written for all described IPC facilities which find out different aspects of their performance.

All performance measurements were done on three different systems to allow more general conclusions:

- SparcCenter 2000 (multi-user timesharing computer, six TMS390Z50 processors), with the operating system Solaris 2.5 (SunOS 5.5), which is based on BSD UNIX. The SparcCenter 2000 computer used in the University of Newcastle upon Tyne has the name `aidan`.
- HP-9000/710 (Processor: PA 7000/50 MHz) computer systems running HP-UX 9.01, which is based on UNIX System V release 3.0 (but has included all important features from other UNIX flavours). The used HP-9000/710 are named `pike`*xy*.
- PC (i586/90 MHz) running Linux 1.2.9. Linux has the more common features of System V and BSD. This computer was not connected to the other computer systems.

The performance of all IPC facilities is measured with different message/buffer sizes (where appropriate) from 1 byte doubling up to the default limit of the particular interprocess facility. This limit is sometimes different on the three systems. Where limits are huge, an appropriate value is chosen.

The performance graphs are labeled double-logarithmic for two reasons:

- the possibility to compare performance graphs for the three different systems in one graph
- the ability to compare results for small buffer sizes.

Please note that even if lines in double-logarithmic graphs are quite close this means a big difference in absolute values.

This chapter summarizes the results derived from the various performance measurement programs. The Appendix of the WWW-Version of this document includes pointers to the generated raw results.

Interesting performance measurement outcomes are:

1. how long it takes to establish a connection
2. how many bytes can be transferred in a certain time (note that this is nearly equal to how long the data is in transit ("travels"))
3. the time needed for a specific operation (this is equivalent to the inverse of how many operations of a certain type can be done in a certain time)

Number 1 is only interesting for interprocess-communication facilities where a connection has to be established. Only the TCP connection example has this property. Possible similar measurements for filesystem dependent IPC facilities were not done for the following reason: the speed of these is highly dependent on the file system (and not of the operating system), and these are distributed for `aidan` and all `pike`s. This would have had an unpredictable impact on the results.

Number 2 is interesting for all examples that are able to transfer larger amounts of data. Most of the interprocess examples include this property. The result is measured in Kb (1024 bytes) per second.

Number 3 is useful for nearly all examples. The resulting graphs show how much non-avoidable overhead is involved with a certain operation. The results are given in μs per operation (where appropriate for a specific buffer size). Some other results are also given in operations per second.

For all measurements passed clock time is used to display the results, as this is the time the user/program becomes aware of. If CPU time is measured, accumulated system and user CPU time is given.

7.1 Measuring Time

To measure the time in the different interprocess communication facilities some functions and classes were coded. They measure the

- system time ("real" time)
  (class `clocktimer` defined in `common/clocktimer.h` (page 117))
- used CPU time together in user and kernel mode
  (class `cputimer` defined in `common/cputimer.h` (page 122))

Time measurements on computers normally have an impact upon the time they should measure. To ensure that the measurements do not influence the measured time too much, it is important to know how long the measurements take. The overhead for the two described classes `clocktimer` and `cputimer` is measured with the program `performance/timeroverhead.c` (page 273). This is the result of running the program on SparcCenter 2000/Solaris 2.5:

```
This program measures the overhead in using the timer classes "cputimer" and
"clocktimer"

all measurements are made 1000000 times...

Measuring the overhead of the 1000000-loop...
Overhead: clock secs:0  clock usecs:51093
Overhead: cpu secs:0  cpu usecs:50000

Measuring the overhead of clocktimer...
Overhead: clock secs:12  clock usecs:140469
Overhead: cpu secs:12  cpu usecs:140000
Average times:  clock secs:1.2e-05(=12 usecs)  cpu secs:1.2e-05(=12 usecs)

Measuring the overhead of cputimer...
Overhead: clock secs:23  clock usecs:993865
Overhead: cpu secs:21  cpu usecs:890000
Average times:  clock secs:2.3e-05(=23 usecs)  cpu secs:2.1e-05(=21 usecs)
```

It can be seen that the overhead is not very high. Therefore the overhead of measuring the time is ignored in all performance measurements. To reduce the influence caused by the measurement and other temporary differences all measurements are done several times (normally 1000 or more times).

The measurements were done in such a way that the influence of other interferences (other users using the computer systems at the same time etc.) was kept to a minimum.

The precision of the internal timers of a computer system has an influence on the accuracy of the measurements. The precision of the used CPU time is normally much lower than the precision of the "real time" clock. The precision of the used CPU time is 1/100 second on all used systems, whereby the clock precision is reported to be accurate to some μs.

To reduce the replication of code for measuring time the class `measurement`, defined in `common/measurement.h` (page 135), was developed. It uses the classes `cputimer` and `clocktimer` internally to measure time. The measured time is written to an internal memory buffer. This avoids time consuming I/O operations which could influence the measurements. The measured times can later be written out in various ways.

7.2 Principle Overhead of a System Call

UNIX system calls introduce a non-avoidable overhead. This overhead was measured with the program `performance/systemcalloverhead.c` (page 259). This program measures the time of a `getpid()` system call. This system call is regarded as one of the cheapest system calls in a UNIX operating system, as it has only to copy an integer value (the process number) from the kernel to the user process. The result is shown in Table 20.

System	**System Call Overhead**
SparcCenter 2000/Solaris 2.5	3.2 μs (312500 op/s)
HP 9000/710/HP-UX 9.01	1.6 μs (625000 op/s)
i586/Linux 1.2.9	2.5 μs (400000 op/s)

Table 20: Principle Overhead of a System Call

In all following measurement descriptions the number of system calls made to get the measured data is included. The non-avoidable overhead is *not* subtracted from the measured results, as programs/users experience these delays.

7.3 Signals

UNIX signals were used in some measurement programs to synchronize the measurement of different processes. The program `performance/signaloverhead.c` (page 255) measures how long it takes to send a signal from one process to another and to get a response signal. The described operation involves two system calls: `kill()` and `sigsuspend()`. This is the result of the program `performance/signaloverhead.c` (page 255) on SparcCenter 2000/Solaris 2.5.

```
This program measures how long it takes to send a signal
from one process to another and getting a reply signal back.

average over 100000 operations
number of operations per second: 1307.27
clock secs:0.000764955(=764.955 usecs)  cpu secs:0.0004031(=403.1 usecs)
```

Table 21 shows how often the described operation can be done on the different systems.

System	Signals and Acknowledgments
SparcCenter 2000/Solaris 2.5	770 μs (1300 op/s)
HP 9000/710/HP-UX 9.01	430 μs (2325 op/s)
i586/Linux 1.2.9	160 μs (6250 op/s)

Table 21: Performance of Signals on different UNIX Systems

Signals are used in some of the local interprocess communication examples to synchronize corresponding processes. Some functions defined in `common/synchronize.c` (page 157) hide the implementation details of how to synchronize related processes via signals.

7.4 Pipes

Program `performance/pipes.c` (page 247) measures the performance of pipes with different buffer sizes for `read()` and `write()` operations. Figure 32 shows the measured unsynchronized `read()` throughput, while Figure 33 shows the time for the (also unsynchronized) `read()` function calls.

The write counterpart is not shown as the `read()` and `write()` results are nearly the same. It can not be said how much `read()/write()` operations are involved in getting the data due to the different `read()` semantic for pipes, FIFOs, sockets etc. The problem is that I/O operations can return less than the requested amount of data,

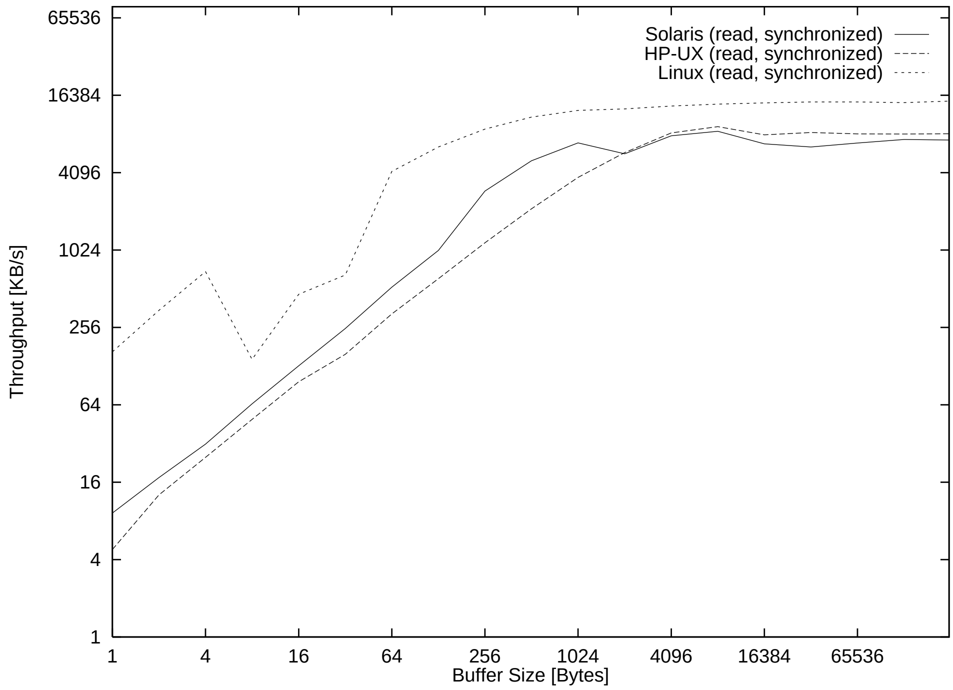

Figure 32: Throughput of Pipes

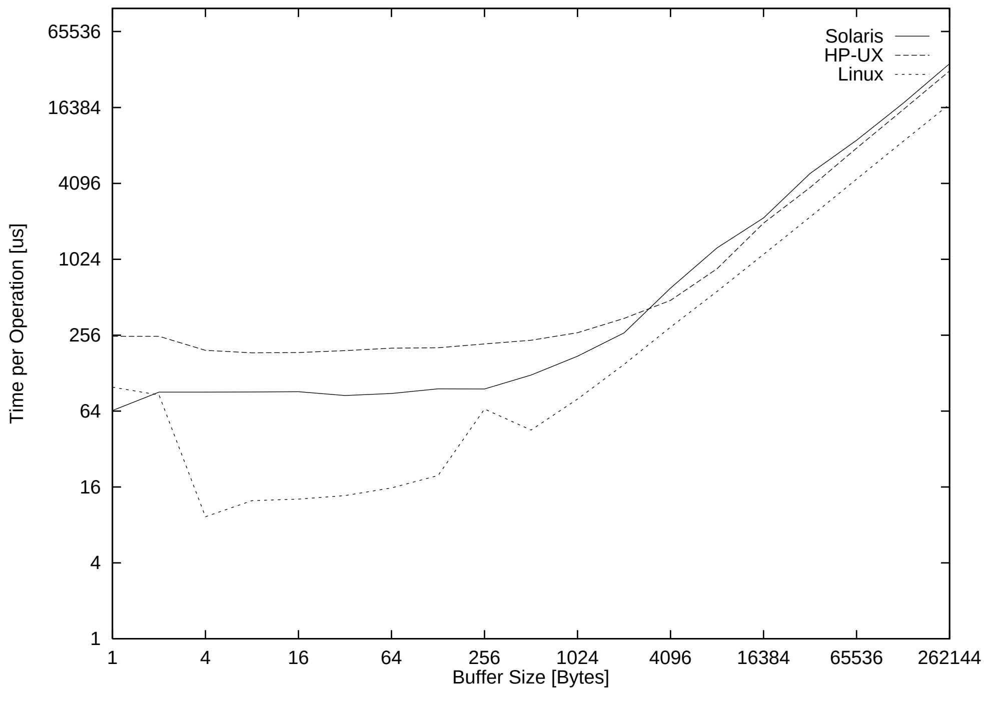

Figure 33: Time per `read()` System Call for Pipes

and therefore the system call might have to be issued again and again till all data is read/written (`common/readn.c` (page 152) and `common/writen.c` (page 164) handle these cases). Details about this problem are given in [Stev92, page 406].

7.5 FIFOs (named pipes)

The performance of named pipes is measured with the client/server pair of programs `performance/fifoclient.c` (page 236) and `performance/fifoserver.c` (page 238). Figures 34 and 35 show the results only for the `read()` system calls, as the corresponding `write()` system calls are nearly equal.

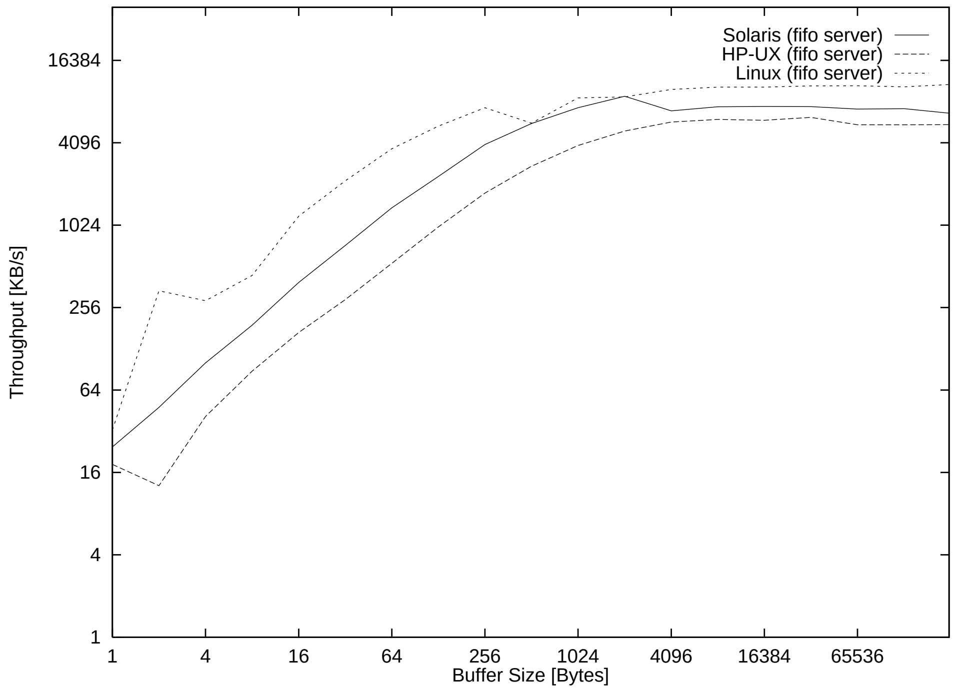

Figure 34: Throughput of FIFOs

At least one `read()/write()` system call is issued for each buffer size.

7.6 Message Queues

The program `performance/message.c` (page 244) was used to measure the performance of message queues. Figure 36 shows their throughput, and Figure 37 the time per `msgsnd()` system call.

Please note that the maximum size of messages is quite small, it was less than 4k on all three systems.

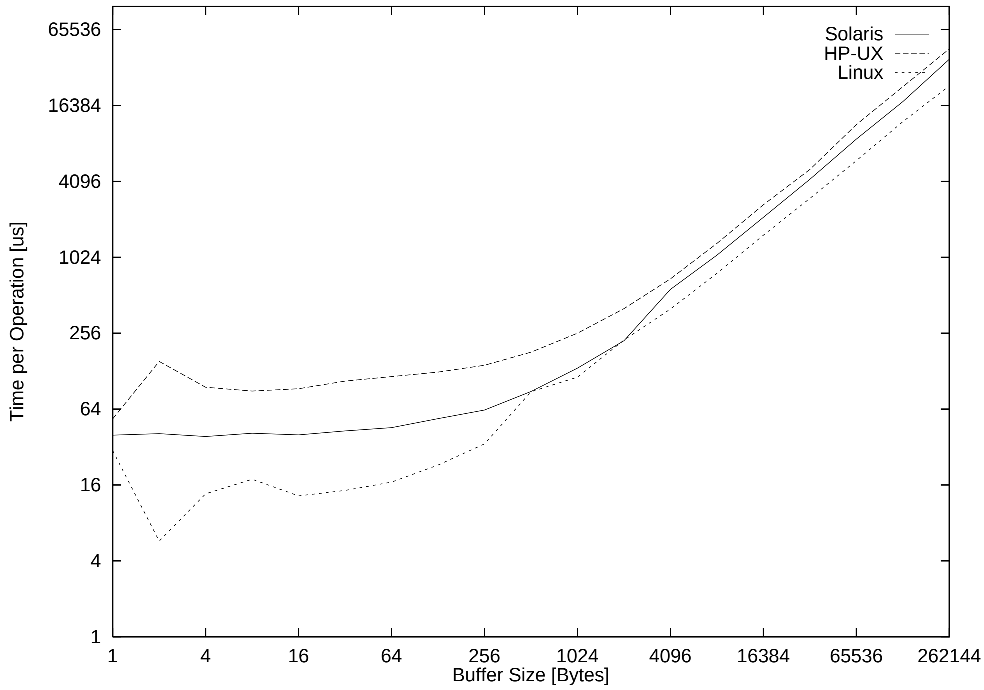

Figure 35: Time per `read()` System Call for FIFOs

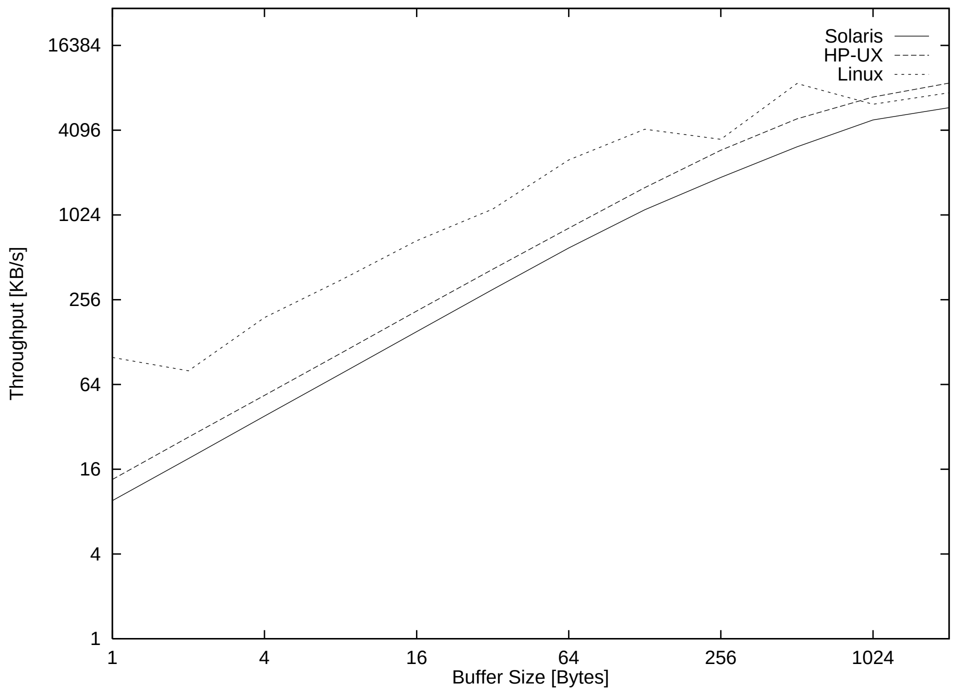

Figure 36: Throughput of Messages

7.7 Semaphores

Semaphores are not used to transfer information, they are used to synchronize access of different processes to other resources. How many semaphore operations can be done

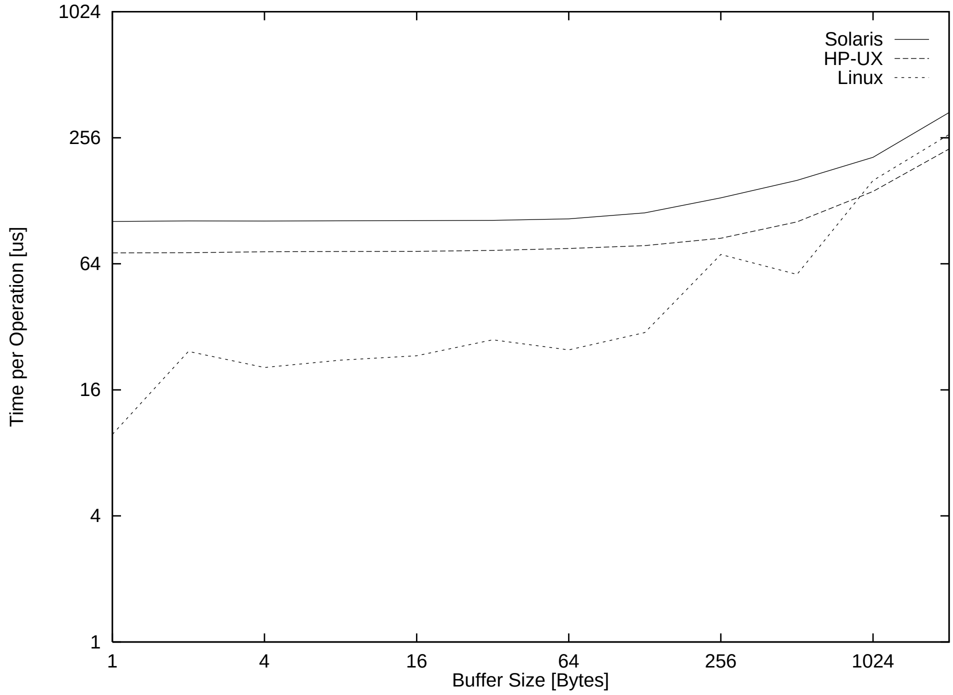

Figure 37: Time per `msgsnd()` System Call

System	Operations with undo	Operations w/o undo
SparcCenter 2000/Solaris 2.5	24.1 μs (41500 op/s)	21.5 μs (46500 op/s)
HP 9000/710/HP-UX 9.01	41.0 μs (24400 os/s)	28.6 μs (35000 op/s)
i586/Linux 1.2.9	7.8 μs (128000 op/s)	7.7 μs (130000 op/s)

Table 22: Semaphore Operations

per second? The rounded result for one `semop()` system call derived with the program `performance/semaphore.c` (page 249) for the different systems is shown in Table 22.

Please refer to Chapter 3.5.2 for the meaning of the undo facility of semaphores.

7.8 Shared Memory

The program `performance/memory.c` (page 240) is used to measure if there is any time penalty in writing to shared memory instead of normal memory. A buffer of 1 Mb in "normal" memory and an equally sized shared memory region is allocated. The program writes zeros in both buffers. A read test could not be performed, as the compilers optimized the read code to "nothing". The best results are given in Table 23.

System	Normal Memory	Shared Memory
SparcCenter 2000/Solaris 2.5	9649 μs (103.6 Mb/s)	9663 μs (103.5 Mb/s)
HP 9000/710/HP-UX 9.01	34181 μs (29.3 Mb/s)	34183 μs (29.3 Mb/s)
i586/Linux 1.2.9	35327 μs (28.3 Mb/s)	35543 μs (28.1 Mb/s)

Table 23: Time needed to write 1 Mb

The results suggest that there is no difference in accessing "normal" and shared memory. No system call was used to write into the memory.

One might ask how big the overhead is to establish memory sharing. Figure 38 shows the time needed to create, attach, detach and remove a shared memory region of different sizes. Four system calls were used per operation: `shmget()`, `shmat()`, `shmdt()`, and `shmctl()`.

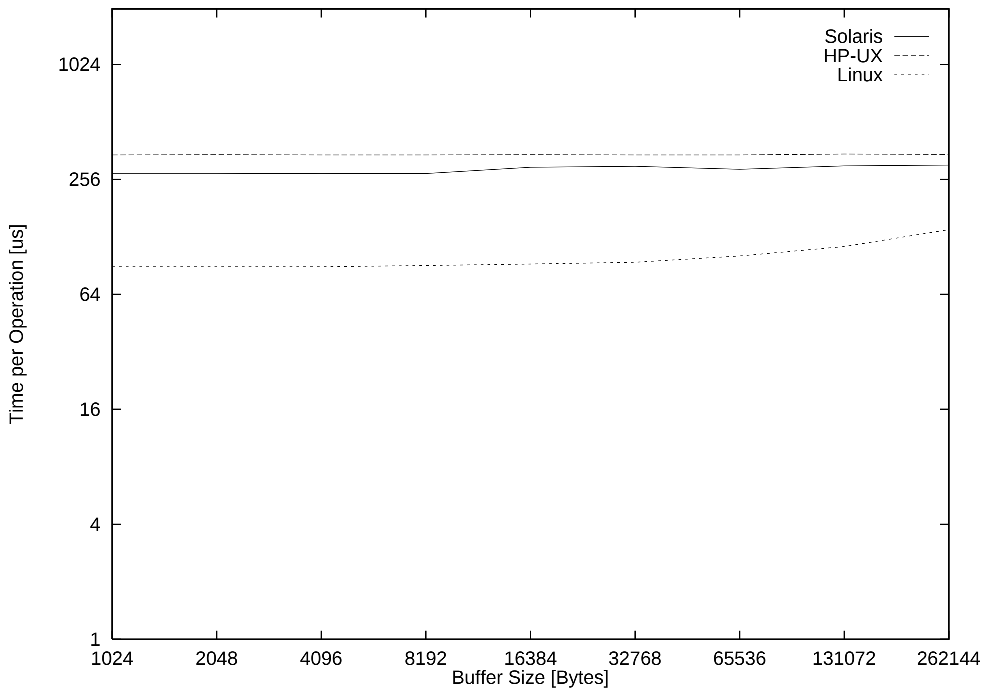

Figure 38: Time needed to create, attach, detach and remove a Shared Memory Region

7.9 Sockets

7.9.1 UNIX Domain

As already described in Chapters 3.4 and 6.1.5 the `socketpair()` system call can be used to create a bi-directional pipe. The program `performance/socketlocal.c` (page 258) is nearly identical to the corresponding program `performance/pipes.c` (page 247) using the `pipe()` system call. Figure 39 shows the throughput of local sockets, Figure 40 the time for the `read()` system calls.

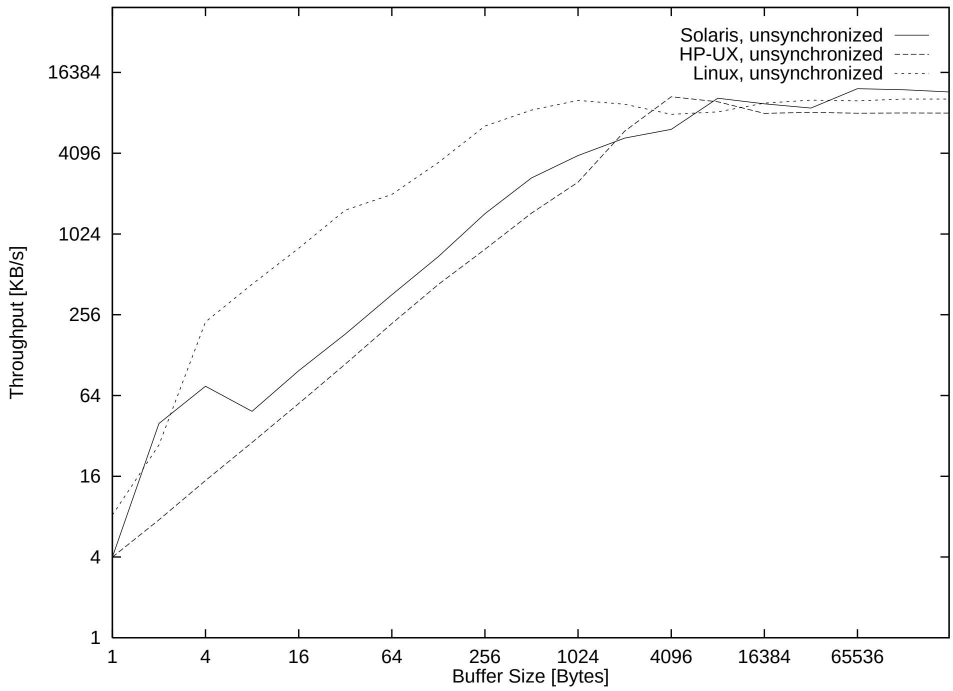

Figure 39: Throughput of local Sockets created by the `socketpair()` System Call

The performance (comparing Figure 39 with Figure 32 on page 75) is roughly that of the `pipe()` version, and in fact most `pipe()` implementations today seem to be implemented by using the `socketpair()` system call.

7.9.2 Internet Domain: UDP

The client program `performance/udpsclient.c` (page 278) measures the time that UDP messages need from one process to another. The sender waits for a short acknowledgment message from the server `performance/udpsserver.c` (page 282). Figure 41 shows the throughput if server and client are running on the same system, and Figure 42 the throughput between `aidan` and `pike17` and also the throughput between

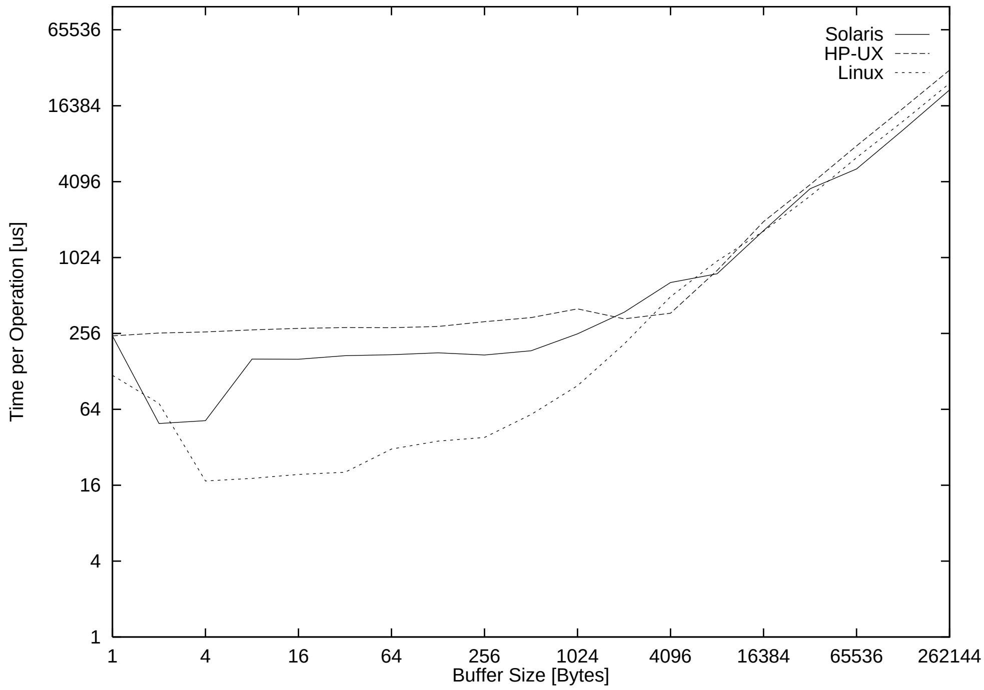

Figure 40: Time for `read()` System Calls for local Sockets

two HP 9000/710 computers on the same physical Ethernet. The time for the corresponding system calls can be seen in Figures 43 and 44. Per operation the following system calls are executed both in the client and in the server process: `recvfrom()` and `sendto()`. Please note that the default maximum size for UDP was different on the used systems, it was 16 Kb on SparcCenter 2000/Solaris 2.5 and i586/Linux 1.2.9, and less than 16 Kb on HP 9000/710/HP-UX 9.01.

The computers `aidan` and `pike17` are not connected to the same physical Ethernet, which explains why the throughput is slower between them compared to the two `pike`s.

UDP is not reliable. This had an impact on the measurement between different computers. The number of repeats for each buffer size had to be reduced, as none of the tests otherwise succeeded (the measurement program relies on data to be delivered, as in the case of a not delivered message both client and server are waiting for each other forever).

7.9.3 Internet Domain: TCP

The client program `performance/tcpsclientoverhead.c` (page 261) together with the server program `performance/tcpsserverconcurrentoverhead.c` (page 266) is used to measure the overhead of establishing a TCP connection. The connection estab-

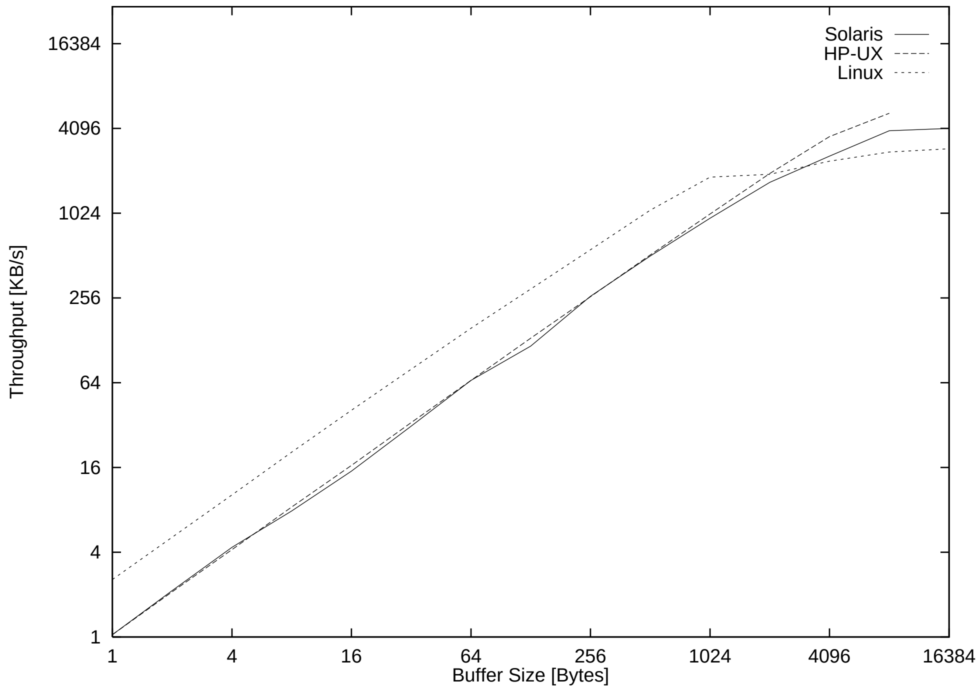

Figure 41: Throughput of UDP on one computer

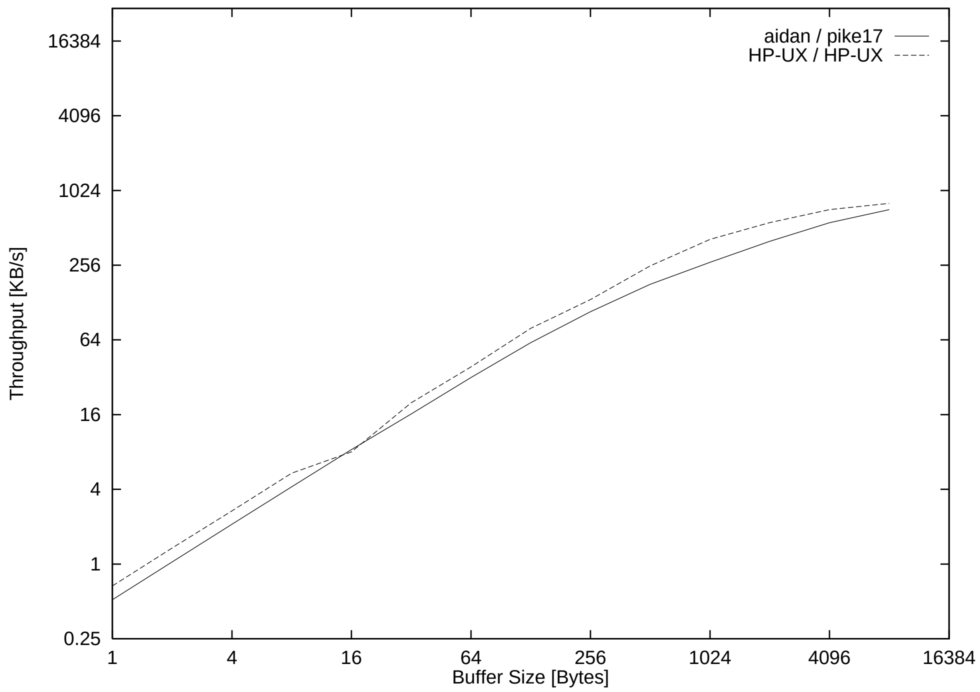

Figure 42: UDP Throughput between different Computers

lishment measurements include the execution of the system calls `socket()`, `connect()`, and `close()` on the client side and `accept()`, `fork()` (the TCP server is implemented as a concurrent server which can handle multiple requests simultaneously), and two

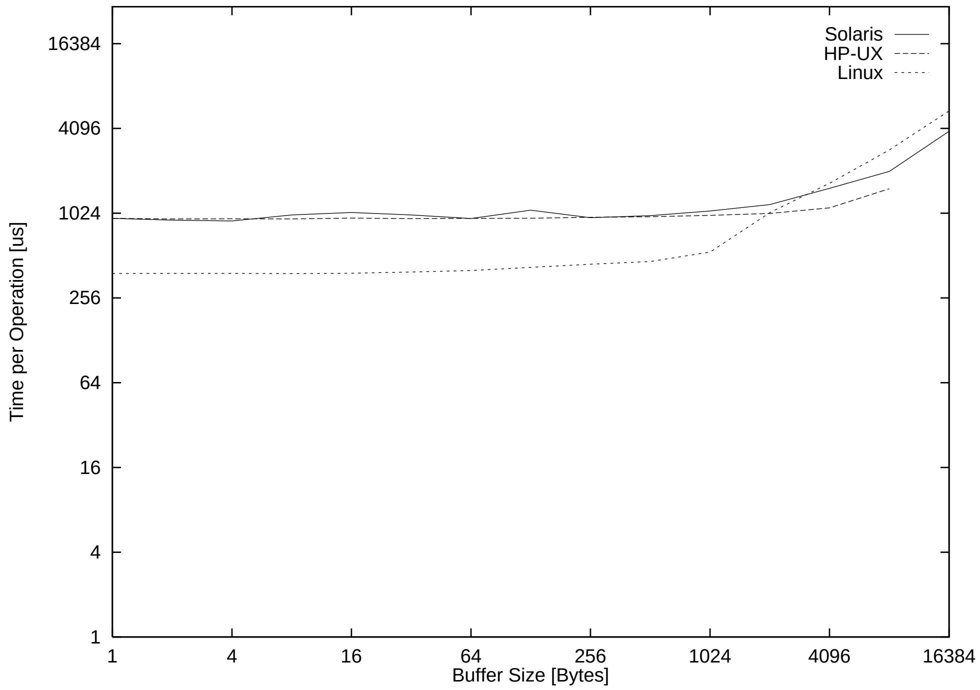

Figure 43: Time per `sendto()` and `recvfrom()` System Calls (both executed in client and server process) on one Computer

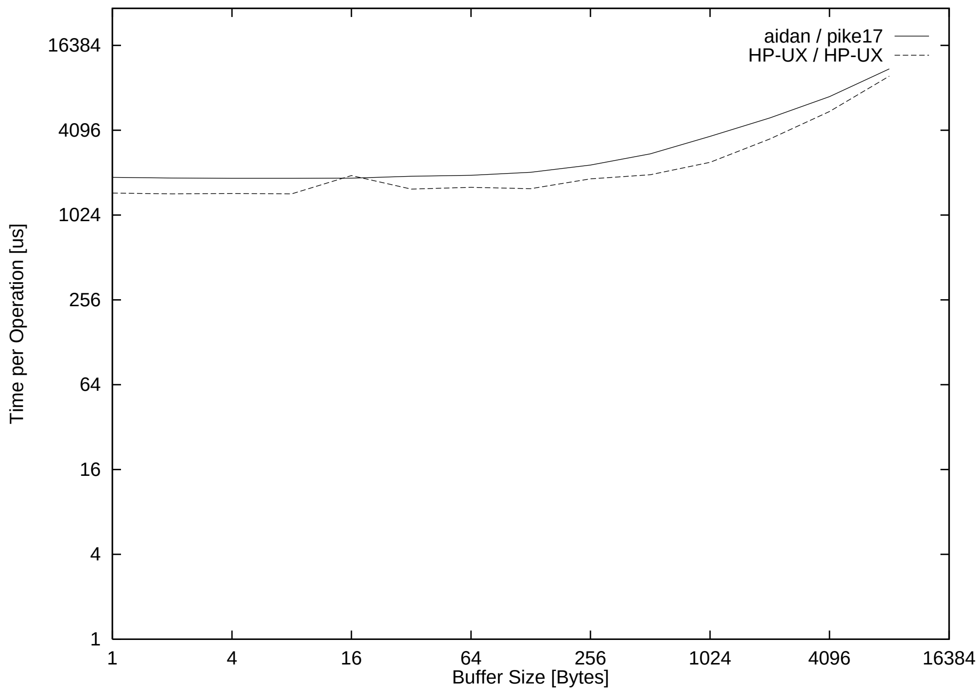

Figure 44: Time per `sendto()` and `recvfrom()` System Calls (both executed in client and server process) between different Computers

System	Overhead
SparcCenter 2000/Solaris 2.5	5500 μs
HP 9000/710/HP-UX 9.01	1900 μs
i586/Linux 1.2.9	1500 μs

Table 24: Average Overhead of Establishing a TCP Connection on one Computer

close()s on the server side. The rounded results are given in Table 24.

For comparison: on average it takes about 6600 μs to establish a connection from aidan to pike17, and 2400 μs between two pikes on the same physical Ethernet.

Figure 45 states the throughput of a local TCP connection. The executed system calls are (eventually several) write()s on the client side and (eventually several) read()s on the server side.

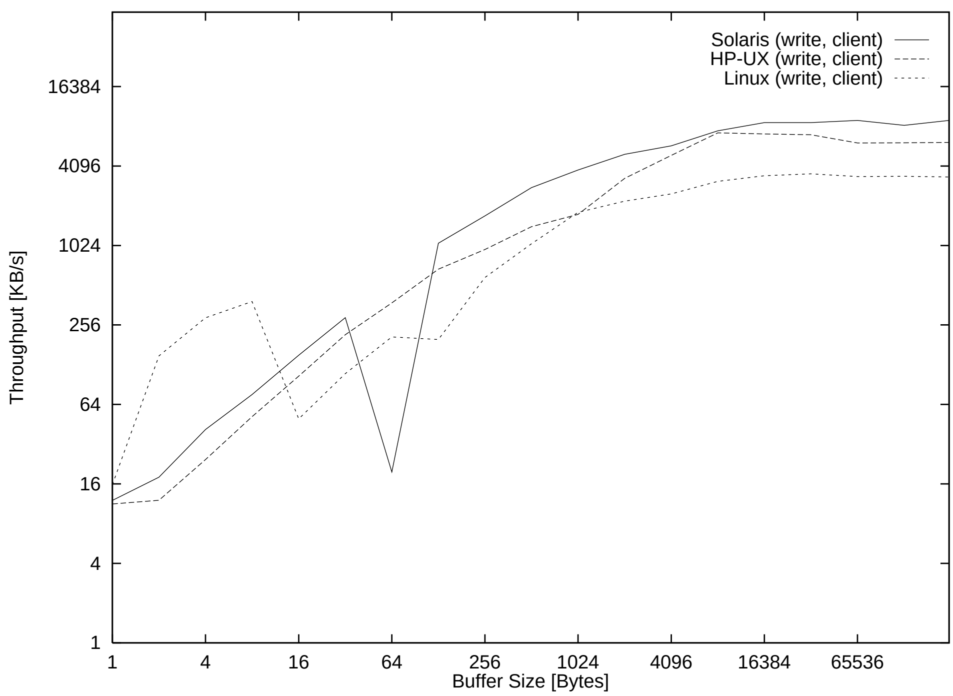

Figure 45: Throughput of a local TCP Connection

The client program performance/tcpsclientthroughput.c (page 264) together with the server program performance/tcpsserverconcurrent.c (page 270), which measured the data for Figure 45, were also used to measure the TCP throughput between two HP-9000/710 computers on the same physical network and the TCP throughput between `aidan` and `pike17`. The result is shown in Figure 46. The used time for the corresponding `read()` system calls is shown in Figures 47 and 48.

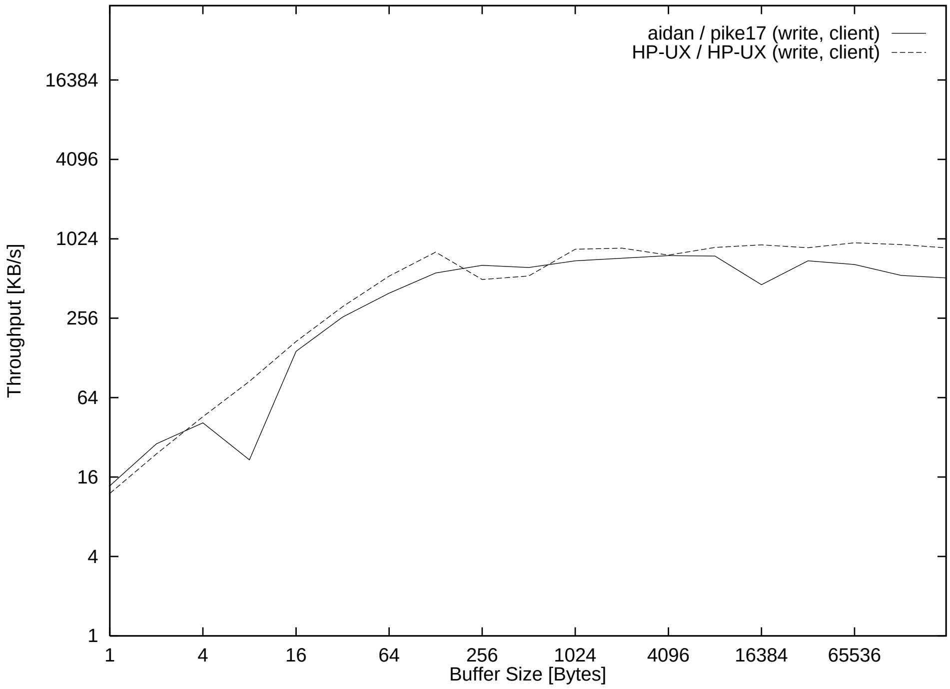

Figure 46: Throughput of a TCP Connection between different Computers

7.10 Transport Layer Interface

The UDP and TCP measurements were also performed using the TLI interface on `aidan` only, as the other systems unfortunately do not support the Transport Layer Interface.

The results were similar to the ones from the socket interface (about 20% slower for TCP throughput and about 15% slower for UDP throughput), therefore the results are not shown here.

7.11 Conclusion

The IPC facilities which were investigated have major differences in performance. The three different systems produce dramatically different performances, but in principle their behaviour for a specific IPC facility is very similar.

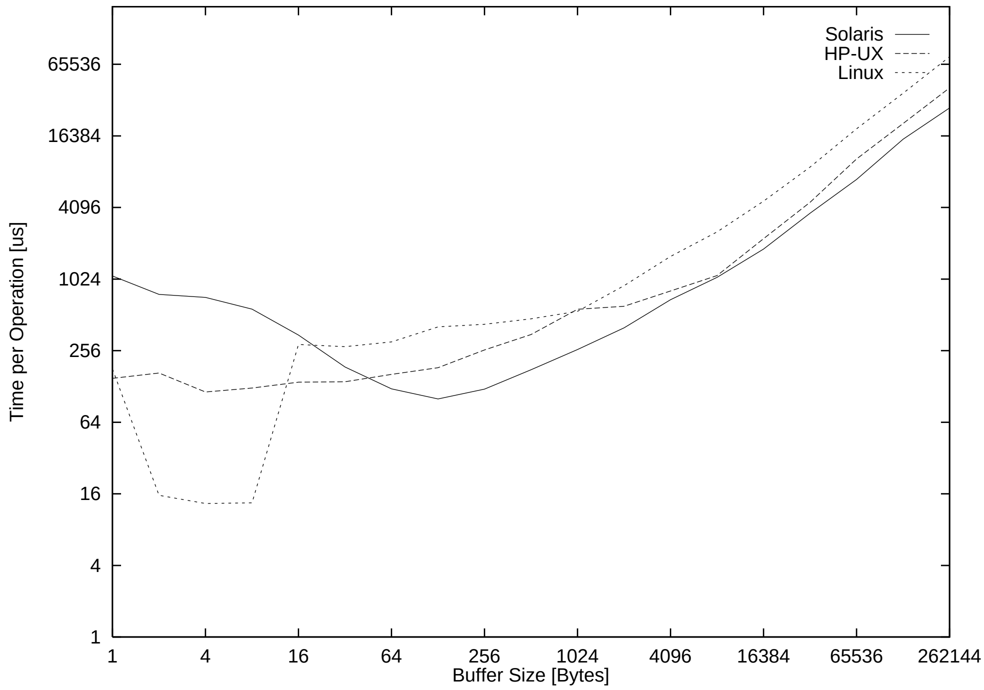

Figure 47: Time for `read()` System Calls of a TCP Connection on one Computer

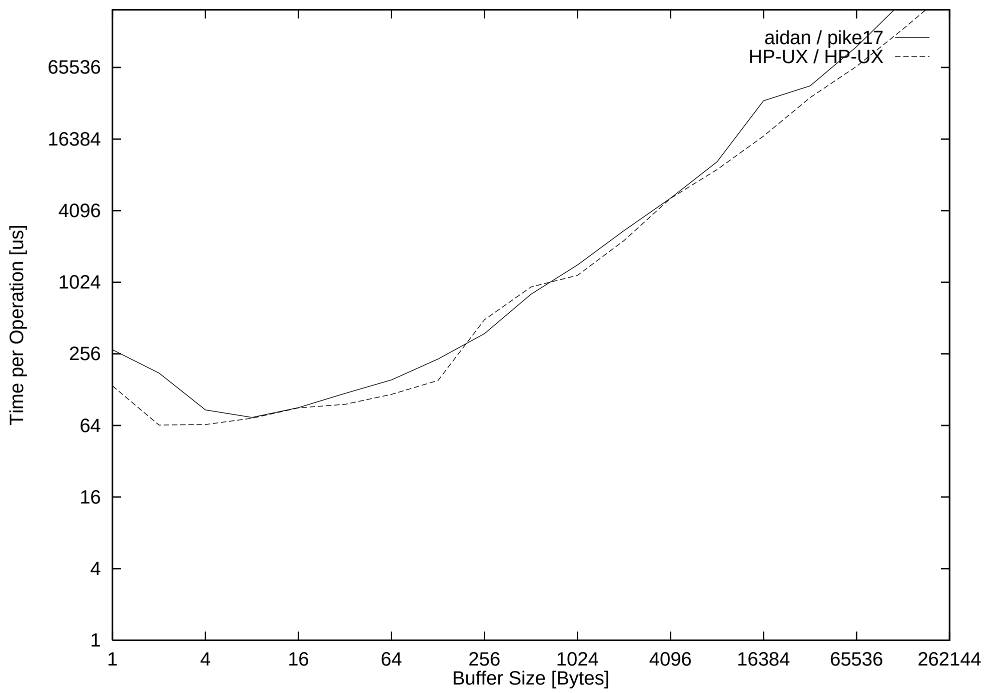

Figure 48: Time for `read()` System Calls of a TCP Connection between different Computers

To synchronize processes on one computer systems semaphores should be used instead of signals, as the overhead of signals is dramatic, caused by the context switch that has to be done.

The size of the buffer has a great impact on the throughput of a local IPC facility. Up to about 4 - 8 Kb (dependent of the system and the IPC facility) larger buffers normally produce a larger throughput. This is likely to be caused by the overheads of the system calls. Larger buffer sizes do not cause major performance loss or gain.

For interprocess communication on one computer, shared memory is the fastest way to transfer huge amounts of information from one process to another. There is no penalty in using shared memory instead of normal memory. The size of the shared memory region has nearly no impact on the time that is needed to create, attach, detach, and remove a shared memory region.

Message queues, originally designed to provide a faster form of IPC are now slower than other forms of IPC (possibly caused by the small buffer sizes). They therefore should not be used for high speed information exchange any more. The possibility of selecting messages by priority may justify their use for some purposes.

For distributed IPC facilities the time to transfer the data has a stronger influence than on the local ones. Therefore the time of an operation is more proportional to the amount of data transfered than in the local IPC facilities.

UDP is faster than TCP at transferring data between different computers, but the application has to take care of lost packages, congestion, flow control etc. On `aidan` and on the `pike`s the local use of TCP has a *much* higher throughput than UDP.

The overheads of establishing a TCP connection are quite high. If only small amounts of data have to be transfered, UDP should be used. In the time needed for a connection establishment, UDP might already have transfered all the data.

The speed of TCP used locally on a system is not dramatically slower than equivalent local IPC facilities (e.g. pipes), at least on `aidan` and the `pike`s. Therefore where speed is not very critical, applications that might be distributed, should use TCP wherever possible instead of pipes and local sockets.

Type	Small amounts of Data	Big amount of Data
local	semaphores (very small amount of information) pipes, FIFOs, sockets etc.	shared memory
distributed	UDP (problem: not reliable)	TCP (reliable, handles congestion etc.)

Table 25: IPC Facility to use for different Amount of Data

Connection Type	Local	Distributed
Connection-oriented, reliable	pipes, FIFOs, stream sockets, (shared memory)	TCP
Connection-oriented, unreliable	–	–
Connectionless, reliable	message queues, semaphores, signals	–
Connectionless, unreliable	datagram sockets	UDP

Table 26: IPC Facilities for different Types of Connections

Most IPC facilities have a fixed time overhead per operation. Up to a specific buffer size the time per operation is nearly independent of the buffer size used.

Tables 25 and 26 present the most appropriate facility which should be used for particular purposes. "UDP" and "TCP" should be read as using sockets or the TLI with the specified protocol.

7.12 Summary

Various measurements were done on three different systems to get information about the performance of different interprocess communication facilities.

The performance difference between the three systems is sometimes high, but the characteristics are very similar. Shared memory is the fastest mechanism to transfer large amounts of data between processes on one computer, while UDP is the IPC mechanism with the highest throughput between different computers. The size of buffers should not be less than 4 Kb if larger amounts of data are exchanged.

The performance measurements demonstrated the unreliability of the UDP protocol. Therefore the next chapter examines what is necessary to ensure reliable datagram exchange.

8 Making Unreliable Communication Reliable

This chapter

- **describes how to implement reliable send()/receive() functions based on unreliable UDP datagrams**
- **shows how these functions have to be designed in terms of**
  1. **lost messages**
  2. **corrupt messages**
  3. **duplicate messages**
  4. **messages out of order**

The UDP performance measurements in the previous chapter have demonstrated that UDP is not reliable even on a local area network. How can communication then made reliable? This chapter gives an overview about what is necessary in principle to archive reliable delivered messages if an unreliable message protocol like UDP is used. This knowledge is then used to implement "safe" functions. For reliable messages ideally the following conditions have to be met:

1. messages are delivered in the order they are sent
2. every message sent is delivered exactly once
3. an acknowledgment is returned for each delivered message

The assumed failure model is:

- communication failures: functioning processes are only temporarily unable to communicate (due to networking problems like lost messages or corruption of messages)
- node crashes: A node either works or crashes (stops working); a crashed node eventually recovers

The second and third goals above are hard to achieve in the presence of crashes and unreliable networks. In particular, it is not possible to achieve them without making changes to stable storage that survives crashes. Details about this can be found in [Mull93, page 251f] or other introduction books on distributed systems.

For the following it is assumed that node crashes do **not** happen, as otherwise the system gets really complicated. An example for a system that covers all goals is the

Arjuna system, developed by the University of Newcastle upon Tyne.

The following problems can arise while two processes are communicating through an unreliable network: Messages from one computer to another are:

1. lost

2. corrupted

3. delivered more than once

4. delivered out of order

It is assumed that normally the receiver of a data message sends a reply message that it got a message. So a normal "save send" looks like this:

1. sender: sends a data message and waits for the acknowledgment message

2. receiver: receives a data message and sends acknowledgment to sender

8.1 Lost Messages

If the underlying message transport system is unreliable, messages can get lost. Without some caution, this could lead to the sender of a message waiting forever for two reasons:

- the data message from the sender to the destination (receiver) is lost

- the acknowledgment message from the receiver to the sender is lost

The response to lost messages is that the sender of a message sets a timer (after sending the message); so the sender never blocks for ever. If a timeout occurs, the sender does not know why it did not get a reply. The sender can either

1. give up

2. try again (send the message again)

If the sender gives up, it has no guarantee that the receiver hasn't got the message (and processed it). If the sender tries again, it just re-sends the message to the destination.

8.2 Corrupt Messages

Every message should contain a checksum, e.g. the popular CRC (Cyclic Redundancy Checksum), which detects with a very high probability whether a message was altered during transmission. Most protocol suites and applications just discard all corrupt messages. It is assumed that the other side timeouts and retransmits the message. For the sender and receiver a corrupt message is handled like a lost message.

8.3 Duplicate Messages

Either through the underlying network, or timeouts from the sender (through lost acknowledgment messages) the receiver of a message can get a message more than once. The normal approach to detect this is to introduce **sequence numbers**. If the communication is synchronous, one sequence number suffices, otherwise two different sequence numbers are useful. Every new message gets a new message number. The sender and receiver have to agree on the initial sequence numbers at the start of communication. Both client and server increment their sequence number if they send data, and wait for an acknowledgment of their sequence number. If a message is delivered more than once, this can easily be recognized, as the sequence number was already seen by the receiver, so it can discard the message. The problem with introducing sequence numbers is that this state information has to be maintained. This is not always desirable. In particular if a connectionless protocol is used, and the exchange between client and server just consists of one request and one reply, it is expensive to maintain the result of all operations. It is normally only possible to maintain a fixed number of results.

A detailed study how to select sequence numbers can be found in [Tom75].

8.4 Messages out of Order

The sequence number can also be used to detect messages that arrive out of order. If every message is acknowledged before a new one is sent, this can not happen.

8.5 Congestion Control

In connection-oriented communication or multiple messages to one destination together with message switching congestion control is a serious issue. Some aspects are investigated and solutions suggested in [Jac88].

8.6 Fundamental Problem of uncertain Communication

Even if all of the problems above can be solved, a certain amount of uncertainty is unavoidable. One side can never be sure that the last acknowledgment was delivered correctly. This is caused by the fact that the receiver of an acknowledgment would have to acknowledge the acknowledgment, which in turn the original sender would have to acknowledge, and so on. This problem is known as the “two army problem” and described in detail in [Tan88, page 397f].

8.7 Implementation

The function `safesendto()` and `saferecvfrom()` in the file `distSOCKET/safesend.c` (page 204) take all of the mentioned points into account. They can be used as plug-in replacement for the system calls `sendto()` and `recvfrom()`, which are used for connectionless protocols. The function `safesendto()` sends a message and waits for a reply. If no reply is returned, the message is sent again after a certain timeout period. The function `saferecvfrom()` acknowledges all incoming messages. Both keep track of a fixed number of sends/receives.

8.8 Summary

The problem of reliable communication using an unreliable communication protocol is discussed. A reliable send()/receive() function using sockets is developed. It was found out that the overhead of getting reliable communication using an unreliable service is considerable, especially in terms of the increased number of exchanged messages (at least two messages instead of one), increased delay till an acknowledgment message is received, and efficient implementation. Therefore the use of connection-oriented protocols like TCP should be considered seriously if the data exchange between two processes is substantial.

9 Conclusion

This chapter

- **summarizes the results derived in this document**
- **briefly discusses which additional facilities exist**
- **describes possible improvements**
- **gives an overview of the future of UNIX communication facilities**

9.1 Results

The UNIX operating system provides several interprocess communication facilities. Some IPC facilities are restricted to processes which are executed on the same computer system. Most described IPC facilities are designed to transport data (pipes, FIFOs, message queues, and shared memory). Others synchronize processes (signals and semaphores). Except for shared memory, the use of all local interprocess communication facilities which transport data is very similar. Two interfaces to networking allow processes to cross system boundaries and make it possible for them to communicate via networks. Networking adds a lot of complexity to interprocess communication.

The performance measurements were the part of this document with the most work for the author. The performance of all more important IPC facilities was measured and compared for three different systems. The results allow recommendations which IPC facilities to use for different purposes.

The rest of this chapter gives an overview of the IPC facilities not covered, describes shortcomings of this document, and gives a look into the UNIX communication future.

9.2 Additional IPC Facilities

Due to space and time not all available UNIX IPC facilities could be investigated in detail. The most important facilities are covered briefly.

9.2.1 Memory Mapped I/O

Strictly speaking memory mapped I/O is not an interprocess communication facility. It is described for completeness only. With memory mapped I/O a file on a disk is mapped into a buffer in memory. If bytes are read in the buffer, the corresponding

System Call	Description
`mmap()`	map pages of memory
`munmap()`	unmap pages of memory

Table 27: System Calls for Memory Mapped I/O

System Call	Description
`ioctl()`	performs various operations on STREAMS
`getmsg()`	read messages from a STREAM head
`getpmsg()`	same as `getmsg()`, but with additional priority
`putmsg()`	write messages to a STREAM head
`putpmsg()`	same as `putmsg()`, but with additional priority
`poll()`	returns number of ready file descriptors

Table 28: System Calls for STREAMs

bytes are read from the file, and if bytes are modified, the corresponding bytes are automatically written to the file. Table 27 shows system calls to use memory mapped I/O.

9.2.2 STREAMS

STREAMS were introduced into UNIX by System V Release 3. They are a general way to interface communication drivers in the kernel. A STREAM provides a full-duplex path between a user process and a device driver, which can be either talk to hardware or be a "pseudo device driver". A STREAM consists of a *stream head*, eventually some processing modules, and a device driver. Processing modules can be *pushed* onto a STREAM. The exchange of data is performed using messages. Figure 49 shows a STREAM with one processing module. Table 28 describes system calls which can be used with STREAMS in addition to the normal `read()`, `write()`, `open()` and `close()` system calls.

STREAMS provide a more general and flexible way than normal UNIX I/O to combine and use device drivers by user processes. STREAMS are used as an example to implement the Transport Layer Interface. Additional information about STREAMS can be found in [Bach86, Chapter 10.4] and [Stev92, Chapter 12.4].

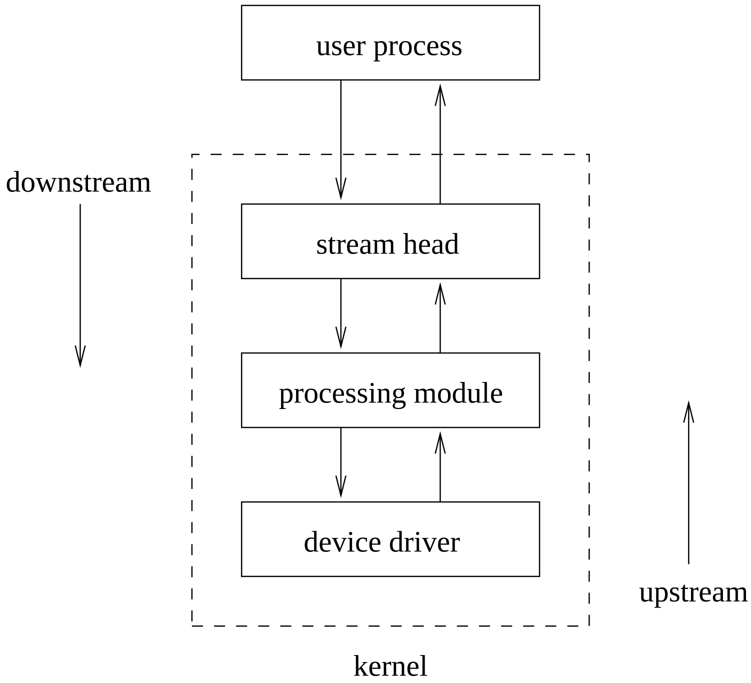

Figure 49: A STREAM with one processing Module

9.2.3 OSF DCE

OSF's Distributed Computing Environment is becoming more and more popular. DCE support can be added to a large number of operating systems, including UNIX. The target for DCE are business applications. DCE provides services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment [OSF DCE]. The benefits of DCE can be categorized into its support of distributed applications; the integration of its components with each other; DCE's relationship to its platforms; its support for data sharing, and DCE's interaction with the world outside of DCE [OSF DCE]. From the programming point of view the supports of threads and RPC is interesting. The services include a distributed file service, time service, security service and directory service.

A good overview of DCE is given in [OSF DCE].

9.3 Possible Improvements

Although this document covers a wide range of UNIX communication facilities, the following improvements are possible:

- describing STREAMS in more detail

- covering of security issues
- covering of synchronization of communication between different computers
- describing higher level facilities like RPC in more detail
- covering of TELNET, HTTP and other protocols
- investigating how all described IPC facilities are implemented
- investigating the errors in measuring the performance (variance, etc.) in more detail
- covering of pseudo-terminals
- better wording

9.4 The Future of UNIX Communication Facilities

With the development of so-called *microkernels* (kernels which try to be as small as possible) like Mach, Spring, Grashopper and others the demand for fast and effective local IPC facilities is increasing. Some results of IPC performance measurements on microkernels are stated in [Lied95].

More and more computers get connected via networks. Therefore the need for fast and effective distributed IPC facilities is growing rapidly. The development and standardization of a reliable connectionless protocol or a “leaner” TCP protocol would be desirable. T/TCP (TCP with extensions for transactions, described in [Stev94] and [RFC 1379]) is a step in the right direction, especially with the increasing number of small data transfers in the Internet due the increased popularity of the World Wide Web.

9.4.1 IPng/IPv6

The Internet Protocol IPv4 has one serious problem: the address space is too small. Therefore a new Version 6 of the internet protocol is currently being developed, called IPng for Internet Protocol next generation. Major changes from IPv4 to IPng are (after [Hin95] and [RFC 1883], see also Figure 50):

- *Expanded Routing and Addressing Capabilities*: IPng increases the IP address size from 32 bits to 128 bits

- *Header Format Simplification*: Some IPv4 header fields have been dropped or made optional, to reduce the common-case processing cost of packet handling and to keep the bandwidth cost of the IPng header as low as possible despite the increased size of the addresses.
- *Improved Support for Options*: Changes in the way IP header options are encoded allows for more efficient forwarding, less stringent limits on the length of options, and greater flexibility for introducing new options in the future.
- *Quality-Of-Service Capabilities*: A new capability is added to enable the labelling of packets belonging to particular traffic "flows" for which the sender requests special handling, such as non-default quality of service or "real-time" service.
- *Authentication and Privacy Capabilities*: IPng includes the definition of extensions which provide support for authentication, data integrity, and (optional) confidentiality. This is included as a basic element of IPng and will be included in all implementations.

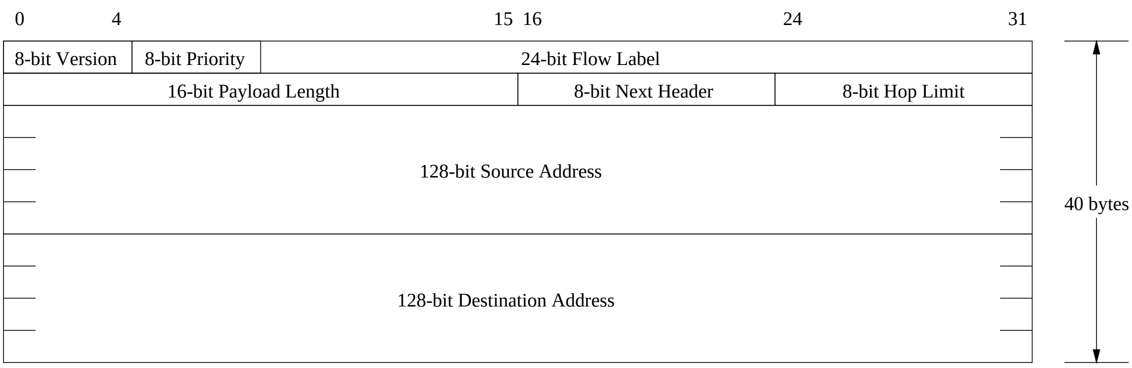

Figure 50: The basic IPng Header

The conversion from IPv4 to IPv6 is designed to be very smooth. It provides a platform for new internet functionality that will be required in the near future.

IPng is specified in [RFC 1883]. A more detailed overview of IPng can be found in [Hin95]. The WWW home page for IPng development is
`http://www.ietf.cnri.reston.va.us/ipng/ipng.html`

9.4.2 C++ Classes for IPC

The use of C++ classes can reduce the complexity of interprocess communication from the programmers point of view and allow a consistent interface for different IPC facilities. They also can make it easier to port applications between different platforms if appropriate implementations exist.

Two freely available collection of classes are:

- ADAPTIVE Communication Environment (ACE):
  `http://www.cs.wustl.edu/ schmidt/ACE.html`

- Socket++:
  `ftp://ftp.th-darmstadt.de/pub/programming/languages/C++/class-libraries/networking/socket++-1.10.tar.gz`

Socket++ provides 32 classes which make the use of sockets and pipes easier. The C++ iostream model is used for communication. Socket++ supports the UNIX platform. ACE however, is a huge package that is very powerful. It provides more than 50 C++ classes for IPC which support sockets, TLI, pipes, Sun RPC and CORBA. ACE implementations exist for most UNIX versions, Windows 3.1 and Windows NT.

9.5 A Final Word

This document has given an overview of the more important UNIX communication facilities, especially interprocess communication. All facilities described are illustrated with example programs. The theory of networking was explained. One example for a protocol suite, TCP/IP, the most important one for UNIX interprocessor communication, was described. The performance of all discussed interprocess communication facilities was measured and compared on three different computer systems. The overhead of making communication safe in spite of unreliable services was discussed.

It is therefore believed by the author that the original aim of this individual project "investigate the different mechanisms for inter-process and inter-processor communication offered by the UNIX operating system" is achieved.

It is hoped by the author that the reader has enjoyed reading this document. For the author the creation of this document was a substantial piece of interesting work.

10 References

[Bach86] M.J. Bach, *The Design of the Unix Operating System*, Prentice-Hall, 1986, ISBN: 0-13-201757-1 025

Explains a lot of SVR2 UNIX internals.

[Bost93] Keith Bostic, *4.4BSD Release*, ;login:, Volume 18, No. 5, September/October 1993, pages 29-32

Information about 4.4BSD release and why BSD UNIX is not developed further.

[Clark88] David D. Clark, *The Design Philosophy of the DARPA Internet Protocols*, Computer Communications Review, Volume 18, Number 4, August 1988, pages 106-114, ACM Press

[CCR95/2] Computer Communication Review, *Special Issue on ATM*, Volume 25, Number 2, April 1995, ACM Press

[Col94] George Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems: Concepts and Design* (Second Edition), Addison-Wesley, 1994, ISBN: 0-201-62433-8

[Com95] Douglas E. Comer, *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture* (Third Edition), Prentice-Hall, 1995, ISBN: 0-13-216987-8

A very good, detailed, and up to date description of TCP/IP, includes IPv6.

[Com94] Douglas E. Comer, David L. Stevens, *Internetworking with TCP/IP, Vol. II: Design, Implementation, and Internals* (Second Edition), Prentice-Hall, 1994, ISBN: 0-13-134677-6

[Day95] John Day, *The (Un)Revised OSI Reference Model*, Computer Communications Review, Volume 25, Number 5, October 1995, pages 39-55, ACM Press

[Elb94] Bruce R. Elbert, Bobby Martyna, *Client/Server Computing*, Artech House, 1994, ISBN: 0-89006-691-4

[GNU C] Sandra Loosemore *et al.*, *The GNU C Library Reference Manual*, Edition 0.05, 1993, Free Software Foundation

[Goldt94] Sven Goldt *et al.*, *The Linux Programmer's Guide*, Version 0.4, March 1994, file `LPG/lpg.dvi` in the archive `ftp://sunsite.unc.edu:/pub/-Linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz`

[Hin95] Robert M. Hinden, *IP Next Generation Overview*, ConneXions, Volume 9, No. 3, 1995, pages 2-18, ISSN: 0894-5926

[HP92] *Berkeley IPC Programmer's Guide*, Hewlett-Packard Company 07/92

[Jac88] Van Jacobson, *Congestion Avoidance and Control*, Computer Communications Review, Volume 18, Number 4, August 1988, pages 314-329, ACM Press
A famous article about how to avoid congestion with TCP/IP, with optimized algorithm.

[Kirch94] Olaf Kirch, *The Linux Administrators' Guide*, 1994, file `nag.ps` in the archive file `ftp://sunsite.unc.edu/pub/Linux/docs/LDP/-network-guide/nag-1.0.ps.tar.gz`

[Leff88] S. Leffler, M. McKusick, J.S. Quarterman, and M. Karels, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1988, ISBN: 0-201-06196-1
Describes all the internals of the 4.3 BSD Tahoe release.

[Leff93] S. Leffler, R. Fabry, W. Joy, P. Lapsley, S. Miller, C. Torek, *An Advanced 4.4BSD Interprocess Communication Tutorial,* UNIX Programmer's Supplementary Documents, 4.4 Berkeley Software Distribution, 1993, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
Provides an introduction to the interprocess communication facilities included in the 4.4BSD release of the UNIX system from the original authors.

[Lied95] Jochen Liedtke, *On μ-Kernel Construction*, Operating Systems Review, Volume 29, Number 5, December 1995, pages 237-250, ACM Press

[Mal92] Carl Malamud, *STACKS: Interoperability in Today's Computer Networks*, Prentice-Hall, 1992, ISBN: 0-13-484080-1
Looks at the question of interoperability. Particulary interesting: chapters about environments and standards.

[Mull93] Sape Mullender (Ed.), *Distributed Systems* (Second Edition), Addison-Wesley, 1993, ISBN: 0-201-62427-3

[OSF DCE] Open Software Foundation, *Introduction to OSF^{TM} DCE*, Prentice-Hall, 1992, ISBN: 0-13-490624-1
Provides an introduction to the OSF Distributed Computing Environment.

[Pab93] Uday O. Pabrai, *UNIX Internetworking*, Artech House, 1993, ISBN: 0-89006-685-X
Gives an overview of all subjects that are related to UNIX networking, including programming principles, user commands, and system configuration.

[Part95] Craig Partridge, Jim Hughes, Jonathan Stone, *Performance of Checksums and CRCs over Real Data*, Computer Communications Review, Volume 25, Number 4, October 1995, pages 68-76, ACM Press

[POSIX.1] Donald Lewine, *POSIX Programmer's Guide*, O'Reilly & Associates, 1994, ISBN: 0-937175-73-0

[POSIX.4] Bill O. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly & Associates, 1995, ISBN: 1-56592-074-0

[RFC 768] J. Postel, *User Datagram Protocol*, RFC 768, USC/Information Sciences Institute, August 1980

[RFC 791] J. Postel (Ed.), *Internet Protocol - DARPA Internet Program Protocol Specification*, RFC 791, USC/Information Sciences Institute, September 1981

[RFC 793] J. Postel (Ed.), *Transmission Control Protocol - DARPA Internet Program Protocol Specification*, RFC 793, USC/Information Sciences Institute, September 1981

[RFC 1014] *XDR: External Data Representation Standard*, RFC 1014, Sun Microsystems, Inc., June 1987

[RFC 1057] *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 1057, Sun Microsystems, Inc., June 1988

[RFC 1071] R. Braden, D. Borman, C. Partridge, *Computing the Internet Checksum*, RFC 1071, September 1988 (updated by RFCs 1141 and 1624)

[RFC 1379] R. Braden, *Extending TCP for Transactions – Concepts*, RFC 1379, November 1992

[RFC 1880] J. Postel (Ed.), *Internet Official Protocol Standards*, RFC 1880, Internet Architecture Board, November 1995

[RFC 1883] S. Deering, R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 1883, Network Working Group, December 1995

[Shri95] Santosh Shrivastava, *private lecture notes of the module CSC309 - Distributed Systems*, held 1995/96 at the University of Newcastle upon Tyne

[Stall95] William Stallings, *Operating Systems* (Second Edition), Prentice-Hall, 1995, ISBN: 0-13-180977-6
Introduction to operating systems with the example systems Windows NT, UNIX, and MVS. Has also small appendix on queuing analysis.

[Stev90] Richard W. Stevens, *UNIX Network Programming*, Prentice-Hall, 1990, ISBN: 0-201-56317-7
A detailed book on UNIX network programming, with a lot of source code examples. Example source code can be found here:
`ftp://ftp.uu.net/published/books/stevens.advprog.tar.Z`

[Stev92] Richard W. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992, ninth printing 1995, ISBN: 0-201-56317-7
A good introduction to programming in the UNIX environment. Example source code can be found here:
`ftp://ftp.uu.net/published/books/stevens.netprog.tar.Z`

[Stev94] Richard W. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1994, ISBN: 0-201-63346-9
A complete and detailed guide to the entire TCP/IP protocol suite.

[Strou91] Bjarne Stroustrup, *The C++ Programming Language* (Second Edition), Addison-Wesley, 1991, ISBN: 0-201-53992-6
The standard book for learning C++, also called *The C++ Bible*.

[Salus95] Peter H. Salus, *Protocol Wars: Is OSI Finally Dead?*, ConneXions, Volume 9, No. 8, 1995, page 16-19, ISSN 0894-5926
Information on the change from GOSIP to POSIT, or why the importance of the OSI protocols will decrease more and more.

[Tan88] Andrew Tanenbaum, *Computer Networks* (Second Edition), Prentice-Hall, 1988, ISBN: 0-13-166836-6
A good, but a little bit out of date, general introduction in computer networks; based on the OSI 7-layer approach.

[Tan92] Andrew Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992, ISBN: 0-13-595752-4
An introduction into the principles of operating systems including UNIX case studies.

[Tom75] Raymond S. Tomlinson, *Selecting Sequence Numbers*, ACM SIGCOMM/SIGOPS Interprocess Communication Workshop, Santa Monica, CA, March 1975, reprinted in: Computer Communication Review, Volume 25, Number 1, January 1995, ACM Press

[Zlo91] Fred Zlotnick, *The POSIX.1 Standard: A Programmer's Guide*, The Benjamin/Cummings Publishing Company, 1991, ISBN: 08053-9605-5
A tutorial and reference to the POSIX.1 standard.

A Complete Source code

The source code for all programs is available under the URL
`http://www.ncl.ac.uk/~n5820119/project/prgsource.tgz` as `tar`ed `gzip` file. It can be extracted with

```
% gtar zxvf prgsource.tgz
```

on most UNIX systems that have the utilities GNU-`tar` and `gzip` installed.

The source code for the different programs are stored in different directories. This makes it easier to remember the purpose of a program or file.

Directory Name	**Purpose**
`common`	commonly used classes and subroutines
`local`	example programs for local interprocess communication
`distSOCKET`	example programs for distributed interprocess communication, socket version
`distTLI`	example programs for distributed interprocess communication, TLI version
`performance`	performance measurement programs
`environment`	example programs which get information about the environment in which they are executed
`rpc`	a modified RPC example from [Stev90]

Table 29: Overview of Source Code Directories

The programs were written portability in mind, as they had to run on three different computer platforms. The platform dependent parts normally deal only with the absence of include files and other easy to fix problems. The layout of all programs is derived from [POSIX.1].

Nearly all programs were tested and run under the systems described in Chapter 7, starting on page 70. The examples in the `distTLI` directory could only be tested on `aidan`, as this was the only platform that had the TLI support enabled.

Nearly all programs needed a common set of utility functions. Therefore these functions were collected in the `common` directory. The link-library `libcommon.a`, together with the include file `common.h` (page 112) allows all programs to access these functions easily.

The programs were not designed to be as short as possible on paper, they were designed to be maintainable. Therefore the layout of all programs is spacious. This was of great help whilst the programs were being developed.

The WWW-Version of this document has links to the generated data files and performance graphs of the three systems used.

A.1 Makefiles

Makefiles were used to keep control over the large number of programs and data files.

File Name	**Purpose**	**Page**
`Makefile`	Makefile for all programs	107
`Makefile.def`	general definitions for all other Makefiles	108
`Makefile.def2`	additional definitions for the end of all other Makefiles	109
`Makefile.hpux`	definitions for HP 9000/710/HP-UX 9.01	110
`Makefile.i386`	definitions for i586/Linux 1.2.9	110
`Makefile.sparc`	definitions for SparcCenter 2000/Solaris 2.5	110

Table 30: Makefiles

The common defines in `Makefile.def` (page 108) and `Makefile.def2` (page 109) allows all other Makefiles in the subdirectories to be reasonable short. Definitions suitable for the used system are read in automatically. To compile the programs on a new platform, only a new file `Makefile.`*hosttype* is needed, where *hosttype* is the contents of the environment variable `$HOSTTYPE`. This allows the compilation of the code on different platforms without any changes in the makefiles.

A.1.1 Makefile

```
# global Makefile for all source directories
# Sun Jan 28 22:23:07 MET 1996 GM: created
# 96/02/26 15:13:36 muellerg: distTLI added
# 96/03/17 05:53:51 muellerg: added perf2

all: libcommon.a
	(cd common; make)
	(cd local; make)
	(cd distSOCKET; make)
	(cd environment; make)
	(cd performance; make)
	(cd rpc; make)
	(cd distTLI; make)

lib:
	(cd common; make lib)

libcommon.a:
	(cd common; make lib)

clean:
	(cd common; make clean)
```

```
	(cd local; make clean)
	(cd distSOCKET; make clean)
	(cd distTLI; make clean)
	(cd environment; make clean)
	(cd performance; make clean)
	(cd rpc; make clean)
	rm -f *.o *~

purge:
	(cd common; make purge)
	(cd local; make purge)
	(cd distSOCKET; make purge)
	(cd distTLI; make purge)
	(cd environment; make purge)
	(cd performance; make purge)
	(cd rpc; make purge)
	rm -f libcommon.a
```

A.1.2 Makefile.def

```
# Common makefile definitions for my makefiles
# look also in Makefile.def2, which is included after everything else
# Sun Jan 28 22:03:55 MET 1996 GM: created
# 96/02/21 04:08:51 muellerg: updated for use with HP-UX and Linux
# 96/03/20 15:41:11 muellerg: made Makefile platform independent

# global variables

SHELL = /bin/sh
.SUFFIXES:
.SUFFIXES: .c .o .eps .data


# Definitions required in all program directories to compile and link
# C programs.


# Definitions for specific host

include ../Makefile.$(HOSTTYPE)

# Common definitions

CFLAGS=
TRAGET_ARCH=
LDFLAGS=
LOADLIBES=

# Our library that almost every program needs.

COMMON = ../libcommon.a


# Common temp files to delete from each directory.

TEMPFILES = core core.* *.o
```

```
# Rules

LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) #$(TARGET_ARCH)

# How to generate an executable from an object file

.o:
	$(LINK.o) $< $(LOADLIBES) $(LDLIBS) -o $@
	$(STRIP) $@

# How to generate an executable from a C source file

.c:
	$(LINK.c) $< $(LOADLIBES) $(LDLIBS) -o $@
	$(STRIP) $@

# How to generate an executable from a C++ source file

.cc:
	$(LINK.c) $< $(LOADLIBES) $(LDLIBS) -o $@
	$(STRIP) $@
```

A.1.3 Makefile.def2

```
# second part of general makefile definitions
# Sun Jan 28 22:06:21 MET 1996
# 96/02/21 04:10:25 muellerg: updated for HP-UX


# How to generate dependency files
# uncomment right lines for Linux and dependency files
#dep: *.c *.h
#	$(CPP) -M -MM *.c >dep	# Linux


# How to generate ../libcommon.a

../libcommon.a:
	(cd ..; make lib)



# clean out some rubbish

clean:
	rm -f *.o *~


# and even kill the executables, together with the dependency file

purge: clean
	rm -f $(ALL) dep


#
# include a dependency file if one exists (otherwise one is creates, see above)
```

```
#
# uncomment this for Linux and dependency files
#ifeq (dep,$(wildcard dep))
#include dep
#endif
```

A.1.4 Makefile.hpux

```
# definitions for HP-UX 9.01

CC      = CC -O
CPP     = CC -E
CXX     = CC -O
STRIP   = strip
RANLIB  = echo "ranlib not necessary"

CPPFLAGS = -O

LDLIBS = -L.. -lcommon
```

A.1.5 Makefile.i386

```
# Definitions for Linux 1.2.9 (PC)

CC      = g++
CPP     = g++ -E
CXX     = g++
STRIP   = strip
RANLIB  = ranlib

CPPFLAGS = -O3 -Wall

LDLIBS = -L.. -lcommon
```

A.1.6 Makefile.sparc

```
# Definitions for Solaris 2.5 (aidan)

CC      = g++
CPP     = g++ -E
CXX     = g++
STRIP   = strip
RANLIB  = echo "ranlib not possible"

CPPFLAGS = -O3 -Wall

LDLIBS = -L.. -lcommon -lsocket -lnsl
```

A.2 Commonly used Subroutines

The `common` directory (see Table 31) contains various functions and classes that are used by the example or performance measurement programs. Most of the files are used to build a link-library `libcommon.a`, which is used together with `common.h` (page 112) in the main directory to be available for all the other programs.

File Name	**Purpose**	**Page**
`Makefile`	Makefile for this directory	112
`../common.h`	Header file which defines the functions and classes in this directory, used by all programs	112
`catchchildren.c`	install a signal handler that cares about zombies generated by fork	116
`clocktimer.h`	defines the class `clocktimer`, which measures used clock time	117
`clocktimer.c`	implements the class `clocktimer`	117
`cputimer.h`	defines the class `cputimer`, which measures used CPU time	122
`cputimer.c`	implements the class `cputimer`	124
`errordekl.c`	declares global error object `error`	133
`errors.h`	defines the class `errors` that makes error handling easier	127
`errors.c`	implements the class `errors`	130
`measurement.h`	defines the class `measurement` that makes it easier to measure performances	135
`measurement.c`	implements the class `measurement`	137
`readdata.c`	for performance measurements: read the data generated by writedata(). The time used to read all the data can be written out	146
`readline.c`	provide a function that reads a line from cin and recognizes EOF immediately	148
`readline2.c`	read a line from a file descriptor	150
`readn.c`	simulate normal I/O read() for sockets, pipes,...	152
`readwritedata.h`	common definitions for readdata.c and writedata.c	154
`ssleep.c`	higher resolution sleep function	155
`synchronize.c`	functions that makes it possible for two processes to synchronize themselves	157

File Name	Purpose	Page
`writedata.c`	for performance measurements: write data read by readdata(). The time used to write all the data can be written out	162
`writen.c`	simulate normal I/O read() for sockets, pipes, ...	164

Table 31: Purpose of Files in the `common` Directory

A.2.1 common/Makefile

```
include ../Makefile.def

ALL = testerror
LIB = errors.o errordekl.o readline.o cputimer.o clocktimer.o \
        readn.o writen.o readline2.o synchronize.o \
        writedata.o readdata.o measurement.o ssleep.o catchchildren.o

all: $(COMMON) $(LIB) $(ALL)

lib: $(LIB)
    rm -f libcommon.a
    ar cq libcommon.a $(LIB)
    $(RANLIB) libcommon.a
    cp libcommon.a ../

include ../Makefile.def2
```

A.2.2 common.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: common.h
////////////////////////////////////////////////////////////////////////////////
// Purpose: includes common other include files, including non-std. ones
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/05 23:24:53 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////

#ifndef __COMMON_H__
#define __COMMON_H__

// not all compilers have buildin bool, so provide it if necessary
```

```
#ifndef linux
#ifndef __GNUC__
#ifndef DEFINED_BOOL
#define DEFINED_BOOL
enum bool { false = 0, true = 1};
#endif /* DEFINED_BOOL */
#endif /* __GNUC__ */
#endif /* linux */


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <sys/types.h>


// Local headers //////////////////////////////////////////////// Local headers //

#include "common/errors.h"
#include "common/clocktimer.h"
#include "common/cputimer.h"
#include "common/measurement.h"

#ifdef sun
// sun has not defined these
#ifndef INADDR_NONE
#define INADDR_NONE    ((unsigned long)0xffffffff)    /* -1 return */
#endif
// this is missing in unistd.h
extern "C" {
int gethostname(char *hostname, size_t size);
}
// sun does not have strings.h. This is the contents of strings.h:
extern "C" {
     extern char *index(const char *, char);
     extern char *rindex(const char *, char);
     extern void bcopy(const char *, char *, int);
     extern int bcmp(const char *, const char *, int);
     extern void bzero(char *, int);
     extern int ffs(int);
}
#endif /* sun */


// Macros ///////////////////////////////////////////////////////////// Macros //

// needed for TLI examples

#define DEV_UDP         "/dev/udp"      /* TLI udp device name */
#define DEV_TCP         "/dev/tcp"      /* TLI tcp device name */
```

```
#ifdef sun
#define WAIT_ANY ((pid_t)-1)
#endif
#ifdef hpux
#define WAIT_ANY ((pid_t)-1)
#endif

// used in various example programs as maximum length of lines

#define MAXLINE         255             /* maximum line length */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //

/* common/readline.h */

// reads a line with recognizing EOF immediately
int readline(istream &in, char *buf, int max_len);


/* common/readn.c */
// read numbytes bytes, even if individual read()s are not fully successful
int readn(int fd, char* ptr, int numbytes);


/* common/writen.h */
// write numbytes bytes, even if individual write()s are not fully successful
int writen(int fd, char* ptr, int numbytes);


/* common/readline2.h */
// read a line from a file descriptor
int readline(int fd, char* ptr, int maxlen);


/* local/synchronize.c */

// Initialize everything (set up signal handler etc.)
void synchronization_init(void);

// Send signal from child to parent process
void synchronization_signal_parent(pid_t pid);

// Send signal from parent to child
void synchronization_signal_child(pid_t pid);

// wait for signal (from parent or child)
void synchronization_wait(void);

// reset signal mask to how it was when calling synchronization_init()
void synchronization_reset(void);
```

```
/* if readdata() and writedata() should be synchronized via
 *  signals, both have to be initialized by synchronization_init()!
 */

/* local/readdata.c */
// reads data generated by writedata() and measures how long this takes
bool readdata(char *basename, char *ext, int fd, pid_t pid=0);


/* local/writedata.c */
// writes data for readdata() and measures how long this takes
bool writedata(char *basename, char *ext, int fd, pid_t pid=0);


/* common/ssleep.c */
// sleep for usecs microseconds
void ssleep(int usecs);


/* common/catchchildren.c */
// install a SIGCHLD catching function so zombies are avoided
void catch_children(void);


/* distSOCK/safesend.c */
// safe sendto(), same parameters
int safesendto(int s, const void *msg, int len, unsigned int flags,
                                    const struct sockaddr *to, int tolen);

// safe recvfrom(), same parameters
int saferecvfrom(int s, void *buf, int len, unsigned int flags,
                                        struct sockaddr *from, int *fromlen);



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

inline int max(int a, int b)
{
    return (a>b) ? a : b;
}



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */

#endif /* __COMMON_H__ */
```

A.2.3 common/catchchildren.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: catchchildren.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: install a signal handler that cares about zombies generated by fork
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 13:42:12 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>



// Local headers //////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //

static void
sig_child(int)      /* signal handler for SIGCHLD */
{
```

```
        waitpid(WAIT_ANY, NULL, WNOHANG);
        return;
}




// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

void catch_children(void)
{
    struct sigaction handle_sigchild;
    handle_sigchild.sa_handler = &sig_child;
    sigemptyset(&handle_sigchild.sa_mask); // no need to block other signals

#ifdef SA_RESTART
    handle_sigchild.sa_flags     = SA_RESTART;  // restart interrupted
                                                // system calls automatically
#else
    handle_sigchild.sa_flags     = 0;   // HP-UX 9.01 does not support this
                                        // option
#endif

    if(sigaction(SIGCHLD, &handle_sigchild, NULL) != 0)
        error.system("catch_children(): sigaction error");
}



// Main ///////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.4 common/clocktimer.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: clocktimer.h
////////////////////////////////////////////////////////////////////////////////
// Purpose: class definition for "clocktimer", a timer which measures
//          real time
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 01:01:29 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLOCKTIMER_H__
#define __CLOCKTIMER_H__
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */


// System headers /////////////////////////////////////////// System headers //

#include <sys/time.h>
#include <iostream.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //


/*
 * The class "clocktimer" is useful for measuring real clock time.
 * It uses the UNIX function gettimeofday() to get the needed information.
 */

class clocktimer
{
public:
    // constructor, initialize everything
    clocktimer(void):gottime(false),started(false){}

    // destructor, nothing to do
    ~clocktimer(void){}

    // return the number of measured seconds or -1 for an error
    long secs(void);
```

```
    // return the number of measured microseconds or -1 for an error
    long usecs(void);

    // start measurement
    bool start(void);

    // end measurement
    bool end(void);

private:
    bool gottime;       // do we have a measured time we can return?
    bool started;       // is the timer started?
    timeval tstart;     // the start time
    timeval tresult;    // the result time
};



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */

#endif /* __CLOCKTIMER_H__ */
```

A.2.5 common/clocktimer.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: clocktimer.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: implementation of class "clocktimer", a class that measures
//          real time
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 01:30:29 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //
    /* NONE */
```

```
// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //


/*
 * start timer
 */

bool
clocktimer::start(void)
{
    gottime = false;

    if(gettimeofday(&tstart, NULL) != 0)
    {
        cerr << "clocktimer::start:gettimeofday() error" << endl;
        return false;
    }

    started = true;
    return true;
}


/*
 * stop timer and calculate result
 */
```

```
bool
clocktimer::end(void)
{
    timeval tend;

    if(!started)
    {
        cerr << "clocktimer::end(): called without calling"
             << " clocktimer::start() first!" << endl;
        return false;
    }

    if(gettimeofday(&tend, NULL) != 0)
    {
        cerr << "clocktimer::end():gettimeofday() error" << endl;
        return false;
    }

    if(tend.tv_usec < tstart.tv_usec)
    {
        int nsec = (tstart.tv_usec - tend.tv_usec) / 1000000 + 1;
        tstart.tv_usec -= 1000000 * nsec;
        tstart.tv_sec += nsec;
    }

    if(tend.tv_usec - tstart.tv_usec > 1000000)
    {
        int nsec = (tstart.tv_usec - tend.tv_usec) / 1000000;
        tstart.tv_usec += 1000000 * nsec;
        tstart.tv_sec -= nsec;
    }

    tresult.tv_sec = tend.tv_sec - tstart.tv_sec;
    tresult.tv_usec = tend.tv_usec - tstart.tv_usec;

    gottime = true;
    started = false;

    return true;
}


/*
 * return number of measured seconds or -1 for an error
 */

long
clocktimer::secs(void)
{
    if(gottime)
        return tresult.tv_sec;
    else
        return -1;
}


/*
 * return number of measured microseconds or -1 for an error
```

```
 */

long
clocktimer::usecs(void)
{
    if(gottime)
        return tresult.tv_usec;
    else
        return -1;
}


// Main //////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.6 common/cputimer.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: cputimer.h
////////////////////////////////////////////////////////////////////////////////
// Purpose: declares a class "cputimer" that measures the cpu time
//          (user+system, without children)
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  -------------------------------------------------
// 96/02/06 01:08:26 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


#ifndef __CPUTIMER_H__
#define __CPUTIMER_H__

// Feature test switches ////////////////////////////// Feature test switches //
    /* NONE */



// System headers //////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <iostream.h>
#include <sys/times.h>



// Local headers ////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros //////////////////////////////////////////////////////////// Macros //
    /* NONE */
```

```
// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * The class "cputimer" is useful for measuring the used cpu time of a process.
 * It uses the UNIX function times() to get the needed information
 * The measured time is the sum of used user and system time
 */

class cputimer
{
public:

    // constructor, initialize everything
    cputimer(void);

    // destructor, nothing to do
    ~cputimer(void){}

    // return the number of measured seconds or -1 for an error
    long secs(void);

    // return the number of measured microseconds or -1 for an error
    long usecs(void);

    // start measurement
    bool start(void);

    // end measurement
    bool end(void);

private:
    bool gottime;           // do we have a measured time we can return?
    bool started;           // is the timer started?
    tms tstart;             // the start time
    tms tresult;            // the result time
```

```
    long ticks_per_sec;     // number of ticks per second, derived with sysconf
};



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */

#endif /* __CPUTIMER_H__ */
```

A.2.7 common/cputimer.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: cputimer.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: implementation of class "cputimer" which measures cpu time
//          (user+system, without children)
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 01:10:57 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
#include "cputimer.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */
```

```
// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */

/*
 * constructor: get the speed of the times() clock
 */

cputimer::cputimer(void):gottime(false),started(false)
{
    if( (ticks_per_sec=sysconf(_SC_CLK_TCK)) < 0)
        error.system("cputimer::cputimer(): sysconf(): can't get CLK_TCK");
}


/*
 * start the cpu timer
 */

bool
cputimer::start(void)
{
    gottime = false;

    if(times(&tstart) == (clock_t)(-1))
    {
        cerr << "cputimer::start:times() error" << endl;
        return false;
    }

    started = true;
    return true;
}


/*
 * end the cpu timer and calculate result
 */

bool
cputimer::end(void)
{
    tms tend;

    if(!started)
    {
        cerr << "cputimer::end(): called without calling"
```

```
            << " cputimer::start() first!" << endl;
        return false;
    }

    if(times(&tend) == (clock_t)(-1))
    {
        cerr << "cputimer::end(): times() error" << endl;
        return false;
    }

    tresult.tms_utime = tend.tms_utime - tstart.tms_utime;
    tresult.tms_stime = tend.tms_stime - tstart.tms_stime;
    tresult.tms_cutime = tend.tms_cutime - tstart.tms_cutime;
    tresult.tms_cstime = tend.tms_cstime - tstart.tms_cstime;

    gottime = true;
    started = false;

    return true;
}


/*
 * return number of measured seconds or -1 for an error
 */

long
cputimer::secs(void)
{
    if(gottime)
        return (tresult.tms_utime + tresult.tms_stime)/ticks_per_sec;
    else
        return -1;
}


/*
 * return number of measured microseconds or -1 for an error
 */

long
cputimer::usecs(void)
{
    if(gottime)
    {
        long ticks = tresult.tms_utime + tresult.tms_stime;
        long secs  = ticks/ticks_per_sec;
        long rest  = ticks - secs*ticks_per_sec;
        long multiplier = 1000000/ticks_per_sec;

        return rest * multiplier;
    }
    else
        return -1;
}


// Main /////////////////////////////////////////////////////////////// Main //
```

```
    /* NONE */
```

A.2.8 common/errors.h

```
///////////////////////////////////////////////////////////////////////////////
// Filename: errors.h
///////////////////////////////////////////////////////////////////////////////
// Purpose: declaration of class "errors"
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ----------------------------------------------
// 96/01/29 00:43:20 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


#ifndef __ERRORS_H__
#define __ERRORS_H__

// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <stdarg.h>



// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */



// Macros /////////////////////////////////////////////////////////// Macros //

// these macros allow to give some information about where a bug was discovered
// from a program

#ifdef __GNUC__
#define BUG(x) error.bug(__PRETTY_FUNCTION__, __FILE__, __LINE__, x)
#else
#define BUG(x) error.bug(NULL, __FILE__, __LINE__, x)
#endif



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */
```

```
// External variables, functions, and classes ///////////// External objects //

class errors;
extern errors error;    // defined in errordekl.c



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //

/* Class errors is a convinient way to handle all the
 * different types of errors.
 * A debug level can be specified.
 * A program should use the function set_program_name()
 * to register its name. This makes it possible to destinguish
 * the errors from different programs.
 */

class errors
{
public:

    enum errorlevel /* the different error levels */
    {
        harmless,   /* a harmless error, continue */
        warn,       /* a warning, continue */
        error,      /* a real error, continue */
        fatal       /* a fatal error, exit the program */
    };

    enum debuglevel
    {
        high=1,   /* show all debug statements */
        medium=2, /* quite a lot */
        some=3,   /* show some of the debug statements */
        low=4,    /* show only important ones */
        none=5    /* don't show debug statements */
    };


    /* constructor, destructor */

    errors(char *name, debuglevel level=none)
        : program_name(name),
          error_debuglevel(level)

    {
        /* nothing to do */
    };
    ~errors(void){}

    errors(debuglevel level=none)
        : error_debuglevel(level)
```

```
    {
        program_name = "(program name not set)";
    };


    /* system errors */


    /* take the number in errno */
    void system(char *message = NULL, errorlevel err_level=fatal);

    /* error number specified in error_number */
    void system(int errno_number, char *message = NULL,
                errorlevel err_level=fatal);



    /* panic: terminate programm with message */

    void panic(char *message);


    /* bug */

    /* program bug. Always terminates a program */
    void bug(char *function, char *sourcefile, int lineno, char *message);


    /* warning, uses printf style for arguments */

    void warning(char *fmt,...);


    /* debugging */

    /* debug output */
    void debug(char *message, debuglevel dlevel=high);

    /* get/set debug level */
    debuglevel get_debug_level(void);

    debuglevel set_debug_level(debuglevel debugl);


    /* set program name */

    char *set_program_name(char *name);

private:
    char *program_name;          /* program name */
    debuglevel error_debuglevel; /* debuglevel */
};


// Functions and class implementation /// Functions and class implementation //
    /* NONE */
```

```
#endif /* __ERRORS_H__ */
```

A.2.9 common/errors.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: errors.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: implemenation of class "errors"
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 00:00:22 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <iostream.h>
#include <string.h>
#include <errno.h>
#include <stdarg.h>
#include <stdio.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "errors.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */
```

```
// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //


/*
 * get error from errno
 */

void errors::system(char *message, errorlevel err_level)
{
    system(errno, message, err_level);
}



/*
 * error number specified in error_number
 */

void errors::system(int errno_number, char *message, errorlevel err_level)
{
    cerr << program_name << ": ";

    if(message)
        cerr << message << ": ";

    cerr << "system error: "
         << strerror(errno_number) << endl;

    if(err_level==fatal)
        exit(EXIT_FAILURE);
}



/*
 * panic: terminate program with message
 */

void errors::panic(char *message)
{
    cerr << program_name << ": Panic";

    if(message)
        cerr << ": " << message;

    cerr << endl;

    exit(EXIT_FAILURE);
}
```

```
/*
 * program bug. Always terminates a program
 */

void errors::bug(char *function, char *sourcefile, int lineno, char *message)
{
    cerr << "Congratulations! You have discovered a bug in "
         << "the program \"" << program_name << "\"!" << endl;
    cerr << "The bug was detected here:" << endl;

    if(sourcefile)
        cerr << "file: " << sourcefile << endl;

    if(function)
        cerr << "function: " << function << endl;

    if(lineno)
        cerr << "line: " << lineno << endl;

    if(message)
        cerr << "Message: " << message << endl;

    exit(EXIT_FAILURE);
}




/*
 * print warning and return
 */

void errors::warning(char *fmt,...)
{
    va_list args;

    if(fmt)
    {
        char *emesgstr=new char[2000];

        if(emesgstr)
        {
            va_start(args, fmt);
            vsprintf(emesgstr, fmt, args);
            va_end(args);

            if(fmt)
                cerr << program_name << ": Warning: " << emesgstr << endl;

            delete[] emesgstr;
        }
        else
            cerr << program_name << ": Warning: not enough memory for "
                                                          "warning" << endl;
    }

    return;
}
```

```
/*
 * debug output
 */

void errors::debug(char *message, debuglevel dlevel)
{
    if(error_debuglevel != none)
    {
        if(dlevel >= error_debuglevel)
        {
            cerr << program_name << ":" << " Debug: "
                 << message << endl;
        }
    }
}



/*
 * get/set debug level
 */

errors::debuglevel errors::get_debug_level(void)
{
    return error_debuglevel;
}

errors::debuglevel errors::set_debug_level(debuglevel debugl)
{
    debuglevel old = error_debuglevel;
    error_debuglevel = debugl;
    return old;
}



/*
 * set program name
 */

char* errors::set_program_name(char *name)
{
    char *old = program_name;
    program_name = name;
    return old;
}
```

A.2.10 common/errordekl.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: errordekl.c
////////////////////////////////////////////////////////////////////////////////
```

```
// Purpose: declare global error object
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 00:05:14 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //
    /* NONE */



// Local headers ///////////////////////////////////////////// Local headers //

#include "errors.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //


// global definition of error class

errors error;



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
```

```
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.11 common/measurement.h

```
///////////////////////////////////////////////////////////////////////////////
// Filename: measurement.h
///////////////////////////////////////////////////////////////////////////////
// Purpose: defines a class "measurement" that makes it easier to measure
//          performances
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/28 22:49:38 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////

#ifndef __MEASUREMENT_H__
#define __MEASUREMENT_H__

// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //

// extensions for filenames of measurements

#define LOGFILE_EXTENSION ".log"
#define COMPLOGFILE_EXTENSION ".data"



// File scope objects ///////////////////////////////////// File scope objects //
    /* NONE */
```

```
// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //


// this structure holds information about one measurement

struct measurement_data
{
    int buffersize;     // number of bytes read or written in each turn
    int how_often;      // how often buffersize is written
    long cpu_secs;      // how long this took in cpu seconds
    long cpu_usecs;     // and microseconds
    long clock_secs;    // how long this took in time seconds
    long clock_usecs;   // and microseconds
    char *comment;      // string to comment this result
};



// The class "measurement" makes it much easier to measure the
// performance of something.
// It uses the classes cputimer and clocktimer to measure the time
// The data is measured with start() and end(). The measured data
// can be written out to cout or a file.
// A data file suitable for gnuplot can be generated as well.


class measurement
{
public:
    // define measurement environment, use name as base filename to write
    // data to, and define max. numbers of measurements

    measurement(char *name, char *ext, int max_number_of_measurements);
    ~measurement(void);

    // start measurement
    // buffersize: size of used buffer (needed to calculate the
    //             throughput
    // how_often:  the number how often the measurement is done with the
    //             given buffersize
    // comment:    a comment for a measurement.
    void start(int buffersize=-1, int how_often=0, char *comment=NULL);

    // end measurement
    void end(void);

    // reset measurement
    void reset(void);
```

```
    // these functions are for applications with kb/s and op/s
    // if use_cout is true, uses cout to display data instead of
    // creating a file
    void writeout_logfile(bool write_kb_per_sec  = true,
                          bool write_ops_per_sec = true,
                          bool use_cout = false);

    // generates data for gnuplot, therefore always creates a file
    void writeout_plain_logfile(bool write_kb_per_sec  = true,
                                bool write_ops_per_sec = false);

    // these two functions are for measurements where only the taken
    // time matters (therefore no kb/s)
    void writeout_histogram(bool use_cout = false);
    void writeout_plain_histogram(void);

private:

    void write_logfile(char *extension, bool verbal, bool write_kb_per_secs,
                        bool write_ops_per_sec, bool use_cout);
    void write_histogram(bool verbal, bool use_cout);

    struct measurement_data *data;
    int max_number;         // class can handle up to this measurements
    char *logbasename;
    char *logbasenameext;

    cputimer cput;          // measures used cpu time
    clocktimer clockt;      // measures clock time

    int actual;             // which measurement comes next?
};



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */

#endif /* __MEASUREMENT_H__ */
```

A.2.12 common/measurement.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: measurement.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: implements class "measurement" that makes it easier to measure
//          performances
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
```

```
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/28 22:59:21 muellerg: created
// 96/02/29 11:21:19 muellerg: histogram functions added
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <string.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */

/*
 * create measurement object that can handle up to
 * max_number_of_measurements measurements
 *
```

```
 * the log file name is derived from name
 */

measurement::measurement(char *name, char *ext, int max_number_of_measurements)
{
    logbasename = name;                          // set logfile name
    logbasenameext = ext;                        // and extension
    max_number = max_number_of_measurements;     // maximum number of
                                                 // measurements
    actual = 0;                                  // next extry to write to
    data = new measurement_data[max_number];     // get buffer data
    if(!data)
        error.panic("measurement::measurement(): out of memory");
}


/*
 * destuctor: delete buffer if we have one
 */

measurement::~measurement(void)
{
    if(data)
        delete[] data;
}


/*
 * start a mesurement as fast as possible
 */

void
measurement::start(int buffersize, int how_often, char *comment)
{
    if(actual < max_number)
    {
        data[actual].buffersize = buffersize;
        data[actual].how_often  = how_often;
        data[actual].comment    = comment;

        cput.start();
        clockt.start();
    }
    else
        error.panic("measurement::start(): buffer overflow");
}


/*
 * end a measurement and copy the data to an internal buffer as fast as
 * possible
 */

void
measurement::end(void)
{
    cput.end();
    clockt.end();
```

```
    data[actual].cpu_secs  = cput.secs();
    data[actual].cpu_usecs = cput.usecs();

    data[actual].clock_secs  = clockt.secs();
    data[actual].clock_usecs = clockt.usecs();

    actual++;
}


/*
 * reset measurements
 */

void
measurement::reset(void)
{
    actual = 0;

}


/*
 * write data out to logfile in human readable format
 */


void
measurement::writeout_logfile(bool write_kb_per_sec,
                              bool write_ops_per_sec,
                              bool use_cout)
{
    // write out human readable logfile

    write_logfile(LOGFILE_EXTENSION, true, write_kb_per_sec,
                  write_ops_per_sec, use_cout);
}


/*
 * write data out for use with other programs (e.g. gnuplot)
 */

void
measurement::writeout_plain_logfile(bool write_kb_per_sec,
                                    bool write_ops_per_sec)
{
    // write out human readable logfile

    write_logfile(COMPLOGFILE_EXTENSION, false, write_kb_per_sec,
                                                write_ops_per_sec, false);
}


/*
 * write logfile with extension
 */
```

```
void
measurement::write_logfile(char *extension, bool verbal,
    bool write_kb_per_sec, bool write_ops_per_sec, bool use_cout)
{
    ostream *out=0;

    char *namebuffer = NULL;
    ofstream out_f;

    if(use_cout)
        out = &cout;
    else
    {
        int bufferlen = strlen(logbasename) +
                        strlen(logbasenameext) +
                        strlen(extension) +2;
        namebuffer = new char[bufferlen];

        // construct file name
        strcpy(namebuffer, logbasename);
        strcat(namebuffer, ".");
        strcat(namebuffer, logbasenameext);
        strcat(namebuffer, extension);

        out_f.open(namebuffer);
        if(out_f)
            out = &out_f;
    }

    if(out)
    {
        // file is available

        if( (!verbal) && write_kb_per_sec && write_ops_per_sec)
            error.panic("measurement: can't define write_ops and write_kb"
                        " if !verbal");

        // write only as much data out as we have

        for(int i = 0; i < actual; i++)
        {
            // compute time

            float clockt_usecs  = data[i].clock_secs*1000000 +
                                    data[i].clock_usecs;

            float cput_usecs    = data[i].cpu_secs*1000000 +
                                    data[i].cpu_usecs;


            if(verbal)
            {
                if(data[i].comment)
                    *out << data[i].comment << endl;

                if(data[i].buffersize != -1)
                {
                    *out << "Result for buffersize "
```

```
                << data[i].buffersize << " (";
        }

        *out << "average over "
             << data[i].how_often << " operations";

        if(data[i].buffersize != -1)
            *out << "):";


        *out << endl;
    }


    if(!verbal)
    {
        if(data[i].buffersize != -1)
            *out << data[i].buffersize << " ";
    }

    if(write_kb_per_sec)
    {
        if(verbal)
            *out << "kb/s: ";

        *out << (float) ( (float)(data[i].buffersize *
                data[i].how_often) /
                (clockt_usecs/1000000.0)) / 1024.0;

        if(write_ops_per_sec)
            *out << "  ";
        else
            *out << endl;
    }

    if(write_ops_per_sec)
    {
        if(verbal)
            *out << "number of operations per second: ";

        *out << (float) ( (float)(data[i].how_often) /
                (clockt_usecs/1000000.0) ) << endl;
    }

    if(verbal)
        *out << "clock secs:"
             << (clockt_usecs/((float)1000000))/(float)data[i].how_often
             << "(="
             << clockt_usecs/(float)data[i].how_often
             << " usecs)"
             << "  cpu secs:"
             << (cput_usecs/((float)1000000))/(float)data[i].how_often
             << "(="
             << cput_usecs/(float)data[i].how_often
             << " usecs)"
             << endl << endl;

}  // for
```

```
    }
    else
    {
        if(use_cout)
            error.warning("cout error - can't write output");
        else
            error.warning("cant open file %s for writing", namebuffer);

        exit(EXIT_FAILURE);
    }

    if(namebuffer)
        delete[] namebuffer;
}


/*
 * write out histogram (for applications where only the time measurement
 * matter) -- verbose version
 */

void
measurement::writeout_histogram(bool use_cout)
{
    write_histogram(true, use_cout);
}


/*
 * write out histogram (for applications where only the time measurement
 * matter) -- data (gnuplot) version
 */

void
measurement::writeout_plain_histogram(void)
{
    write_histogram(false, false);
}


/*
 * write out histogram (for applications where only the time measurement
 * matter) -- generic version
 *
 * verbose: output min, max, average
 * !verbose: output raw data (numerated from 1 to actual)
 */


void
measurement::write_histogram(bool verbal, bool use_cout)
{
    ostream *out=0;

    char *namebuffer = NULL;
    ofstream out_f;

    if(use_cout)
```

```
    out = &cout;
else
{
    int bufferlen = strlen(logbasename) +
                    strlen(logbasenameext) +
                    max(strlen(COMPLOGFILE_EXTENSION),
                         strlen(COMPLOGFILE_EXTENSION)) + 2;

    namebuffer = new char[bufferlen];

    // construct file name
    strcpy(namebuffer, logbasename);
    strcat(namebuffer, ".");
    strcat(namebuffer, logbasenameext);

    if(verbal)
        strcat(namebuffer, LOGFILE_EXTENSION);
    else
        strcat(namebuffer, COMPLOGFILE_EXTENSION);

    out_f.open(namebuffer);
    if(out_f)
        out = &out_f;
}

if(out)
{
    // file is available


    // write only as much data out as we have

    float min_clockt, max_clockt;
    float min_cput, max_cput;

    float sum_clockt, sum_cput;

    // initialize

    min_clockt = max_clockt = data[0].clock_secs*1000000 +
                              data[0].clock_usecs;

    min_cput = max_cput = data[0].cpu_secs*1000000 +
                              data[0].cpu_usecs;

    sum_clockt = sum_cput = 0;


    for(int i = 0; i < actual; i++)
    {
        // compute time

        float clockt_usecs  = data[i].clock_secs*1000000 +
                              data[i].clock_usecs;

        float cput_usecs    = data[i].cpu_secs*1000000 +
                              data[i].cpu_usecs;
```

```
    // update statistics

    sum_clockt += clockt_usecs;
    sum_cput   += cput_usecs;


    // maximum numbers

    if(clockt_usecs > max_clockt)
        max_clockt = clockt_usecs;

    if(cput_usecs > max_cput)
        max_cput = cput_usecs;


    // minimum numbers

    if(clockt_usecs < min_clockt)
        min_clockt = clockt_usecs;

    if(cput_usecs < min_cput)
        min_cput = cput_usecs;



    // output raw data

    if(!verbal)
        *out << i+1 << " " << clockt_usecs << endl;

}   // for


if(verbal)
{
    // display comment of first measurement if we have one

    if(data[0].comment)
        *out << data[0].comment << endl;


    // average time

    *out << "Average time: clock secs: "
         << ((sum_clockt/(float)actual)/1000000.0)/(float)actual
         << "(=" << sum_clockt/(float)actual
         << " usecs)"
         << "  cpu secs: "
         << ((sum_cput/(float)actual)/1000000.0)/(float)actual
         << "(=" << sum_cput/(float)actual
         << " usecs)"
         << endl;


    // minimum time

    *out << "Minimum time: clock secs: "
         << min_clockt/1000000.0
```

```
                 << "(=" << min_clockt
                 << " usecs)"
                 << "  cpu secs: "
                 << min_cput/1000000.0
                 << "(=" << min_cput
                 << " usecs)"
                 << endl;

            // maximum time

            *out << "Maximum time: clock secs: "
                 << max_clockt/1000000.0
                 << "(=" << max_clockt
                 << " usecs)"
                 << "  cpu secs: "
                 << max_cput/1000000.0
                 << "(=" << max_cput
                 << " usecs)"
                 << endl;
        } // verbal
    }
    else
    {
        if(use_cout)
            error.warning("cout error - can't write output");
        else
            error.warning("cant open file %s for writing", namebuffer);

        exit(EXIT_FAILURE);
    }

    if(namebuffer)
        delete[] namebuffer;
}


// Main ////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.13 common/readdata.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: readdata.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: for performance measurements: read the data generated by
//          writedata(). The time used to read all the data is displayed.
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/27 18:32:15 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */


// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
#include "readwritedata.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //

/*
 * readdata(): read the data generated by writedata() and writes out how
 *             long it took to read all the data
 *             the function needs a descriptor to read from, so it is
 *             only usable for IPC facilities which allow reading and writing
 *             data to descriptors.
 *
 * in: name: base name for logfile(s)
 *      ext: extension (e.g. "read" or "write")
 *       fd: descriptor to read from
 *      pid: process to synchronize with (if 0, no synchronization is done,
 *           e.g. if processes run on different computers...)
 *           to use these facility the caller has to install a special
 *           signal handler with  synchronization_init();
 *
```

```
 * returns: true if everything is ok, else false
 *
 */

bool
readdata(char *name, char *ext, int fd, pid_t pid)
{
    char *buffer=new char[size_max];
    if(!buffer) return false;

    measurement mes(name, ext, measurement_max);

    int i, size;

    for(size=1; size <= size_max; size *= 2 )
    {
        if(pid)
        {
            // signal other process that we are ready to read

            synchronization_signal_parent(pid);

            // and wait for acknowledgment

            synchronization_wait();
        }


        // start timer

        mes.start(size, how_often);

        for(i=0; i < how_often; i++)
        {
            readn(fd, buffer, size);
        }

        // store data about measurement

        mes.end();
    }

    mes.writeout_logfile();
    mes.writeout_plain_logfile();

    delete[] buffer;

    return true;
}

// Main ////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.14 common/readline.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: readline.c
```

```
//////////////////////////////////////////////////////////////////////////////
// Purpose: provide a function that reads a line from cin and recognizes
//          EOF immediately
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  -----------------------------------------------
// 96/02/06 00:44:30 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////// System headers //

#include <stdlib.h>



// Local headers //////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
```

```
/*
 * readline: reads a line from istream in, with recognizing EOF
 *           immediately.
 * returns:  -1 for error, 0 for ok
 */

int
readline(istream &in, char *buf, int max_len)
{
    int ok = -1;

    if(buf && max_len)
    {
        char *t=buf;
        bool no_return = true;

        while(max_len && in && no_return)
        {
            in.read(t,1);

            if(*t == '\n')
                no_return=false;

            t++;
            max_len--;

        }

        *t=0;

        if(in) ok=0;
    }
    return ok;
}



// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.15 common/readline2.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: readline2.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: read a line from a file descriptor
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/12 02:26:27 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include <unistd.h>



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * Read a line from a descriptor.  Read the line one byte at a time,
 * looking for the newline.  We store the newline in the buffer,
 * then follow it with a null (the same as fgets(3)).
 * We return the number of characters up to, but not including,
 * the null (the same as strlen(3)).
 *
 * This function is taken from [Stev90].
 */


int
readline(int fd, char* ptr, int maxlen)
{
```

```
    int n, rc;
    char    c;

    for (n = 1; n < maxlen; n++)
    {
        if ( (rc = read(fd, &c, 1)) == 1)
        {
            *ptr++ = c;
            if (c == '\n')
                break;
        }
        else
            if (rc == 0)
            {
                if (n == 1)
                    return(0);  // EOF, no data read
                else
                    break;      // EOF, some data was read
            }
            else
                return(-1);     // error
    }

    *ptr = 0;
    return(n);
}


// Main ////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.16 common/readn.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: readn.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: simulate normal I/O read() for sockets, pipes,...
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/12 01:09:58 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

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

```
#include <unistd.h>


// Local headers //////////////////////////////////////////// Local headers //
    /* NONE */


// Macros ////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects //////////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //

/*

Taken from [Stev92], page 406: (code from page 408)

------------------------------------------------------------------------------
Some devices, notably terminals, networks, and any SVR4 streams devices,
have the following two properties.

 1. A "read" operation may return less than asked for, even though we have not
    encountered the end of file. This is not an error, and we should just
    continue reading from the device.

 2. A "write" operation cam also return less than we specified. This may be
    caused by flow control constraints by downstream modules, for example.
    Again, it's not an error, and we should continue writing the reminder of
    the data. (Normally this short return from "write" only occurs with a
    nonblocking descriptor if a signal is caught.)
------------------------------------------------------------------------------

*/

// normal read arguments
```

```
int
readn(int fd, char* ptr, int numbytes)
{
    int nleft, nread;

    nleft = numbytes;
    while (nleft > 0)
    {
        nread = read(fd, ptr, nleft);
        if (nread < 0)
            return(nread);      /* error, return < 0 */
        else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(numbytes - nleft);       /* return >= 0 */
}


// Main ////////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.17 common/readwritedata.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: readwritedata.h
////////////////////////////////////////////////////////////////////////////////
// Purpose: common sizes for readdata.c and writedata.c
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/27 18:42:46 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers ///////////////////////////////////////////// System headers //
    /* NONE */



// Local headers ////////////////////////////////////////////// Local headers //
    /* NONE */
```

```
// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes //////////// External objects //

const int size_max = 1024*256;  // 256 k max
const int how_often = 1000;     // 1000 times

const int measurement_max = 30; // maximum number of measurements


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.18 common/ssleep.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: ssleep.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: higher resolution sleep function
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 13:14:26 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */
```

```
// System headers ////////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>


// Local headers //////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ////////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * ssleep() - sleeps for a specific time given in microseconds
 *
 * ssleeps is designed for sleeping only less than a second (could easily
 * be changed)
 */

void ssleep(int usecs)
{
    fd_set zero;
    FD_ZERO(&zero);

    struct timeval t;
    t.tv_sec  = 0;
    t.tv_usec = usecs;
```

```
#ifndef hpux
    select(0, &zero, &zero, &zero, &t);
#else
    select(0, NULL, NULL, NULL, &t);
#endif
}



// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.19 common/synchronize.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: synchronize.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: functions that makes it possible for two processes to
//          synchronize themselves
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/26 22:53:51 muellerg: created
// 96/02/29 01:07:55 muellerg: added synchronization_reset
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <signal.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

static volatile sig_atomic_t sigflag;       // set to nonzero by signal handler
static sigset_t newmask, oldmask, zeromask;
```

```
// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //

static void
sig_usr(int)    /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
    return;
}




// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/* These functions enable it for two unrelated processes to synchronize.
 * This implementation uses signals (SIGUSR1 and SIGUSR2), but it would
 * also be possible to use e.g. pipes.
 *
 * To destinguish the two processes one is called the parent, the
 * other one the child. This must not be true in reality.
 *
 * The processes to use the following functions have to decide which
 * one is the parent and which one is the child. This decision can be
 * changed while the program is running.
 *
 * The following functions are available:
 *
 * void synchronization_init(void): initialize everything,
 *      has to be called before any of the other functions can be used
 *
 *
 * void synchronization_signal_parent(pid_t pid): send signal SIGUSR2
 *      to pid
 *
 * void synchronization_signal_child(pid_t pid): send signal SIGUSR1
 *      to pid
 *
 * void synchronization_wait(void): wait for signal (either from
 *      child or parent)
 *
 * void synchronization_reset(void): reset signal mask to how it was when
 *      calling synchronization_init()
 */
```

```
/*
 * Initialize everything
 */

void
synchronization_init(void)
{
    // set up signal handler

    struct sigaction handle_sigusr;
    handle_sigusr.sa_handler = &sig_usr;
    sigemptyset(&handle_sigusr.sa_mask);   // no need to block other signals
    handle_sigusr.sa_flags   = 0;

    if(sigaction(SIGUSR1, &handle_sigusr, NULL) != 0)
        error.system("synchronization_init(): sigaction() error");

    if(sigaction(SIGUSR2, &handle_sigusr, NULL) != 0)
        error.system("synchronization_init(): sigaction() error");


    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    // block SIGUSR1 and SIGUSR2, and save current signal mask

    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        error.system("synchronization_init(): SIG_BLOCK error");
}


/*
 * Send signal from child to parent process
 */

void
synchronization_signal_parent(pid_t pid)
{
    kill(pid, SIGUSR2);
}



/*
 * Send signal from parent to child
 */

void
synchronization_signal_child(pid_t pid)
{
    kill(pid, SIGUSR1);
}
```

```
/*
 * wait for signal (from parent or child)
 */

void
synchronization_wait(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  // wait for signal

    sigflag = 0;    // reset signal mask to original value
}


/*
 * reset signal mask to how it was when calling synchronization_init()
 */

void
synchronization_reset(void)
{
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        error.system("synchronization_reset(): SIG_SETMASK error");
}


// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.20 common/testerror.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: testerror.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: tests class error
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 00:09:17 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <errno.h>
#include <iostream.h>
```

```
// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */
#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);


    errno=EPERM;
    error.system("system error test EPERM", errors::harmless);

    error.system(ENOENT, "system error test ENOENT", errors::harmless);

    error.warning("Test warning 1");
    error.warning("Test warning %d", 2);
    error.warning("Test warning %s %d", "three", 3);

    error.set_debug_level(errors::high);
    error.debug("debug test 1", errors::high);
```

```
    error.set_debug_level(errors::low);
    error.debug("debug test 2", errors::high);

    BUG("Bug test!");


    /* never reached */
    return(EXIT_SUCCESS);
}
```

A.2.21 common/writedata.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: writedata.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: for performance measurements: write data read by
//          readdata(). The time used to write all the data is displayed.
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/27 18:57:56 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
#include "readwritedata.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */
```

```
// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * writedata(): write data which is read by readdata() and writes out how
 *              long it took to write all the data
 *              the function needs a file descriptor to read from, so it is
 *              only usable for IPC facilities which allow reading and writing
 *              data to descriptors.
 *
 * in: name: base name for logfile(s)
 *      ext: extension (e.g. "read" or "write")
 *       fd: descriptor to write to
 *      pid: process to synchronize with (if 0, no synchronation is done,
 *           e.g. if processes run on different computers...)
 *           to use these facility the caller has to install a special
 *           signal handler with  synchronization_init();
 *
 *
 * return: true if everything is ok, else false
 *
 */

bool
writedata(char *name, char *ext, int fd, pid_t pid)
{
    char *buffer=new char[size_max];
    if(!buffer) return false;

    int i, size;

    measurement mes(name, ext, measurement_max);

    for(size=1; size <= size_max; size*=2 )
    {
        if(pid)
        {
            // wait for start signal from other process
            synchronization_wait();

            // and acknowledge

            synchronization_signal_child(pid);
        }


        // start timer
```

```
        mes.start(size, how_often);

        for(i=0; i < how_often; i++)
        {
            writen(fd, buffer, size);
        }

        // store data about measurement

        mes.end();
    }

    mes.writeout_logfile();
    mes.writeout_plain_logfile();

    delete[] buffer;

    return true;
}

// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.2.22 common/writen.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: writen.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: simulate normal I/O write() for sockets, pipes,...
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/12 01:11:10 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>



// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */
```

```
// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*

Taken from [Stev92], page 406: (code from page 408)

-------------------------------------------------------------------------------
Some devices, notably terminals, networks, and any SVR4 streams devices,
have the following two properties.

 1. A "read" operation may return less than asked for, even though we have not
    encountered the end of file. This is not an error, and we should just
    continue reading from the device.

 2. A "write" operation cam also return less than we specified. This may be
    caused by flow control constraints by downstream modules, for example.
    Again, it's not an error, and we should continue writing the reminder of
    the data. (Normally this short return from "write" only occurs with a
    nonblocking descriptor if a signal is caught.)
-------------------------------------------------------------------------------

*/

// normal write arguments

int
writen(int fd, char* ptr, int numbytes)
{
    int nleft, nwritten;

    nleft = numbytes;
    while (nleft > 0)
```

```
    {
        nwritten = write(fd, ptr, nleft);
        if (nwritten <= 0)
            return(nwritten);   /* error */

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(numbytes - nleft);
}



// Main /////////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.3 Local IPC

The `local` directory (see Table 32) contains the example programs for Chapter 3. They implement various local IPC mechanisms.

File Name	Purpose	Page
`Makefile`	Makefile for this directory	167
`fifoclient.c`	together with fifoserver.c: demonstrates the use of FIFOs	167
`fifoserver.c`	together with fifoclient.c: demonstrates the use of FIFOs	169
`message.c`	shows how to use System V IPC message queues	171
`namedstreampipe.c`	implements function that creates a named stream pipe	175
`pipe.c`	demonstrates the use of UNIX pipes	177
`semaphore.c`	shows how to use System V IPC semaphores	179
`shared.c`	shows how to use System V IPC shared memory	184
`signal.c`	demonstrates the use of signals	186

Table 32: Purpose of Files in the `local` Directory

A.3.1 local/Makefile

```
include ../Makefile.def

ALL = pipe fifoserver fifoclient message semaphore shared signal namedstreampipe

all: $(ALL)

include ../Makefile.def2
```

A.3.2 local/fifoclient.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: fifoclient.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: together with fifoserver.c: demonstrates the use of fifos
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
```

```
// -------- -------- --------  ---------------------------------------------
// 96/02/06 02:22:30 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const char *fifo_file="example_fifo";



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////// Main //
```

```
/*
 * This program sends a string (the first argument to this program)
 * to a FIFO, which can be read by fifoserver
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if ( argc != 2 )
    {
        cout << argv[0] << " usage: fifo_client [string]" << endl;
        exit(EXIT_FAILURE);
    }

    ofstream out(fifo_file);
    if(out)
        out << argv[1] << endl;

    return(EXIT_SUCCESS);
}
```

A.3.3 local/fifoserver.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: fifoserver.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: together with fifoclient.c: demonstrates how to use FIFOs in UNIX
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 02:25:59 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */




// System headers /////////////////////////////////////////// System headers //

#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fstream.h>
#include <string.h>
```

```
// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const char *fifo_file = "example_fifo";
const mode_t permission =
    (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
// = 0666 = rw-rw-rw-


// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////////// Main //

/*
 * This program displays strings which are written to a FIFO by fifoclient.
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    char buffer[80];

    // Create FIFO

    umask(0);

    mkfifo(fifo_file, permission);
```

```
    // this would be equivalent:
    // mknod(fifo_file, S_IFIFO|permission, 0);

    while(true)
    {
        ifstream in(fifo_file);
        if(in)
        {
            while(in)
            {
                in.getline(buffer, 80);
                if(strlen(buffer))
                {
                    cout << "Received: " << buffer << endl;
                    if(strncmp(buffer, "quit", 4) == 0)
                    {
                        unlink(fifo_file);
                        exit(EXIT_SUCCESS);
                    }
                }
            }
        }
        else
        {
            cerr << "can't open fifo file!" << endl;
            exit(EXIT_FAILURE);
        }
    }

    return(EXIT_SUCCESS);
}
```

A.3.4 local/message.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: message.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: shows how to use System V IPC message queues
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/19 02:57:14 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////////// System headers //

#include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int N = 10;               // empty buffer slots at the beginning

const int produceitems = 100;   // how much items to produce alltogether
                                // should be lower than the maximum messages
                                // per message queue divided by three

const int MSG_MODE = 0600;      // only processes of same user have access to
                                // the message queue

const int consumer = 1;         // type for consumer
const int producer = 2;         // type for producer


// External variables, functions, and classes //////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //


// messages to be passed arount look loke this

struct message
{
    long msgtype;       // who sent message? producer or consumer?
                        // (has to be a positive long)
                        // needed by message queue system calls

    long data;          // information to transfer
};
```

```
// Functions and class implementation /// Functions and class implementation //

/*
 * build message containing the given number.
 */

void build_message(message *msg, int i)
{
    if(msg)
    {
        msg->data = i;
    }
}


/*
 * extract item from message
 */

void extract_item(message *msg, int *i)
{
    if(msg)
    {
        *i = msg->data;
    }
}


/*
 * send message msg to message queue msgqid
 */

void sendm(int msgqid, int who, message *msg)
{
    // send message from who

    msg->msgtype = who;                                     // set msgtype
    msgsnd(msgqid, (struct msgbuf *)msg, sizeof(message), 0);
                                                            // no special flags
}

/*
 * receive message from message queue msgqid at memory position msg
 */

void receivem(int msgqid, int who, message *msg)
{
    // get message from sender who
    msgrcv(msgqid, (struct msgbuf *)msg, sizeof(message), who, 0);
                                                            // no special flags
}



// Main ///////////////////////////////////////////////////////////////// Main //

/* This example program implements a solution for the producer/consumer
 * problem using messages. The principle idea of this code is from
```

```
 * [Tan92, page 51].
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);


    // create message queue

    int msgqid = msgget(IPC_PRIVATE, MSG_MODE);
    if(msgqid == -1)
        error.system("msgget error");

    // use two processes for producer and consumer

    pid_t child;

    if( (child = fork() ) < 0)
        error.system("fork error");

    if(child)
    {
        // child acts as producer
        message m;
        int item;

        for(int i=1; i <= produceitems; i++)
        {
            // produce new item
            item = i;                   // produce(&item)
            cout << "producer: item " << item << " produced" << endl << flush;

            // wait for an empty to arrive
            receivem(msgqid, consumer, &m);

            // construct a message to send
            build_message(&m, item);

            // send item to consumer
            sendm(msgqid, producer, &m);
            cout << "producer: item " << item << " sent" << endl << flush;
        }
    }
    else
    {
        // parent acts as consumer

        message m;
        int item;

        // send N empties

        for(int j=0; j < N; j++)
            sendm(msgqid, producer, &m);

        for(int i=1; i <= produceitems; i++)
```

```
        {
            // get message containing item
            receivem(msgqid, producer, &m);

            // take item out of message
            extract_item(&m, &item);
            cout << "consumer: item " << item << " received" << endl << flush;

            // send back empty reply
            sendm(msgqid, consumer, &m);

            // do something with the item
            // consume(item)
            cout << "consumer: item " << item << " consumed" << endl << flush;
        }

        // destroy message queue

        if(msgctl(msgqid, IPC_RMID, NULL) == -1)
            error.system("msgctl error");

    }

    return(EXIT_SUCCESS);
}
```

A.3.5 local/namedstreampipe.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: namedstreampipe.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: function that creates a named stream pipe
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/27 15:16:31 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////

    // uncomment this if the test code should NOT be compiled
#define TEST


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ///////////////////////////////////////////// System headers //

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

```
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //

/*
 * namedstreampipe()
 *
 * this function creates a named stream pipe
 *
 * in: name: file name for stream pipe
 *     fd:   file descriptor pair returned here
 *
 * result: 0 if everything ok, -1 for error (with errno set accordingly)
 */


int
namedstreampipe(char *name, int fd[2])
{
    int len;
    struct sockaddr_un unix_addr;

    if(socketpair(AF_UNIX, SOCK_STREAM, 0, fd) < 0)
        return -1;

    unlink(name);   // remove file name if it exists
```

```
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);

    if(bind(fd[0], (struct sockaddr *) &unix_addr, len) < 0)
        return -1;

    return 0;
}

// Main ////////////////////////////////////////////////////////////////// Main //

#ifdef TEST
// this program tests if the function above works.

#include <iostream.h>
#include "../common.h"

int main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int fd[2];
    int len;

    char buffer[512];

    if( namedstreampipe("testnamed", fd) != 0)
        error.system("can,t open named stream pipe");

    if( write(fd[0], "Hello ", 6) != 6)
        error.system("write error to fd[0]");

    if( write(fd[1], "World!", 6) != 6)
        error.system("write error to fd[1]");

    if( (len=read(fd[0], buffer, 512)) != 6)
        error.system("read error on fd[0]");

    if( strncmp(buffer, "World!", 6) != 0)
        error.panic("write != read!");

    if( (len=read(fd[1], buffer, 512)) != 6)
        error.system("read error on fd[1]");

    if( strncmp(buffer, "Hello ", 6) != 0)
        error.panic("write != read!");

    unlink("testnamed");

    cout << "namedstreampipe() works" << endl;
}
#endif
```

A.3.6 local/pipe.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: easypipe.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: demonstrates the use of UNIX pipes
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 02:19:46 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */
```

```
// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int     fd[2], nbytes;
    pid_t   childpid;
    char    *string = "Hello, world!";
    char    buffer[80];

    pipe(fd);

    if((childpid = fork()) == -1)
        error.system("fork");

    if(childpid == 0)
    {
        // child: close input

        close(fd[0]);


        // write to pipe

        write(fd[1], string, strlen(string));

        return(EXIT_SUCCESS);
    }
    else
    {
        // parent: close output

        close(fd[1]);


        // read from pipe

        nbytes = read(fd[0], buffer, sizeof(buffer));
        cout << "Received: " << buffer << endl;

        return(EXIT_SUCCESS);
    }
}
```

A.3.7 local/semaphore.c

```
////////////////////////////////////////////////////////////////////////////////
```

```
// Filename: semaphore.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how UNIX semaphores work
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 00:32:54 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers ///////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>



// Local headers /////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int N = 10;              // empty buffer slots at the beginning

const int produceitems = 100;  // how much items to produce altogether

const int SEM_MODE = 0600;     // only processes of same user have access to
                               // semaphores

// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ////////////////////// Signal catching functions //
    /* NONE */
```

```
// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * do a "P" or "down" operation on the given semaphore
 */

void down(int semid)
{
    // initialize values

    struct sembuf op;
    op.sem_num =  0;         // member number (we only have number 0)
    op.sem_op  = -1;         // dec by one
    op.sem_flg =  0;         // no flags

    if( semop(semid, &op, 1) == -1)
        error.system("semop error");
}


/*
 * do a "V" or "UP" operation on the given semaphore
 */

void up(int semid)
{
    // initialize values

    struct sembuf op;
    op.sem_num = 0;         // member number (we only have number 0)
    op.sem_op  = 1;         // inc by one
    op.sem_flg = 0;         // no flags

    if( semop(semid, &op, 1) == -1)
        error.system("semop error");
}



// Main ///////////////////////////////////////////////////////////// Main //

/* This example program implements a solution for the producer/consumer
 * problem using semaphores. The principle idea of this code is from
 * [Tan92, page 43]. The code only shows the synchronization, not
 * the exchange of data, as this complicates the example. Shared memory,
 * files, or other IPC facilities might be used to actually transfer
 * the produced "items".
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);
```

```
    // semaphore ids

    int mutexid;    // controls access to critical region
    int emptyid;    // counts empty buffer slots
    int fullid;     // counts full buffer slots

    // create semaphores

    mutexid = semget(IPC_PRIVATE, 1, SEM_MODE);
    emptyid = semget(IPC_PRIVATE, 1, SEM_MODE);
    fullid  = semget(IPC_PRIVATE, 1, SEM_MODE);

    if( (mutexid == -1) || (emptyid == -1) || (fullid == -1))
        error.system("semget error");

    // initialise semaphores:
    // mutex = 1
    // empty = N
    // full = 0

#ifdef linux
    semun initial_value;
#else
    union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
    } initial_value;
#endif

    // mutexid

    initial_value.val = 1;
    if( semctl(mutexid, 0, SETVAL, initial_value) == -1)
        error.system("semctl error");


    // emptyid

    initial_value.val = N;
    if( semctl(emptyid, 0, SETVAL, initial_value) == -1)
        error.system("semctl error");

    // fullid

    initial_value.val = 0;
    if( semctl(fullid, 0, SETVAL, initial_value) == -1)
        error.system("semctl error");


    // use two processes for producer and consumer

    pid_t child;

    if( (child = fork() ) < 0)
        error.system("fork error");

    if(child)
```

```
{
    // child acts as producer

    for(int i=1; i <= produceitems; i++)
    {
        // object item

        // produce new item
        // produce(&item)
        cout << "producer: item " << i << " produced" << endl << flush;

        // decrement empty count
        down(emptyid);

        //  enter critical region
        down(mutexid);

        // put item into buffer
        // enter(item)
        cout << "producer: item " << i << " put into buffer"
             << endl << flush;

        // leave critical region
        up(mutexid);

        // increment count of full slots
        up(fullid);
    }
}
else
{
    // parent acts as consumer

    // expect produceitems to be generated

    for(int i=1; i <= produceitems; i++)
    {
        // decrement full count
        down(fullid);

        //  enter critical region
        down(mutexid);

        // remove item from buffer
        cout << "consumer: item " << i << " removed from buffer"
             << endl << flush;

        // leave critical region
        up(mutexid);

        // increment count of empty slots
        up(emptyid);

        // do something with the item
        // consume(item)
        cout << "consumer: item " << i << " consumed" << endl << flush;
    }
```

```
        // destroy semaphores after last access to semaphores

#ifdef linux
        union semun nop;
#else
        int nop=0;
#endif

        if(semctl(mutexid, 0, IPC_RMID, nop) == -1)
            error.system("semctl error");

        if(semctl(fullid, 0, IPC_RMID, nop) == -1)
            error.system("semctl error");

        if(semctl(emptyid, 0, IPC_RMID, nop) == -1)
            error.system("semctl error");
    }

    return(EXIT_SUCCESS);
}
```

A.3.8 local/shared.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: shared.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how UNIX shared memory works
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  -------------------------------------------------
// 96/02/06 00:28:57 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ////////////////////////////// Feature test switches //
    /* NONE */



// System headers //////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
#include <unistd.h>


// Local headers ////////////////////////////////////////////// Local headers //

#include "../common.h"
```

```
// Macros ///////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ///////////////////////////////////// File scope objects //

const int SHM_SIZE = 1000;
const int SHM_MODE = (SHM_R | SHM_W);       // user read/write

                                // two equal long strings to display
const char *STRING1 = "Hello, World!        ";
const char *STRING2 = "Hello, changed World!";



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////// Main //

/* This program demonstrates how two different processes can access
 * shared memory. They do not synchronize their access. This is normally
 * done with semaphores.
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int     shmid;
    char    *shmptr;

    // set up shared memory

    if( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        error.system("shmget error");

    if( (shmptr = (char *)shmat(shmid, 0, 0)) == (void *) -1)
```

```
        error.system("shmat error");


    // initialize memory that we use for communication

    strcpy(shmptr, STRING1);

    pid_t child;

    if( (child = fork() ) < 0)
        error.system("fork error");

    if(child == 0)
    {
        /* child */

        // wait a little bit
        sleep(3);

        // change memory a bit

        strcpy(shmptr, STRING2);

        sleep(2);

        // and change back to original string

        strcpy(shmptr, STRING1);
    }
    else
    {
        /* parent */

        // display memory location for 10 seconds

        time_t t=time(NULL);        // get time

        while((time(NULL) - t) < 10)
        {
            cout << shmptr << "\r" << flush;
        }

        cout << endl;

        // destry shared memory

        if (shmdt(shmptr) == -1)
            error.system("shmdt error");

        if (shmctl(shmid, IPC_RMID, 0) == -1)
            error.system("shmctl error");
    }



    return(EXIT_SUCCESS);
}
```

A.3.9 local/signal.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: signal.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: demonstrates the use of signals
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/19 01:11:12 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <signal.h>
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //

static void signalhandler(int signo)
{
    cout << "Signal " << signo << " recognized!" << endl << flush;
    return;
}
```

```
// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ////////////////////////////////////////////////////////////// Main //

/* This example program shows how to protect the interrupt signal
 * to be delivered in a critical region of code
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    // signal masks
    sigset_t newmask, oldmask, zeromask, testmask;
    bool interrupted = false;

    // set up signal handler

    struct sigaction handle_sig;
    handle_sig.sa_handler = &signalhandler; // function to call
    sigemptyset(&handle_sig.sa_mask);       // no need to block other signals
    handle_sig.sa_flags   = 0;              // no special flags.

    // set new handler into action
    if(sigaction(SIGINT, &handle_sig, NULL) != 0)
            error.system("sigaction error: can't install signal handler");

    sigemptyset(&zeromask);                 // clear all signals in signal mask

    sigemptyset(&newmask);                  // generate signal mask with SIGINT
    sigaddset(&newmask, SIGINT);


    // ...

    // the program is doing some non-critical things
    cout << "The program can be interrupted here with CTRL-C" << endl;
    cout << "(sleeping for 5 seconds)" << endl;
    sleep(5);

    // ...

    // now the program wants to make sure that it is not interrupted
    // via the SIGINT signal: the SIGINT signal is blocked and the
    // current signal mask saved

    if(sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        error.system("can't block signal");
```

```
    cout << "Entering critical section... CTRL-C is not delivered..." << endl;
    // now the program can do all the critical work...
    sleep(5);
    // ...

    // We can test for a pending signal like this:
    if(sigpending(&testmask) <0)
        error.system("sigpending error");

    if(sigismember(&testmask, SIGINT))
    {
        cout << "You have pressed CTRL-C in the critical section"
             << " without the signal beeing delivered to the process yet"
             << endl;
        interrupted = true;
    }

    // The program might now want to wait for any signal

    // sigsuspend() is used to atomically swap out the signal mask,
    // while waiting for a signal.

    cout << "Now all signals are delivered." << endl;

    if(interrupted == false)
    {
        cout << "Please generate a signal for this process (e.g. by"
             << " pressing CTRL-C) to continue" << endl;
    }

    if(sigsuspend(&zeromask) != -1)
        error.system("sigsuspend error");


    // now reset signal mask which unblocks SIGINT

    if(sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        error.system("SIG_SETMASK error");

    cout << "Now CTRL-C is recognized again. Sleeping for 5 seconds." << endl;

    sleep(5);

    // ...

    return(EXIT_SUCCESS);
}
```

A.4 Distributed IPC

The `distSOCKET` directory (see Table 33) contains the socket version of the programs described/referred to in Chapter 6 and Chapter 8, while the directory `distTLI` contains the corresponding TLI example programs (see Table 34).

File Name	**Purpose**	**Page**
`Makefile`	Makefile for this directory	190
`udpclient.c`	show how to use UDP, client, socket version, needs udpserver(.c) as UDP server	191
`udpserver.c`	show how to use UDP, server, socket version, needs udpclient(.c) as UDP client	194
`tcpclient.c`	show how to use TCP, client, socket version, needs tcpserver(.c) as TCP server	197
`tcpserver.c`	show how to use TCP, server, socket version, needs tcpclient(.c) as TCP client	200
`safesend.c`	implements "safe" sendto()/recvfrom() functions for Chapter 8	204

Table 33: Purpose of Files in the `distSOCKET` Directory

File Name	**Purpose**	**Page**
`Makefile`	Makefile for this directory	216
`udpclient.c`	show how to use UDP, client, TLI version, needs udpserver(.c) as UDP server	217
`udpserver.c`	show how to use UDP, server, TLI version, needs udpclient(.c) as UDP client	220
`tcpclient.c`	show how to use TCP, client, TLI version, needs tcpserver(.c) as TCP server	224
`tcpserver.c`	show how to use TCP, server, TLI version, needs tcpclient(.c) as TCP client	227
`acceptcall.c`	help function for tcpserver.c: accept an incoming connection request	231

Table 34: Purpose of Files in the `distTLI` Directory

A.4.1 distSOCKET/Makefile

```
include ../Makefile.def

ALL = udpserver udpclient tcpserver tcpclient safeudpclient safeudpserver

all: $(COMMON) $(ALL)

safeudpclient: safeudpclient.o safesend.o
	$(LINK.c) $< safesend.o $(LOADLIBES) $(LDLIBS) -o $@
	$(STRIP) $@

safeudpserver: safeudpserver.o safesend.o
	$(LINK.c) $< safesend.o $(LOADLIBES) $(LDLIBS) -o $@
	$(STRIP) $@


include ../Makefile.def2
```

A.4.2 distSOCKET/udpclient.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: udpclient.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use UDP (this file: UDP client) (socket version)
//          needs udpserver(.c) as UDP server
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/20 08:57:43 muellerg: created
// 96/02/21 05:39:23 muellerg: documentation cleanup
// 96/03/19 04:41:19 muellerg: updated, functionality added
//
// possible improvements:
//
// more error checking
// should accept also ascii names for computer names
// port number check
// use time-out if no reply from server
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>


// Local headers ////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * send data to udp server and get result
 *
 * in: data:       data to send (up to MAXLINE bytes)
 *     sockfd:     fd to use
 *     pserv_addr: ptr to appropriate sockaddr_XX structure
 *     servlen:    actual sizeof(*pserv_addr)
 */

void do_it(char *data, int sockfd, struct sockaddr *pserv_addr, int servlen)
{
    int n;
    char buffer[MAXLINE];

    if (sendto(sockfd, data, strlen(data)+1, 0, pserv_addr, servlen) !=
                                                    (int)(strlen(data)+1) )
        error.system("sendto error");
```

```
    // get result

    n = recvfrom(sockfd, buffer, MAXLINE, 0, (struct sockaddr *) 0, (int *) 0);
    if (n < 0)
        error.system("recvfrom error");

    if(strcmp(data,buffer) != 0)
    {
        cout << "UDP client: returned result is not equal to sent string!"
             << endl;
    }
    else
        cout << "UDP client: returned string is equal to sent string" << endl;
}



// Main ///////////////////////////////////////////////////////////////// Main //

/*
 * Example of client using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 *   argv[3]: string to send to server
 */


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=4)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port string" << endl;
        exit(EXIT_FAILURE);
    }


    int sockfd;
    struct sockaddr_in  cli_addr, serv_addr;
    int portnumber=-1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }
```

```
    // check length of data argument
    if( strlen(argv[3]) > (MAXLINE-1) )
    {
        error.warning("argument 3: string too long");
        exit(EXIT_FAILURE);
    }

    // Fill in the structure "serv_addr" with the address of the
    // server that we want to send to.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // server addr
    serv_addr.sin_port        = htons(portnumber);


    // create a UDP socket (an Internet datagram socket)

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        error.system("client: can't create datagram socket");


    // bind any local address for us

    bzero((char *) &cli_addr, sizeof(cli_addr));    /* zero out */
    cli_addr.sin_family      = AF_INET;
    cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    cli_addr.sin_port        = htons(0);

    if (bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
        error.system("client: can't bind local address");


    do_it(argv[3], sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));


    close(sockfd);

    return(EXIT_SUCCESS);
}
```

A.4.3 distSOCKET/udpserver.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: udpserver.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use UDP (this file: udp server) (socket version)
//          uses udpclient(.c) as UDP client
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ----------------------------------------------
// 96/02/20 08:54:53 muellerg: created
// 96/02/21 05:34:44 muellerg: documentation cleanup
// 96/03/19 04:42:44 muellerg: updated, functionality added
```

```
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif

// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * read and return all incoming data upto MAXLINE bytes (a echo server)
 *
 * in: sockfd   : fd to use
 *     pcli_addr: ptr to appropriate sockaddr_XX structure
```

```
 *     maxclilen: sizeof(*pcli_addr)
 */

void do_it(int sockfd, struct sockaddr *pcli_addr, int maxclilen)
{
    int             n, clilen;

    char *mesg;
    mesg = new char[MAXLINE];

    while(true)
    {
        clilen = maxclilen;
        n = recvfrom(sockfd, mesg, MAXLINE, 0, pcli_addr, &clilen);
        if (n < 0)
            error.system("recvfrom error");

        cout << "UDP server: got string \"" << mesg << "\", sending it back"
             << endl;

        // send received data back

        if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)
            error.system("sendto error");
    }
}



// Main //////////////////////////////////////////////////////////////// Main //

/*
 * Example of server using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    int sockfd;
    struct sockaddr_in  serv_addr, cli_addr;
    int portnumber=-1;


    // get port number
```

```
    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // create a UDP socket (an Internet datagram socket)

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        error.system("server: can't create datagram socket");


    // bind local address so that the client can send to us.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error.system("server: can't bind local address");

    // do something...

    do_it(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));

    // NOTREACHED

    return(EXIT_SUCCESS);
}
```

A.4.4 distSOCKET/tcpclient.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: tcpclient.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use TCP (this file: TCP client) (socket version)
//          needs tcpserver(.c) as TCP server
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/20 09:36:18 muellerg: created
// 96/02/21 05:40:07 muellerg: documentation cleanup
// 96/03/19 05:22:00 muellerg: functionality added
//
// possible improvements:
//
// more error checking
// should accept also ascii names for computer names
// port number check
///////////////////////////////////////////////////////////////////////////////
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * send data to tcp server and get result
 *
 * in: data:    data to send
```

```
 *     sockfd:  fd to use
 */

void do_it(char *data, int sockfd)
{
    int n;
    char *buffer = new char[strlen(data)+1];

    if (writen(sockfd, data, strlen(data)+1) != (int)(strlen(data)+1) )
        error.system("TCP client: write error");

    // force TCP to send data (it does not if \n is not sent)

    if (writen(sockfd, "\n", 1) != 1 )
        error.system("TCP client: write error");

    // get result

    n = readn(sockfd, buffer, strlen(data)+1);
    if (n < 0)
        error.system("TCP client: read error");

    if(strcmp(data,buffer) != 0)
    {
        cout << "TCP client: returned result is not equal to sent string!"
             << endl;
    }
    else
        cout << "TCP client: returned string is equal to sent string" << endl;

    delete[] buffer;
}




// Main /////////////////////////////////////////////////////////////// Main //


/*
 * Example of client using the TCP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 *   argv[3]: string to sent to server
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=4)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port string" << endl;
        exit(EXIT_FAILURE);
```

```
    }

    int sockfd;
    struct sockaddr_in  serv_addr;
    int portnumber=-1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }

    // Fill in the structure "serv_addr" with the address of the
    // server that we want to connect with.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // addr of host
    serv_addr.sin_port        = htons(portnumber);


    // create a TCP socket (an Internet stream socket).

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error.system("client: can't create stream socket");


    // connect to the server

    if (connect(sockfd, (struct sockaddr *) &serv_addr,
                        sizeof(serv_addr)) < 0)
        error.system("client: can't connect to server");

    do_it(argv[3], sockfd);

    close(sockfd);

    return(EXIT_SUCCESS);
}
```

A.4.5 distSOCKET/tcpserver.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: tcpserver.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use TCP (this file: TCP server) (socket version)
//          uses tcpclient(.c) as TCP client
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
```

```
// 96/02/20 09:40:57 muellerg: created
// 96/02/21 05:31:36 muellerg: documentation cleanup
// 96/03/19 05:13:20 muellerg: functionality added
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>
#include <sys/wait.h>



// Local headers //////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////////// File scope objects //

const int BACKLOG = 5;      // maximum number of pending connections



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */
```

```
// Functions and class implementation /// Functions and class implementation //

/*
 * read and return all incoming data (a echo server)
 */

void do_it(int sockfd)
{
    int     n;
    char    buffer[MAXLINE];

    cout << "TCP server: receiving data:" << endl;

    while(true)
    {
        n = readline(sockfd, buffer, MAXLINE);
        if(n)
        {
            if (n < 0)
                error.system("TCP server: read error");
            else
            {
                // output buffer
                cout << buffer << endl;

                // write result back to sender
                if (writen(sockfd, buffer, n) != n)
                    error.system("TCP server: write error");
            }
        }
        else
            return;     // connection closed
    }
}


// Main ///////////////////////////////////////////////////////////////// Main //




/*
 * Example of a concurrent server using the TCP protocol
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
```

```
        exit(EXIT_FAILURE);
    }

    // establish a function that changes the signalhandler
    // to report child death so that they can be catched and
    // zombie processes are avoided

#ifndef hpux
    catch_children();
#endif


    int sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in  cli_addr, serv_addr;
    int portnumber=-1;


    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // create a TCP socket (an Internet stream socket)

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error.system("server: can't create stream socket");


    // bind local address so that the client can send to us

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error.system("server: can't bind local address");

    // listen for connection on socket with a maximum length of
    // pending connections of 5 (normal limited to 5)

    listen(sockfd, BACKLOG);


    while(true)
    {

        // wait for a connection from a client process.
        // This server is an example of a concurrent server.

        clilen = sizeof(cli_addr);
```

```
        // check for zombies

        waitpid(WAIT_ANY, NULL, WNOHANG);


        // accept connection

        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

        if (newsockfd < 0)
            error.system("server: accept error");


        // spawn child process to handle request

        if ( (childpid = fork()) < 0)
            error.system("server: fork error");
        else
            if (childpid == 0)
            {   // child process

                close(sockfd);          // close original socket

                do_it(newsockfd);       // process the request

                return(EXIT_SUCCESS);   // and exit..
            }

        close(newsockfd);               // parent process: close socket
                                        // to be able to reuse it for the
                                        // next connection
    }
}
```

A.4.6 distSOCKET/safesend.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: safesend.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: implements "safe" sendto()/recvfrom() functions
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  -----------------------------------------------
// 96/03/19 20:04:26 muellerg: created
//
// possible improvements:
//
// use better timeout algorithm (e.g. TCPs). Not done to keep code simpler
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */
```

```
// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

struct recinfo;                    // defined later

const int MAXKEEPSTATE = 1000;     // remember the last MAXKEEPSTATE sequence
                                   // numbers sent to the process

const int MAXKEEP = 100;           // maximum number of "false" reads

static recinfo *data = NULL;       // information from "false" read
static int actnumber = 0;          // number of allocated slots
static int firstdata = 0;          // number of first data slot to look in
static int nextdata = 0;           // number of next slot to use

static unsigned long sequencenumber;// is initialized in initialize() to a
                                   // random number. Sequence numbers are
                                   // used for all outgoing messages

static int MAXRETRY = 10;          // how often to retry a message
static int TIMEOUT = 10;           // timeout in seconds

static int MAXBUFFER = 16384;      // maximum size of messages, some bytes
                                   // needed for own message header

static bool initialized = false;   // initialize data only once

static unsigned long crc_32_tab[256];// CRC-32 table
```

```
// External variables, functions, and classes //////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //

enum messagetype {SEND, ACK};       // type of message


// format of all messages sent and received by the safe functions

struct newmessage
{
    unsigned long sequencenumber;// sequence number for this message
    messagetype type;            // type of message (only for safety)
    unsigned long CRC;           // checksum

    char data[0];                // actual data starts here
};


// message for maintaining information about wrong "reads" in safesendto()

struct recinfo
{
    bool valid;                 // is entry allocated?
    int fd;                     // file descriptor
    char *msg;                  // actual data
    int len;                    // len of message

    int clilen;                 // length of from
    struct sockaddr from;       // origin of send
};


// information about last sequence numbers

struct seqinfo
{
    unsigned long seqno;        // sequence number
    struct sockaddr sock;       // for this source
    int fd;                     // using this file descriptor
};


// Functions and class implementation /// Functions and class implementation //


////////////////////////////////////////////////////////////////////////////////
// CRC functions
////////////////////////////////////////////////////////////////////////////////
```

```
/*
 * Run a set of bytes through the crc shift register.  If s is a NULL
 * pointer, then initialize the crc shift register contents instead.
 * Return the current crc in either case.
 *
 * s: pointer to bytes to pump through
 * n: number of bytes in s[]
 */

static unsigned long
updcrc(unsigned char *s, unsigned int n)
{
    register unsigned long c;         // temporary variable

    static unsigned long crc = (unsigned long)0xffffffffL;
                                      // shift register contents

    if (s == NULL)
    {
        c = 0xffffffffL;
    }
    else
    {
        c = crc;
        while (n--)
            c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^ (c >> 8);
    }
    crc = c;
    return c ^ 0xffffffffL;       // (instead of ~c for 64-bit machines)
}


/*
 * Code to compute the CRC-32 table. Borrowed from
 * gzip-1.0.3/makecrc.c.
 */

static void
makecrc(void)
{
/* Not copyrighted 1990 Mark Adler      */

  unsigned long c;      /* crc shift register */
  unsigned long e;      /* polynomial exclusive-or pattern */
  int i;                /* counter for all possible eight bit values */
  int k;                /* byte being shifted into crc apparatus */

  /* terms of polynomial defining this crc (except x^32): */
  static int p[] = {0,1,2,4,5,7,8,10,11,12,16,22,23,26};

  /* Make exclusive-or pattern from polynomial */
  e = 0;
  for (i = 0; i < (int)(sizeof(p)/sizeof(int)); i++)
    e |= 1L << (31 - p[i]);

  crc_32_tab[0] = 0;

  for (i = 1; i < 256; i++)
```

```
  {
    c = 0;
    for (k = i | 256; k != 1; k >>= 1)
    {
      c = c & 1 ? (c >> 1) ^ e : c >> 1;
      if (k & 1)
        c ^= e;
    }
    crc_32_tab[i] = c;
  }
}



/*
 * fill in propper CRC value
 */

void fillin_CRC(newmessage *mes, int len)
{
    mes->CRC = 0;

    updcrc(NULL, 0);
    mes->CRC = updcrc((unsigned char *)mes, len);
}


/*
 * chek if message is valid (true = ok, false = error)
 */

bool check_CRC(newmessage *mes, int len)
{
    // set crc field to 0
    unsigned long crc32_old = mes->CRC;
    mes->CRC = 0;

    updcrc(NULL, 0);
    unsigned long crc32_new = updcrc((unsigned char *)mes, len);
    mes->CRC = crc32_old;

    if(crc32_old == crc32_new)  // better readable than
        return true;                //  return crc32_old == crc32_new;
    else
        return false;
}



////////////////////////////////////////////////////////////////////////////////
// sequence number handling
////////////////////////////////////////////////////////////////////////////////

/*
 * maintain a list of last used sequence numbers. If a message arrives
 * with a sequence number that we already got, we can then safely discard
 * that message. Therefore duplicated messages are no rpoblem any more.
 *
```

```
 * As we can not keep track of all sequence numbers we got, only a list
 * of the recent ones can be maintained. This function checks if a sequence
 * number is already "registered" (then the function returns with true)
 * or otherwise registers the sequencenumber and returns false.
 *
 * It is important to store where a sequence number comes from, as
 * different sources could have the same sequence numbers (even if the
 * sequence number is initialized randomly).
 *
 */


bool check_sequencenumber(unsigned long seqno, int fd, struct sockaddr *from)
{
    // circular array of appropriate entries...

    static int position = 0;

    static struct seqinfo *seq = NULL;

    if(!seq)
    {
        // allocate fixed number of last connections

        seq = new seqinfo[MAXKEEPSTATE];
        if(!seq)
            error.panic("out of memory error!");

        // initialize memory
        memset(seq, 0, MAXKEEPSTATE * sizeof(seqinfo));
    }


    // check if we have this sequence number already

    for(int i = 0; i < MAXKEEPSTATE; i++)
    {
        if(seq[i].seqno == seqno)
        {
            if(memcmp(&(seq[i].sock.sa_family), from, 8) == 0)
            {                               // 8 bytes is IP secific
                if(seq[i].fd == fd)
                    return true;        // entries are identical
            }
        }
    }


    // if we get here, no entry was found, so store entry

    seq[position].seqno = seqno;
    memcpy(&(seq[position].sock.sa_family), from, 8);
    seq[position].fd = fd;

    position = (position + 1) % MAXKEEPSTATE;
    return false;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// help functions
//////////////////////////////////////////////////////////////////////////////

/*
 * initialize some variables for safexxxx()
 */

static void initialize(void)
{
    data = new recinfo[MAXKEEP];
    if(!data)
        error.panic("out of memory error!");

    // mark all slots as invalid

    for(int i = 0; i < MAXKEEP; i++)
        data[i].valid = false;


    // start with random sequence number

    srandom(time(NULL));
    sequencenumber = random()+getpid();

    makecrc();      // generate CRC - 32 table

    initialized = true;
}



/*
 * send reply message. Returns result from sendto()
 */

static int send_reply(int fd, unsigned long seqn, int flags,
                            struct sockaddr *to, int tolen)
{
    // get memory for reply message
    char *bufferreply = new char[sizeof(newmessage)];
    if(!bufferreply)
        error.panic("out of memory error!");

    // copy all needed information
    newmessage *reply = (newmessage *)bufferreply;

    reply->type = ACK;
    reply->sequencenumber = seqn;

    fillin_CRC(reply, sizeof(newmessage));

    int sendresult = sendto(fd, bufferreply, sizeof(newmessage), flags,
                                                                 to, tolen);

    delete[] bufferreply;
```

```
    return sendresult;
}



////////////////////////////////////////////////////////////////////////////////
// the safe functions (the aim of the game)
////////////////////////////////////////////////////////////////////////////////

/*
 * implementation of a "reliable" UDP sendto(). Same parameters as to
 * the normal sendto() system call.
 * Return: same as sendto, but errno = 0 and return = -1 indicate
 *         time-out (message was not acknowledged in time)
 */

int safesendto(int s, const void *msg, int len, unsigned int flags,
                                const struct sockaddr *to, int tolen)
{
    // initialize everything

    if(!initialized)
        initialize();

    if(len > (int)(MAXBUFFER - sizeof(newmessage)))
        error.panic("safesendto(): message len to long!");

    // get memory for message
    char *buffer= new char[len+sizeof(newmessage)];
    if(!buffer)
        error.panic("out of memory error!");


    // copy all needed information
    newmessage *newmsg = (newmessage *)buffer;

    newmsg->type = SEND;
    newmsg->sequencenumber = sequencenumber++;
    memcpy(newmsg->data, msg, len);

    fillin_CRC(newmsg, len+sizeof(newmessage));

    for(int times_to_do=0; times_to_do < MAXRETRY; times_to_do++)
    {
        int sendresult = sendto(s, newmsg, len+sizeof(newmessage),
                                                     flags, to, tolen);
        fd_set set;
        FD_ZERO(&set);
        FD_SET(s, &set);

        struct timeval t;
        t.tv_sec  = TIMEOUT;
        t.tv_usec = 0;
                           // maxnumber, read, write, except, time
        int available = select(s+1,      &set, NULL,  NULL,   &t);
        int n;

        if(available > 0)
```

```
{
    // read result
    char *recbuffer = new char[MAXBUFFER];
    struct sockaddr pcli_addr;
    int clilen;

    n = recvfrom(s, recbuffer, MAXBUFFER, 0, &pcli_addr, &clilen);
    if (n < 0)
    {
        // recvfrom error, ignore
        // error.system("recvfrom error");

        while((available > 0) && (n < (int)sizeof(newmessage)))
        {
            FD_ZERO(&set);
            FD_SET(s, &set);

            available = select(s+1, &set, NULL,  NULL, NULL);

            if(available > 0)
                n = recvfrom(s, recbuffer, MAXBUFFER, 0, &pcli_addr,
                            &clilen);
        }
    }

    if(n < 0)
    {
        // ignore error
    }
    else
    {
        // recvfrom ok

        // check if this is the requested ack message, othervise save
        // message

        newmessage *rcv=(newmessage *)recbuffer;

        if(check_CRC(rcv, n))
        {
            // CRC is ok

            if(rcv->type == ACK)
            {
                if(rcv->sequencenumber == (sequencenumber-1) )
                {
                    // we got correct ACK, clean up
                    delete[] recbuffer;
                    delete[] buffer;

                    // and return original result

                    return sendresult-sizeof(newmessage);
                }
                else
                {
                    // wrong sequence number
                }
```

```
                    }

                    // we could not deal with the message, so save it for
                    // further reads

                    if(rcv->type != ACK)    // don't buffer ACKs
                    {
                        data[nextdata].fd     = s;
                        data[nextdata].msg    = recbuffer;
                        data[nextdata].len    = n;
                        data[nextdata].clilen = clilen;
                        memcpy(&(data[nextdata].from.sa_family), &pcli_addr,
                                            sizeof(struct sockaddr));
                        data[nextdata].valid  = true;

                        nextdata = (nextdata + 1) % MAXKEEP;
                        actnumber++;
                        if(actnumber > MAXKEEP)
                            error.panic("safesendto(): buffer overflow");
                    }
                    else
                    {
                        // discarding message (unwanted ACK error)
                        delete[] recbuffer;
                    }
                }
                else
                {
                    // discarding message (CRC error)
                    delete[] recbuffer;
                }
            } // recvfrom ok
        } // available
    } // for


    // unsuccessful, clean up
    delete[] buffer;


    // message could not be sent, return -1 for error

    // special errno value for timeout

    errno = 0;
    return -1;
}



/*
 * implementation of a "reliable" UDP recvfrom(). Same parameters and
 * result as the normal recvfrom system call
 */


int saferecvfrom(int s, void *buf, int len, unsigned int flags,
                            struct sockaddr *from, int *fromlen)
```

```
{
    // initialize everything

    if(!initialized)
        initialize();

    if(len > (int)(MAXBUFFER - sizeof(newmessage)))
        error.panic("saferecvfrom(): message len to long!");

    sockaddr tfrom;                     // if from = NULL use this one
    int tfromlen;                       // if tfrom = NULL use this one

    if(!from)
        from = &tfrom;

    if(!fromlen)
        fromlen = &tfromlen;


    char *buffer = NULL;                // pointer to message
    int n=0;                            // len of read message

    bool usealreadyread;                // set to true if we use message
                                        // that is already read

    newmessage *get;                    // pointer to buffer, but different
                                        // type

    bool done = false;                  // have we got an appropriate message?

    while(!done)
    {
        usealreadyread = false;
        tfromlen=sizeof(sockaddr);

        // check if we have some saved messages for this file descriptor
        if(actnumber)
        {
            int start = firstdata;

            for(int j=0; j < actnumber; j++)
            {
                if(data[start].valid == true)
                {
                    if(data[start].fd == s)
                    {
                        // wow, message found! return this instead of getting
                        // a new one.

                        buffer = data[start].msg;
                        n = data[start].len;
                        *fromlen = data[start].clilen;
                        memcpy(from, &(data[start].from.sa_family),
                            sizeof(struct sockaddr));

                        data[start].valid = false;

                        usealreadyread = true;
```

```
                firstdata++;
                actnumber--;
            }
        }
        start++;
    }
}


if(!usealreadyread)
{
    // get memory for message
    buffer= new char[MAXBUFFER];
    if(!buffer)
        error.panic("out of memory error!");

    // get message

    bool message_ok=false;
    while(!message_ok)
    {
        n = recvfrom(s, buffer, MAXBUFFER, 0, from, fromlen);
        if(n >= 0)
        {
            message_ok=true;
        }
        else
        {
            // error.system("recvfrom error");
            // ignore errors
        }
    }
}

get=(newmessage *)buffer;

// handle message, copy stuff etc.
// memorize that we have acknowledged this message....

if(check_CRC(get, n))
{
    if(get->type == ACK)
    {
        // ignore message (got unwanted ACK message)
    }
    else
    {
        // check if we have already got this sequence number from
        // that particular address

        if(check_sequencenumber(get->sequencenumber, s, from))
        {
            // This sequence number was already handled. The Reply
            // message was probably lost or got corrupted. Therefore
            // sent reply again, but DO NOT return this message as
            // new message

            // send reply again...
```

```
                send_reply(s, get->sequencenumber, flags, from, *fromlen);
            }
            else
            {
                // we got a new message that we can return
                // copy message...

                memcpy(buf, get->data, n-sizeof(newmessage));
                done = true;

                // send reply
                send_reply(s, get->sequencenumber, flags, from, *fromlen);
            }
        } // else ACK
    } // CRC check ok
    else
    {
        // ignore message that has no correct CRC
    }

    // clean up memory
    delete[] buffer;

} // while(!done)

return n-sizeof(newmessage);
}



// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.4.7 distTLI/Makefile

```
include ../Makefile.def

ALL = tcpserver  tcpclient  udpserver  udpclient \
      ptcpserver ptcpclient pudpserver pudpclient \
      potcpserver potcpclient

all: $(COMMON) $(ALL)

tcpserver: tcpserver.o acceptcall.o
    $(LINK.c) $< tcpserver.o acceptcall.o strecho.o $(LOADLIBES) $(LDLIBS) -o $@
    $(STRIP) $@

ptcpserver: ptcpserver.o acceptcall.o
    $(LINK.c) $< ptcpserver.o acceptcall.o $(LOADLIBES) $(LDLIBS) -o $@
    $(STRIP) $@

potcpserver: potcpserver.o acceptcall.o
    $(LINK.c) $< potcpserver.o acceptcall.o $(LOADLIBES) $(LDLIBS) -o $@
    $(STRIP) $@

# generate performance data
```

```
dataget:
	#
	# local TCP / UDP
	#
	# measure (local) UDP throughput and ops/s
	pudpserver 6541 &
	sleep 3
	pudpclient 127.0.0.1 6541
	#
	# measure (local) TCP throughput
	ptcpserver 6543 &
	sleep 3
	ptcpclient 127.0.0.1 6543
	#
	# measure overhead of establishing a (local) TCP connection
	potcpserver 6542 &
	sleep 3
	potcpclient 127.0.0.1 6542

include ../Makefile.def2
```

A.4.8 distTLI/udpclient.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: udpclient.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use UDP (this file: UDP client) (TLI version)
//          needs udpserver(.c) as UDP server
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/26 14:06:12 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */


// System headers /////////////////////////////////////////// System headers //

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>

#include <tiuser.h>
```

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

#ifdef sun
// O_RDWR
#include <sys/fcntl.h>
#endif


// Local headers ////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * Read the contents of the FILE *fp, write each line to the transport
 * endpoint (to the server process), then read a line back from
 * the transport endpoint and print it on the standard output.
 *
 * in: tfd:       tli descriptor
 *     sudataptr: unitdata for sends
 *     fp:        FILE pointer
 */

void doit(int tfd, struct t_unitdata *sudataptr, FILE *fp)
{
    int n, flags;
    char sendline[MAXLINE], recvline[MAXLINE + 1];
    struct t_unitdata *rudataptr;   // unitdata for receives
```

```
    /*
     * Allocate memory for the t_unitdata structure and the address field
     * in that structure.  This allows any size of address to be handled
     * by this function.
     */

    rudataptr = (struct t_unitdata *) t_alloc(tfd, T_UNITDATA, T_ADDR);
    if (rudataptr == NULL)
        error.system("server: t_alloc error for T_UNITDATA");

    while (fgets(sendline, MAXLINE, fp) != NULL)
    {
        n = strlen(sendline);
        sudataptr->udata.len = n;
        sudataptr->udata.buf = sendline;
        if (t_sndudata(tfd, sudataptr) < 0)
            error.system("client: t_sndudata error");


        // Now read a message from the transport endpoint and
        // write it to our standard output.

        rudataptr->opt.maxlen = 0;  // don't care about options
        rudataptr->udata.maxlen = MAXLINE;
        rudataptr->udata.buf    = recvline;

        if (t_rcvudata(tfd, rudataptr, &flags) < 0)
            error.system("client: t_rcvudata error");

        recvline[rudataptr->udata.len] = 0; // null terminate
        fputs(recvline, stdout);
    }

    if (ferror(fp))
        error.system("client: error reading file");
}


// Main ////////////////////////////////////////////////////////////////// Main //

/*
 * Example of client using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=3)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port" << endl;
        exit(EXIT_FAILURE);
```

```
    }

    int tfd;
    struct t_unitdata unitdata;
    struct sockaddr_in serv_addr;

    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }

    // Open a UDP endpoint.

    if ( (tfd = t_open(DEV_UDP, O_RDWR, (struct t_info *) 0)) < 0)
    {
        error.warning("client: can't t_open %s", DEV_UDP);
        exit(EXIT_FAILURE);
    }


    // Bind any local address for us.

    if (t_bind(tfd, (struct t_bind *) 0, (struct t_bind *) 0) < 0)
        error.system("client: t_bind error");


    // Initialize a sockaddr_in structure with the address of the
    // the server we want to send datagrams to.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // server addr
    serv_addr.sin_port        = htons(portnumber);

    // Now initialize a unitdata structure for sending to the server.

    unitdata.addr.maxlen = sizeof(serv_addr);   // server's addr
    unitdata.addr.len    = sizeof(serv_addr);
    unitdata.addr.buf    = (char *) &serv_addr;
    unitdata.opt.maxlen  = 0;                   // no options
    unitdata.opt.len     = 0;
    unitdata.opt.buf     = (char *) 0;

    doit(tfd, &unitdata, stdin);    // do it all

    t_close(tfd);
    return(EXIT_SUCCESS);
}
```

A.4.9 distTLI/udpserver.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: udpserver.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use UDP (this file: udp server) (TLI version)
//          uses udpclient(.c) as UDP client
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  -----------------------------------------------
// 96/02/26 14:17:40 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>

#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
// #include <sys/in.h>

#ifdef sun
// O_RDWR
#include <sys/fcntl.h>
#endif


// Local headers //////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros //////////////////////////////////////////////////////////// Macros //
    /* NONE */
```

```
// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * Read the contents of the TLI descriptor and write each line back to
 * the sender.
 */

void do_it(int tfd)
{
    int n, flags;
    char line[MAXLINE];
    struct t_unitdata   *udataptr;

    // Allocate memory for the t_unitdata structure and the address field
    // in that structure.  This allows any size of address to be handled
    // by this function.

    udataptr = (struct t_unitdata *) t_alloc(tfd, T_UNITDATA, T_ADDR);
    if (udataptr == NULL)
        error.system("server: t_alloc error for T_UNITDATA");

    while(true)
    {
        // Read a message from the TLI descriptor and send it back
        // to whomever sent it.

        udataptr->opt.maxlen   = 0;     // don't care about options
        udataptr->opt.len      = 0;
        udataptr->udata.maxlen = MAXLINE;
        udataptr->udata.len    = MAXLINE;
        udataptr->udata.buf    = line;
        if (t_rcvudata(tfd, udataptr, &flags) < 0)
            error.system("server: t_rcvudata() error");

        if (t_sndudata(tfd, udataptr) < 0)
            error.system("server: t_sndudata() error");
    }
}
```

```
// Main ///////////////////////////////////////////////////////////////// Main //

/*
 * Example of server using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    int tfd;
    struct sockaddr_in serv_addr;
    struct t_bind req;

    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // Open a UDP endpoint.

    if ( (tfd = t_open(DEV_UDP, O_RDWR, (struct t_info *) 0)) < 0)
    {
        error.warning("server: can't t_open %s", DEV_UDP);
        exit(EXIT_FAILURE);
    }


    // Bind our local address so that the client can send to us.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    req.addr.maxlen = sizeof(serv_addr);
    req.addr.len    = sizeof(serv_addr);
    req.addr.buf    = (char *) &serv_addr;
```

```
    req.qlen        = 5;

    if (t_bind(tfd, &req, (struct t_bind *) 0) < 0)
        error.system("server: can't t_bind local address");

    do_it(tfd);     // do it all
        /* NOTREACHED */

    return(EXIT_SUCCESS);
}
```

A.4.10 distTLI/tcpclient.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: tcpclient.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use TCP (this file: TCP client) (TLI version)
//          uses tcpserver(.c) as TCP server
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/26 13:52:19 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>

#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
// #include <sys/in.h>

#ifdef sun
// O_RDWR
#include <sys/fcntl.h>
#endif
```

```
// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * Read the contents of the FILE *fp, write each line to the
 * transport endpoint (to the server process), then read a line back from
 * the transport endpoint and print it on the standard output.
 */

void doit(FILE *fp, int tfd)
{
    int n, flags;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];

    while (fgets(sendline, MAXLINE, fp) != NULL) {
        n = strlen(sendline);
        if (t_snd(tfd, sendline, n, 0) != n)
            error.system("client: t_snd error");


        // Now read a line from the transport endpoint and write it to
        // our standard output.

        n = t_rcv(tfd, recvline, MAXLINE, &flags);
        if (n < 0)
            error.system("client: t_rcv error");
        recvline[n] = 0;    /* null terminate */
        fputs(recvline, stdout);
```

```
    }

    if (ferror(fp))
        error.system("client: error reading file");
}



// Main /////////////////////////////////////////////////////////////// Main //

/*
 * Example of client using TCP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=3)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port" << endl;
        exit(EXIT_FAILURE);
    }

    int tfd;
    struct t_call *callptr;
    struct sockaddr_in serv_addr;

    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // Create a TCP transport endpoint and bind it.

    if ( (tfd = t_open(DEV_TCP, O_RDWR, 0)) < 0)
    {
        error.warning("client: can't t_open %s", DEV_TCP);
        exit(EXIT_FAILURE);
    }

    if (t_bind(tfd, (struct t_bind *) 0, (struct t_bind *) 0) < 0)
        error.system("client: t_bind error");
```

```
    // Fill in the structure "serv_addr" with the address of the
    // server that we want to connect with.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_addr.sin_port        = htons(portnumber);


    // Allocate a t_call structure, and initialize it.
    // Let t_alloc() initialize the addr structure of the t_call structure.

    if ( (callptr = (struct t_call *) t_alloc(tfd, T_CALL, T_ADDR)) == NULL)
        error.system("client: t_alloc error");

    callptr->addr.maxlen = sizeof(serv_addr);
    callptr->addr.len    = sizeof(serv_addr);
    callptr->addr.buf    = (char *) &serv_addr;
    callptr->opt.len     = 0;       // no options
    callptr->udata.len   = 0;       // no user data with connect


    // Connect to the server.

    if (t_connect(tfd, callptr, (struct t_call *) 0) < 0)
        error.system("client: can't t_connect to server");

    doit(stdin, tfd);   // do it all

    close(tfd);
    exit(EXIT_SUCCESS);
}
```

A.4.11 distTLI/tcpserver.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: tcpserver.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: show how to use TCP (this file: TCP server) (TLI version)
//          uses tcpclient(.c) as TCP client
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/26 13:52:19 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */
```

```
// System headers ////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>

#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
// #include <sys/in.h>

#ifdef sun
// O_RDWR
#include <sys/fcntl.h>
#endif
#include <sys/wait.h>


// Local headers //////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //

int accept_call(int listenfd, struct t_call *callptr, char *name, int rwflag);



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */
```

```
// Functions and class implementation /// Functions and class implementation //


/*
 * Read a stream socket one line at a time, and write each line back
 * to the sender.
 *
 * Return when the connection is terminated.
 */

void
do_it(int sockfd)
{
    int n;
    char buffer[MAXLINE];

    while(true)
    {
        n = readline(sockfd, buffer, MAXLINE);
        if (n == 0)
            return;     // connection terminated
        else
            if (n < 0)
                error.system("str_echo: readline error");

        if (writen(sockfd, buffer, n) != n)
            error.system("str_echo: writen error");
    }
}


// Main ///////////////////////////////////////////////////////////////// Main //

/*
 * Example of server using TCP protocol. This program is lightly based on
 * example programs in [Stev90]
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    // establish a function that changes the signalhandler
    // to report child death so that they can be catched and
    // zombie processes are avoided

#ifndef hpux
```

```
    catch_children();
#endif

    int tfd, newtfd, clilen, childpid;
    struct sockaddr_in cli_addr, serv_addr;
    struct t_bind req;
    struct t_call *callptr;

    int portnumber = -1;

    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // create a TCP transport endpoint.

    if ( (tfd = t_open(DEV_TCP, O_RDWR, (struct t_info *) 0)) < 0)
    {
        error.warning("server: can't t_open %s", DEV_TCP);
        exit(EXIT_FAILURE);
    }

    // Bind our local address so that the client can send to us.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    req.addr.maxlen = sizeof(serv_addr);
    req.addr.len    = sizeof(serv_addr);
    req.addr.buf    = (char *) &serv_addr;
    req.qlen        = 5;

    if (t_bind(tfd, &req, (struct t_bind *) 0) < 0)
        error.system("server: can't t_bind local address");

    // Allocate a t_call structure for t_listen() and t_accept().

    if ( (callptr = (struct t_call *) t_alloc(tfd, T_CALL, T_ADDR)) == NULL)
        error.system("server: t_alloc error for T_CALL");

    while(true)
    {
        // check for zombies

        waitpid(WAIT_ANY, NULL, WNOHANG);


        // Wait for a connection from a client process.
        // This is an example of a concurrent server.
```

```
        if (t_listen(tfd, callptr) < 0)
            error.system("server: t_listen error");

        if ( (newtfd = accept_call(tfd, callptr, DEV_TCP, 1)) < 0)
            error.system("server: accept_call error");

        if ( (childpid = fork()) < 0)
            error.system("server: fork error");

        else if (childpid == 0)
             {  /* child process */
                t_close(tfd);       // close original endpoint

                do_it(newtfd);      // process the request

               exit(EXIT_SUCCESS);
             }

        close(newtfd);      /* parent process */
    }
}
```

A.4.12 distTLI/acceptcall.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: acceptcall.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: accept an incoming connection request
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/12 02:37:12 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */


// System headers /////////////////////////////////////////// System headers //

#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>
#include <unistd.h>
#include <stdlib.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
```

```
// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * Accept an incoming connection request.
 *
 * Return the new descriptor that refers to the newly accepted connection,
 * or -1.  The only time we return -1 is if a disconnect arrives before
 * the accept is performed.
 *
 *
 * in: listenfd: the descriptor caller used for t_listen()
 *     callptr:  from t_listen(), passed to t_accept()
 *     name:     name of transport provider
 *     rwflag;   if nonzero, push read/write module
 *
 * out: new descriptor
 */


int
accept_call(int listenfd, struct t_call *callptr, char *name, int rwflag)
{
    int     newfd;
    extern int  t_errno;


    // Open the transport provider to get a new file descriptor.

    if ( (newfd = t_open(name, O_RDWR, (struct t_info *) 0)) < 0)
        error.system("t_open error");
```

```
    // Bind any local address to the new descriptor.  Since this
    // function is intended to be called by a server after a
    // connection request has arrived, any local address will suffice.

    if (t_bind(newfd, (struct t_bind *) 0, (struct t_bind *) 0) < 0)
        error.system("t_bind error");


    // Accept the connection request on the new descriptor.

    if (t_accept(listenfd, newfd, callptr) < 0)
    {
        if (t_errno == TLOOK)
        {
            // An asynchronous event has occurred.  We must have
            // received a disconnect.  Go ahead and call t_rcvdis(),
            // then close the new file descriptor that we opened
            // above.

            if (t_rcvdis(listenfd, (struct t_discon *) 0) < 0)
                error.system("t_rcvdis error");
            if (t_close(newfd) < 0)
                error.system("t_close error");
            return(-1);     // return error to caller
        }
        error.system("t_accept error");
    }


    // If the caller requests, push the streams module "tirdwr" onto
    // the new stream, so that the read(2) and write(2) system calls
    // can be used.  We first have to pop the "timod" module (the
    // default).

    if (rwflag)
    {
        if (ioctl(newfd, I_POP, (char *) 0) < 0)
            error.system("I_POP of timod failed");

        if (ioctl(newfd, I_PUSH, "tirdwr") < 0)
            error.system("I_PUSH of tirdwr failed");
    }

    return(newfd);
}


// Main ///////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.5 Performance

The `performance` directory (see Table 35) contains the programs to measure the performance of various IPC facilities for Chapter 7. The WWW version of this document contains hyper-links to the measured data and performance/timing graphs in Postscript.

File Name	**Purpose**	**Page**
`Makefile`	Makefile for this directory	235
`fifoclient.c`	measures performance of FIFOs (together with fifoserver.c)	236
`fifoserver.c`	measures performance of FIFOs (together with fifoclient.c)	238
`memory.c`	measures the throughput of memory in normal and shared memory	240
`message.c`	measures how long it takes to send/receive System V IPC messages	244
`pipes.c`	measures how fast pipes are	247
`semaphore.c`	measures how fast semaphore operations are	249
`shared.c`	measures how long it takes to add/remove a shared memory region	252
`signaloverhead.c`	measures the overhead of synchronizing tasks with signals	255
`socketlocal.c`	test the performance of local sockets via socketpair()	258
`systemcalloverhead.c`	measures the overhead of a system call	259
`tcpsclientoverhead.c`	measures the overhead in establishing a TCP connection	261
`tcpsclientthroughput.c`	measures performance of a TCP connection, uses tcpserverconcurrent(.c) as TCP server	264
`tcpsservercon-currentoverhead.c`	needed for measurements by tcpsclientoverhead.c	266
`tcpsserverconcurrent.c`	measures performance of a TCP connection, uses tcpsclientthroughput(.c) as TCP client	270
`timeroverhead.c`	measures the overhead of using the timer classes	273

File Name	Purpose	Page
`udps.h`	some common definitions for udpsserver.c and udpssclient.c	277
`udpsclient.c`	measures performance of UDP, client, needs udpsserver(.c) as UDP server	278
`udpsserver.c`	measures performance of UDP, server, needs udpsclient(.c) as UDP client	282

Table 35: Purpose of Files in the `performance` Directory

A.5.1 performance/Makefile

```
include ../Makefile.def

ALL = signaloverhead timeroverhead pipes socketlocal \
      fifoserver fifoclient shared \
      tcpsclientoverhead tcpsserverconcurrentoverhead semaphore \
      tcpsserverconcurrent tcpsclientthroughput \
      udpsserver udpsclient message \
      systemcalloverhead memory

all: $(ALL)

# generate performance data

dataget:
    # measure memory throughput
    memory >memory.log
    #
    # measure system call overhead
    systemcalloverhead >systemcalloverhead.log
    #
    # measure signal time
    signaloverhead >signaloverhead.log
    #
    # measure timer overhead
    timeroverhead >timeroverhead.log
    #
    # measure pipe throughput
    pipes
    #
    # measure throughput of local sockets
    socketlocal
    #
    # measures fifo throughput
    fifoserver &
    sleep 3
```

```
    fifoclient
    #
    # System V IPC
    #
    # measure how many sem ops per sec
    semaphore >semaphore.log
    #
    # measure how many shared memory region ops per sec
    shared >shared.log
    #
    # measure throughput and ops/s of messages
    message >message.log
    #
    # local TCP / UDP
    #
    # measure (local) UDP throughput and ops/s
    udpsserver 6543 &
    sleep 3
    udpsclient 127.0.0.1 6543
    #
    # measure (local) TCP throughput
    tcpsserverconcurrent 6542 &
    sleep 3
    tcpsclientthroughput 127.0.0.1 6542
    #
    # measure overhead of establishing a (local) TCP connection
    tcpsserverconcurrentoverhead 6541 &
    sleep 3
    tcpsclientoverhead  127.0.0.1 6541
    #
    #
    ## dist TCP / UDP
    ##
    ## measure (dist) UDP throughput and ops/s
    #udpsserver 6543 &
    #udpsclient 127.0.0.1 6543
    ##
    ## measure (dist) TCP throughput
    #tcpsserverconcurrent 6542 &
    #tcpsclientthroughput 127.0.0.1 6542
    ##
    ## measure overhead of establishing a (dist) TCP connection
    #tcpsserverconcurrentoverhead 6541 &
    #tcpsclientoverhead 127.0.0.1 6541


include ../Makefile.def2
```

A.5.2 performance/fifoclient.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: fifoclient.c
///////////////////////////////////////////////////////////////////////////////
```

```
// Purpose: measures performance of fifos (together with fifoserver.c)
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 06:42:57 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ///////////////////////////////////// File scope objects //

const char *fifo_file="example_fifo";



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions /////////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
```

```
    /* NONE */



// Main ////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int fd = open(fifo_file, O_WRONLY);
    if(fd)
    {
        // tell server that we are there
        write(fd, "GO", 2);

        // do measurements
        writedata(argv[0], "write", fd, 0); // we don't know servers pid
    }
    else
        error.system("can't open fifo");

    return(EXIT_SUCCESS);
}
```

A.5.3 performance/fifoserver.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: fifoserver.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measures performance of fifos (together with fifoclient.c)
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 06:43:04 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
```

```
#include <fcntl.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const char *fifo_file="example_fifo";

const mode_t permission =
     (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
// = 0666 = rw-rw-rw


// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);
    char buffer[10];

    umask(0);

    mkfifo(fifo_file, permission);

    // this would be equivalent:
    // mknod(fifo_file, S_IFIFO|permission, 0);
```

```
    int fd = open(fifo_file, O_RDONLY);
    if(fd)
    {
        // wait for a client

        read(fd, buffer, 2);

        // and do measurements

        readdata(argv[0], "read", fd, 0);   // we don't know clients pid
    }
    else
        error.system("can't open fifo");

    if( unlink(fifo_file) == -1)
        error.system("can't remove special fifo file");

    return(EXIT_SUCCESS);
}
```

A.5.4 performance/memory.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: memory.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure the throughput of memory in normal memory and shared
//          memory
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/17 02:33:04 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
```

```
// Macros ////////////////////////////////////////////////////////// Macros //

// #define READMEMORY 1
                        /* most compilers fool this test, as they optimize
                           the operations so that the results are not
                           meaningful. The assembler source code was
                           checked that this is not the case for the write */



// File scope objects ////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 100;         // how often to do measurements

const int SHM_SIZE = 1024*1024;        // shared and normal memory size, 1 MB

const int SHM_MODE = (SHM_R | SHM_W);  // user read/write


// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */




// Functions and class implementation /// Functions and class implementation //

/*
 * perform measurements with memory pointed to by ptr
 */

void writememory(char *ptr, char *prgname, char *comment)
{
    register long *mem;
    register long i = 0;
    register long value = 0;

    measurement mes(prgname, "write", NUMBER_REPEAT);

    int numberoftimes = SHM_SIZE/(sizeof(long) * 64);

    // make sure that memory is attached and eventually partly cached

    for(char *p=ptr; i < SHM_SIZE; i++)
        p[i]=255;

#ifdef READMEMORY
    cout << "\nWrite to memory" << endl;
```

```
#endif

    for(int n=0; n < NUMBER_REPEAT; n++)
    {
        mem = (long *)ptr;
        mes.start(-1, 0, comment);

        // a little bit of loop unrolling

        // this writes 256 bytes if sizeof(long) is 4 bytes

        for(i=0; i < numberoftimes; i++)
        {
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;

            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
            *mem++= value; *mem++= value; *mem++= value; *mem++= value;
        }

        mes.end();
    }

    mes.writeout_histogram(true);

#ifdef READMEMORY
    measurement mes2(prgname, "read", NUMBER_REPEAT);

    cout << "\nRead from memory" << endl;

    for(int n=0; n < NUMBER_REPEAT; n++)
    {
        mem = (long *)ptr;
        mes2.start(-1, 0, comment);

        // a little bit of loop unrolling

        // this reads 256 bytes if sizeof(long) is 4 bytes

        for(i=0; i < numberoftimes; i++)
        {
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
```

```
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;

            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
            value= *mem++; value= *mem++; value= *mem++; value= *mem++;
        }

        mes2.end();
    }

    mes2.writeout_histogram(true);
#endif /* READMEMORY */

}



// Main ///////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    cout << "This program measures the throughput of normal and shared memory."
         << "\nAll measurements are done " << NUMBER_REPEAT << " times with "
         << "memory regions\nof the size " << SHM_SIZE << " bytes.\n" << endl;


    // two measurements: one in normal memory, one in shared


    // normal memory

    char *mem= new char[SHM_SIZE];

    writememory(mem, argv[0], "Normal Memory:");

    delete[] mem;


    cout << endl;


    // shared memory

    int shmid;
    char *shmptr;

    // get shared memory region
```

```
    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) == -1)
        error.system("shmget error");

    if ((shmptr = (char *)shmat(shmid, 0, 0)) == (char *)-1)
        error.system("shmat error");

    writememory(shmptr, argv[0], "Shared memory:");

    // destroy shared memory region

    if (shmdt(shmptr) == -1)
        error.system("shmdt error");

    if (shmctl(shmid, IPC_RMID, 0) == -1)
        error.system("shmctl error");

    return(EXIT_SUCCESS);
}
```

A.5.5 performance/message.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: message.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure how long it takes to send/receive SYSV IPC messages
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/02 02:30:17 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
```

```
// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 10000;        // how often for measurements

#ifndef MSGMAX
#define MSGMAX 2048
#endif

const int size_max = MSGMAX;            // maximum size of messages

const int MSG_MODE = 0666; // MSG_R | MSG_W;        // read and write

const int measure_max = 32;             // maximum number of measurements



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int i;


    // do some performance measurements

    // check size

#ifdef linux
    if( size_max > MSGMAX)
```

```
        error.panic("MSGMAX too small, reduce size_max!");
#endif

    // create message queue

    int msgid = msgget(IPC_PRIVATE, MSG_MODE);
    if(msgid == -1)
        error.system("msgget error");


    synchronization_init();

    // get buffer which we can work

    struct msgbuf *buffer = (struct msgbuf *)malloc(size_max+4);
                             // +4 for long at beginning of buffer
    if(!buffer)
        error.panic("out of memory");

    // set type (priority)
    buffer->mtype = 1;      // has to be positive


    pid_t childpid;
    int size;

    childpid = fork();
    if(childpid == -1)
        error.system("fork error");

    if(childpid == 0)
    {
        // child, will receive data from parent

        for(size = 1; size <= size_max; size*=2)
        {
            for(i = 0; i < NUMBER_REPEAT; i++)
            {
                if( msgrcv(msgid, buffer, size_max+4, 0, 0) == -1)
                    error.system("msgrcv error");
            }
        }

        synchronization_signal_parent(getppid());
    }
    else
    {
        // parent, will send data to child

        measurement mes(argv[0], "write", measure_max);

        for(size = 1; size <= size_max; size*=2)
        {
            // start timer
            mes.start(size, NUMBER_REPEAT);

            for(i = 0; i < NUMBER_REPEAT; i++)
            {
```

```
                if( msgsnd(msgid, buffer, size, 0) == -1)
                    error.system("msgsnd error");
            }

            // end timer
            mes.end();
        }


        // wait till child has read all data
        synchronization_wait();

        mes.writeout_logfile(true, true, true);     // use cout as output file
        mes.writeout_plain_logfile(true, false);    // and write gnuplot data file

        // destroy message queue

        if(msgctl(msgid, IPC_RMID, NULL) == -1)
            error.system("msgctl error");
    }

    if(buffer)
        free(buffer);

    return(EXIT_SUCCESS);
}
```

A.5.6 performance/pipes.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: pipes.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure how fast pipes are
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/27 19:07:59 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <unistd.h>
#include <string.h>


// Local headers ///////////////////////////////////////////// Local headers //
```

```
#include "../common.h"


// Macros ////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //
    /* NONE */


// Main ////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    pid_t parent = getpid();
    pid_t child;

    // esablish signal handler
    synchronization_init();

    // open connection

    int filedes[2];

    if( pipe(filedes) == -1)
        error.system("pipe error");

    if( (child = fork() ) < 0)
        error.system("fork error");

    if(child == 0)
```

```
    {
        // child

        readdata(argv[0], "read", filedes[0], parent);  // synchronized
        readdata(argv[0], "read2", filedes[0], 0);      // unsynchronized
    }
    else
    {
        // parent

        writedata(argv[0], "write", filedes[1], child); // synchronized
        writedata(argv[0], "write2", filedes[1], 0);    // unsynchronized
    }

    return(EXIT_SUCCESS);
}
```

A.5.7 performance/semaphore.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: semaphore.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure how fast semaphore operations are
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 13:49:41 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */
```

```
// File scope objects ////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 10000;     // how often for measurements


const int SEM_MODE = 0666;           // SEM_R | SEM_A;  // read and alter



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main //////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int i;


    // do some performance measurements

    // check size

#ifdef linux
    if( NUMBER_REPEAT > SEMVMX)
        error.panic("SEMVMX too small, reduce NUMBER_REPEAT!");
#endif

    // create semaphore

    int semid = semget(IPC_PRIVATE, 1, SEM_MODE);
    if(semid == -1)
        error.system("semget error");


    // set to initial value
```

```
#ifdef linux
    semun initial_value;
#else
    union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
    } initial_value;
#endif

    initial_value.val = 0;

    if( semctl(semid, 0, SETVAL, initial_value) == -1)
        error.system("semctl error");


    // initialize default values

    struct sembuf op;
    op.sem_num = 0;     // member number (we only have number 0)
    op.sem_op  = 1;     // inc
    op.sem_flg = 0;     // no flags


    measurement mes(argv[0], "mes", 4);


    // do measurements without undo


    // start timer
    mes.start(-1, NUMBER_REPEAT, "Measuring semop() +1 operations without undo");

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        if( semop(semid, &op, 1) == -1)
            error.system("semop error");
    }

    // end timer
    mes.end();


    // now do the reverse..

    op.sem_op = -1;     // dec

    // start timer
    mes.start(-1, NUMBER_REPEAT, "Measuring semop() -1 operations without undo");

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        if( semop(semid, &op, 1) == -1)
            error.system("semop error");
    }

    // end timer
    mes.end();
```

```
    // do measurements with undo

    op.sem_op  = 1;         // inc
    op.sem_flg = SEM_UNDO;  // remeber change if program crashes or exit()s

    // start timer
    mes.start(-1, NUMBER_REPEAT, "Measuring semop() +1 operations with undo");

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        if( semop(semid, &op, 1) == -1)
            error.system("semop error");
    }

    // end timer
    mes.end();


    // now do the reverse..

    op.sem_op = -1;     // dec

    // start timer
    mes.start(-1, NUMBER_REPEAT, "Measuring semop() -1 operations with undo");

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        if( semop(semid, &op, 1) == -1)
            error.system("semop error");
    }

    // end timer
    mes.end();



    // destroy semaphore

#ifdef linux
    union semun nop;
#else
    int nop=0;
#endif

    if(semctl(semid, 0, IPC_RMID, nop) == -1)
        error.system("semctl error");


    mes.writeout_logfile(false, true, true);    // use cout for output


    return(EXIT_SUCCESS);
}
```

A.5.8 performance/shared.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: shared.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure how long it takes to add/remove a shared memory segment
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ----------------------------------------------
// 96/02/29 07:17:38 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>



// Local headers //////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 10000;// how often for measurements

const int measurement_max = 64; // how many measurements do we want to do?
                                // 16 should be enough....

const int SHM_SIZE = 100000;    // shared memory size for getting number of
                                // available shared memory segments

const int SHM_MODE = (SHM_R | SHM_W);       // user read/write


// External variables, functions, and classes ///////////// External objects //
    /* NONE */
```

```
// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int i;


    // do some performance measurements

    cout << "Measuring how long it takes to create, attach, detach and\n"
         << "destroy a shared memory region (buffersize = how big is region "
         << "in bytes)" << endl << endl;

    measurement mes(argv[0], "mes", measurement_max);

    int size;
    int shmid;
    char *shmptr;

#ifndef SHMMAX
#define SHMMAX 2000000
#endif

    for(size=1024; size <= SHMMAX; size*=2 )
    {
        // start timer

        mes.start(size, NUMBER_REPEAT);

        for(i=0; i < NUMBER_REPEAT; i++)
        {
            // get shared memory

            if ((shmid = shmget(IPC_PRIVATE, size, SHM_MODE)) == -1)
                error.system("shmget error");

            if ((shmptr = (char *)shmat(shmid, 0, 0)) == (char *)-1)
                error.system("shmat error");
```

```
            // destry shared memory

            if (shmdt(shmptr) == -1)
                error.system("shmdt error");

            if (shmctl(shmid, IPC_RMID, 0) == -1)
                error.system("shmctl error");
        }

        // store data about measurement

        mes.end();

    }

    mes.writeout_logfile(false, true, true);// use cout to display results
    mes.writeout_plain_logfile(false, true);// and write gnuplot result file


    return(EXIT_SUCCESS);
}
```

A.5.9 performance/signaloverhead.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: signaloverhead.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure the overhead of synchronizing tasks with signals
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 00:27:43 muellerg: created
// 96/02/27 00:06:14 muellerg: updated, now uses the functions
//                             synchronization_XXX
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <unistd.h>
#include <signal.h>



// Local headers ///////////////////////////////////////////// Local headers //
```

```
#include "../common.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 100000;          /* how often are signals send? */



// External variables, functions, and classes //////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //
    /* NONE */


// Main /////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    cout << "This program measures how long it takes to send a signal" << endl;
    cout << "from one process to another and getting a reply signal ";
    cout << "back." << endl;
    cout << endl;

    pid_t childpid;
    pid_t parentpid=getpid();

    // set up signal handler

    synchronization_init();

    // spawn child

    if ( (childpid = fork()) < 0)
```

```
    error.system("fork error");
else
    if (childpid == 0)
    {
        // child

        // inform parent that we are existing

        synchronization_signal_parent(parentpid);

        for(int i=0; i < NUMBER_REPEAT; i++)
        {
            // wait for signal

            synchronization_wait();

            // and send signal to parent

            synchronization_signal_parent(parentpid);
        }

        return(EXIT_SUCCESS);   // and exit..
    }


// parent


measurement mes(argv[0], "", 1);


// wait for "I am alive" signal of child

synchronization_wait();

// start timer
mes.start(-1, NUMBER_REPEAT);

for(int i=0; i < NUMBER_REPEAT; i++)
{
    // send signal

    synchronization_signal_child(childpid);

    // wait for signal of child (receive acknowledgment)

    synchronization_wait();
}

// end timer
mes.end();


// write ops/s on cout

mes.writeout_logfile(false, true, true);

return(EXIT_SUCCESS);
```

```
}
```

A.5.10 performance/socketlocal.c

```
//////////////////////////////////////////////////////////////////////////////
// Filename: socketlocal.c
//////////////////////////////////////////////////////////////////////////////
// Purpose: test the performance of local sockets via socketpair()
//////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ----------------------------------------------
// 96/02/29 05:54:20 muellerg: created
//
//////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */
```

```
// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ////////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    pid_t parent = getpid();
    pid_t child;

    // esablish signal handler
    synchronization_init();

    // open connection

    int socketdes[2];

              // namespace, style,       protocol, filedes
    if( socketpair(AF_UNIX, SOCK_STREAM, 0       , socketdes) == -1)
        error.system("socketpair() error");

    if( (child = fork() ) < 0)
        error.system("fork error");

    if(child == 0)
    {
        // child

        readdata(argv[0], "read", socketdes[0], parent);    // synchronized
        readdata(argv[0], "read2", socketdes[0], 0);        // unsynchronized
    }
    else
    {
        // parent

        writedata(argv[0], "write", socketdes[1], child);   // synchronized
        writedata(argv[0], "write2", socketdes[1], 0);      // unsynchronized
    }

    return(EXIT_SUCCESS);
}
```

A.5.11 performance/systemcalloverhead.c

```
////////////////////////////////////////////////////////////////////////////////
```

```
// Filename: systemcalloverhead.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measure the overhead of a system call
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/15 18:28:24 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>


// Local headers //////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros ////////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 500000;           /* how often are signals send? */




// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */
```

```
// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    cout << "This program measures the overhead of a system call." << endl;
    cout << endl;

    measurement mes(argv[0], "", 1);

    // start timer
    mes.start(-1, NUMBER_REPEAT);

    pid_t dummy;

    for(int i=0; i < NUMBER_REPEAT; i++)
    {
        // so very cheap system call (nearly no work done in kernel)

        dummy = getpid();
    }

    // end timer
    mes.end();


    // write ops/s on cout

    mes.writeout_logfile(false, true, true);

    return(EXIT_SUCCESS);
}
```

A.5.12 performance/tcpsclientoverhead.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: tcpsclientoverhead.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measures the overhead in establishing a TCP connection
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 10:39:38 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>



// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 100;                  // how often
const int measurement_max = NUMBER_REPEAT;      // space for measurements


// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */
```

```
// Main ////////////////////////////////////////////////////////////// Main //


/*
 * Example of client using the TCP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=3)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port" << endl;
        exit(EXIT_FAILURE);
    }

    int sockfd;
    struct sockaddr_in  serv_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber < 1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }

    // Fill in the structure "serv_addr" with the address of the
    // server that we want to connect with.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // addr of host
    serv_addr.sin_port        = htons(portnumber);


    int i;


    measurement mes(argv[0], "mes", measurement_max);

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        mes.start();

        // create a TCP socket (an Internet stream socket).

        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
            error.system("client: can't create stream socket");


        // connect to the server

        if (connect(sockfd, (struct sockaddr *) &serv_addr,
                                sizeof(serv_addr)) < 0)
            error.system("client: can't connect to server");


        // do something


        // close connection

        close(sockfd);

        mes.end();

        ssleep(100000); // sleep for 100000 mikroseconds = 100 ms = 0.1 s
                        // to allow the server to cope with all the
                        // connections
    }

    mes.writeout_histogram();
    mes.writeout_plain_histogram();

    return(EXIT_SUCCESS);
}
```

A.5.13 performance/tcpsclientthroughput.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: tcpsclientthroughput.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measures performance of a TCP connection (socket version)
//          uses tcpserverconcurrent(.c) as TCP server
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 17:00:19 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ///////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects ///////////////////////////////// File scope objects //
    /* NONE */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions /////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //
    /* NONE */


// Main /////////////////////////////////////////////////////////////// Main //


/*
 * Example of client using the TCP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
```

```
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=3)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port" << endl;
        exit(EXIT_FAILURE);
    }

    int sockfd;
    struct sockaddr_in  serv_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }

    // Fill in the structure "serv_addr" with the address of the
    // server that we want to connect with.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // addr of host
    serv_addr.sin_port        = htons(portnumber);


    // create a TCP socket (an Internet stream socket).

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error.system("client: can't create stream socket");


    // connect to the server

    if (connect(sockfd, (struct sockaddr *) &serv_addr,
                        sizeof(serv_addr)) < 0)
        error.system("client: can't connect to server");

    writedata(argv[0], "write", sockfd);

    close(sockfd);

    return(EXIT_SUCCESS);
}
```

A.5.14 performance/tcpsserverconcurrentoverhead.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: tcpserverconcurrentoverhead.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: needed for measurements by tcpsclientoverhead.c
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 11:40:52 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers ///////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>



// Local headers ////////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros //////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects //////////////////////////////////// File scope objects //

const int BACKLOG = 5;      // maximum number of pending connections
```

```
// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ////////////////////////////////////////////////////////////// Main //




/*
 * Example of a concurrent server using the TCP protocol
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    int sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in  cli_addr, serv_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }
```

```
    // establish a function that changes the signalhandler
    // to report child death so that they can be catched and
    // zombie processes are avoided

#ifndef hpux
    catch_children();
#endif

    // create a TCP socket (an Internet stream socket)

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error.system("server: can't create stream socket");


    // bind local address so that the client can send to us

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error.system("server: can't bind local address");

    // listen for connection on socket with a maximum length of
    // pending connections

    listen(sockfd, BACKLOG);


    while(true)
    {
        // wait for a connection from a client process.
        // This server is an example of a concurrent server.

        clilen = sizeof(cli_addr);


        // check for zombies

        waitpid(WAIT_ANY, NULL, WNOHANG);


        // accept connection

        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

        if (newsockfd < 0)
            error.system("server: accept error");


        // spawn child process to handle request

        if ( (childpid = fork()) < 0)
            error.system("server: fork error");
        else
            if (childpid == 0)
            {   // child process
```

```
                close(sockfd);          // close original socket

                // do nothing for performance measurement

                return(EXIT_SUCCESS);   // and exit..
            }

        close(newsockfd);               // parent process: close socket
                                        // to be able to reuse it for the
                                        // next connection
    }
}
```

A.5.15 performance/tcpsserverconcurrent.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: tcpserverconcurrent.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measures performance of a TCP connection (socket version)
//         uses tcpsclientthroughput(.c) as TCP client
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 16:52:19 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>



// Local headers ///////////////////////////////////////////// Local headers //
```

```
#include "../common.h"


// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //

const int BACKLOG = 5;      // maximum number of pending connections


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //
    /* NONE */


// Main ///////////////////////////////////////////////////////////////// Main //


/*
 * Example of a concurrent server using the TCP protocol
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    // establish a function that changes the signalhandler
```

```
    // to report child death so that they can be catched and
    // zombie processes are avoided

#ifndef hpux
    catch_children();
#endif

    int sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in  cli_addr, serv_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // create a TCP socket (an Internet stream socket)

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error.system("server: can't create stream socket");


    // bind local address so that the client can send to us

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error.system("server: can't bind local address");

    // listen for connection on socket with a maximum length of
    // pending connections

    listen(sockfd, BACKLOG);


    while(true)
    {

        // wait for a connection from a client process.
        // This server is an example of a concurrent server.

        clilen = sizeof(cli_addr);


        // check for zombies

        waitpid(WAIT_ANY, NULL, WNOHANG);
```

```
        // accept connection

        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

        if (newsockfd < 0)
            error.system("server: accept error");


        // spawn child process to handle request

        if ( (childpid = fork()) < 0)
            error.system("server: fork error");
        else
            if (childpid == 0)
            {   // child process

                close(sockfd);          // close original socket

                readdata(argv[0], "read", newsockfd); // process the request

                return(EXIT_SUCCESS);   // and exit..
            }

        close(newsockfd);               // parent process: close socket
                                        // to be able to reuse it for the
                                        // next connection
    }
}
```

A.5.16 performance/timeroverhead.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: timeroverhead.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measures the overhead of using the timer classes
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/14 00:58:48 muellerg: created
// 96/03/11 00:41:11 muellerg: removed some parts from compilation, as their
//                             result says more or less nothing
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers ////////////////////////////////////////////// System headers //

#include <stdlib.h>
```

```
// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"


// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */


// File scope objects /////////////////////////////////// File scope objects //

const int NUMBER_REPEAT = 1000000; /* how often are clock measurements done? */


// External variables, functions, and classes ///////////// External objects //
    /* NONE */


// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */


// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */


// Functions and class implementation /// Functions and class implementation //
    /* NONE */


// Main /////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int i;

    cout << "This program measures the overhead in using the timer classes "
         << "\"cputimer\" and \"clocktimer\"" << endl << endl;

#ifdef pedantic
    cout << "First all measurements are made once..." << endl << endl;


    cout << "Measuring the overhead of clocktimer..." << endl;
#endif
    clocktimer c1;
```

```
    // do dummy operations, as Linux needs about 470 usecs for the first
    // time... strange... probably some internal things have to be
    // initialized...

    c1.start();             // start clock timer
    c1.end();               // end clock timer

#ifdef pedantic
    c1.start();             // start clock timer
    c1.end();               // end clock timer

    cout << "Overhead: clock secs:" << c1.secs()
         << "  clock usecs:" << c1.usecs() << endl << endl;




    cout << "Measuring the overhead of cputimer..." << endl;
#endif
    cputimer c2;

#ifdef pedantic
    c2.start();             // start cpu timer
    c2.end();               // end cpu timer

    cout << "Overhead: cpu secs:" << c2.secs()
         << "  cpu usecs:" << c2.usecs() << endl << endl;




    cout << "Measuring the overhead of clocktimer and cputimer..." << endl;

    c1.start();
    c2.start();
    c2.end();
    c1.end();

    cout << "Overhead: clock secs:" << c1.secs()
         << "  clock usecs:" << c1.usecs() << endl;
    cout << "Overhead: cpu secs:" << c2.secs()
         << "  cpu usecs:" << c2.usecs() << endl << endl;


    cout << "Second ";
#endif
    cout << "all measurements are made " << NUMBER_REPEAT
         << " times..." << endl << endl;


    cout << "Measuring the overhead of the " << NUMBER_REPEAT
         << "-loop..." << endl;


    c1.start();
    c2.start();

    // This measurement is not bullet-proof: the compiler might optimize
```

```
// this loop much easier as the following ones. But it is better
// than nothing...

for(i=0; i < NUMBER_REPEAT; i++)
{
    ;
}

c2.end();
c1.end();

cout << "Overhead: clock secs:" << c1.secs()
     << "  clock usecs:" << c1.usecs() << endl;
cout << "Overhead: cpu secs:" << c2.secs()
     << "  cpu usecs:" << c2.usecs() << endl << endl;



cout << "Measuring the overhead of clocktimer..." << endl;

clocktimer  temp1;
cputimer    temp2;


c1.start();
c2.start();

for(i=0; i < NUMBER_REPEAT; i++)
{
    temp1.start();
    temp1.end();
}

c2.end();
c1.end();

cout << "Overhead: clock secs:" << c1.secs()
     << "  clock usecs:" << c1.usecs() << endl;
cout << "Overhead: cpu secs:" << c2.secs()
     << "  cpu usecs:" << c2.usecs() << endl;
cout << "Average times:"
     << "  clock secs:" << (float)(c1.secs()+(c1.usecs()/1000000.0)) /
                                                  (float)NUMBER_REPEAT
     << "(=" << (c1.secs()*1000000.0+c1.usecs())/NUMBER_REPEAT << " usecs)
     << "  cpu secs:" << (float)(c2.secs()+(c2.usecs()/1000000.0)) /
                                                  (float)NUMBER_REPEAT
     << "(=" << (c2.secs()*1000000.0+c2.usecs())/NUMBER_REPEAT << " usecs)"
     << endl << endl;


cout << "Measuring the overhead of cputimer..." << endl;

c1.start();
c2.start();

for(i=0; i < NUMBER_REPEAT; i++)
{
    temp2.start();
```

```
        temp2.end();
    }

    c2.end();
    c1.end();

    cout << "Overhead: clock secs:" << c1.secs()
         << "  clock usecs:" << c1.usecs() << endl;
    cout << "Overhead: cpu secs:" << c2.secs()
         << "  cpu usecs:" << c2.usecs() << endl;
    cout << "Average times:"
         << "  clock secs:" << (float)(c1.secs()+(c1.usecs()/1000000.0)) /
                                                        (float)NUMBER_REPEAT
         << "(=" << (c1.secs()*1000000.0+c1.usecs())/NUMBER_REPEAT << " usecs)"
         << "  cpu secs:" << (float)(c2.secs()+(c2.usecs()/1000000.0)) /
                                                        (float)NUMBER_REPEAT
         << "(=" << (c2.secs()*1000000.0+c2.usecs())/NUMBER_REPEAT << " usecs)"
         << endl << endl;

    return(EXIT_SUCCESS);
}
```

A.5.17 performance/udps.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: udps.h
////////////////////////////////////////////////////////////////////////////////
// Purpose: some common definitions for udpsserver.c and udpssclient.c
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 18:12:16 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //
    /* NONE */



// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */



// Macros /////////////////////////////////////////////////////////// Macros //

#ifdef hpux
```

```
const int MAX_BUFFER = 8192;        // maximum buffer size
#else
const int MAX_BUFFER = 16384;       // maximum buffer size
#endif


const int NUMBER_REPEAT = 1000;     // how often?
const int max_size = MAX_BUFFER;    // maximum size to send/receive
const int max_measurement = 32;     // maximum number of measurements

// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */
```

A.5.18 performance/udpsclient.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: udpsclient.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: measure performance of UDP (this file: UDP client) (socket version)
//          needs udpsserver(.c) as UDP server
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 17:56:03 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
```

```
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#ifndef sun
#include <strings.h>
#endif
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
#include "udps.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * send data to UDP server and measure time
 *
 * in: sockfd:     fd to use
 *     pserv_addr: ptr to appropriate sockaddr_XX structure
```

```
 *     servlen:    actual sizeof(*pserv_addr)
 */

void do_it(char *name, int sockfd, struct sockaddr *pserv_addr, int servlen)
{
    int     size, i, n;

    char *mesg;
    mesg = new char[MAX_BUFFER];


    // measure throughput

    measurement mes(name, "mes", max_measurement);

    for(size=1; size <= max_size; size*=2)
    {
        mes.start(size, NUMBER_REPEAT);

        for(i=0; i < NUMBER_REPEAT; i++)
        {
            if (sendto(sockfd, mesg, size, 0, pserv_addr, servlen) != size)
                    error.system("sendto error");

            // get acknowledgment

            n = recvfrom(sockfd, mesg, MAX_BUFFER, 0,
                    (struct sockaddr *) 0, (int *) 0);
            if (n < 0)
                error.system("recvfrom error");
        }

        mes.end();
    }


    // measure difference in speed

    measurement mes2(name, "mes2", NUMBER_REPEAT);

    size = 4;       // one long word

    for(i=0; i < NUMBER_REPEAT; i++)
    {
        mes2.start(-1, 0);

        if (sendto(sockfd, mesg, size, 0, pserv_addr, servlen) != size)
                error.system("sendto error");

        // get acknowledgment

        n = recvfrom(sockfd, mesg, MAX_BUFFER, 0,
                (struct sockaddr *) 0, (int *) 0);
        if (n < 0)
            error.system("recvfrom error");

        mes2.end();
    }
```

```
    mes.writeout_logfile(true, true);
    mes.writeout_plain_logfile();

    mes2.writeout_histogram();
    mes2.writeout_plain_histogram();
}




// Main ///////////////////////////////////////////////////////////////// Main //

/*
 * Example of client using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: server internet address (in dot-form)
 *   argv[2]: port number of server
 */


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=3)
    {
        cerr << "Usage: " << argv[0] << " serveraddr port" << endl;
        exit(EXIT_FAILURE);
    }


    int sockfd;
    struct sockaddr_in  cli_addr, serv_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[2]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // Fill in the structure "serv_addr" with the address of the
    // server that we want to send to.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // server addr
    serv_addr.sin_port        = htons(portnumber);
```

```
    // create a UDP socket (an Internet datagram socket)

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        error.system("client: can't create datagram socket");


    // bind any local address for us

    bzero((char *) &cli_addr, sizeof(cli_addr));    /* zero out */
    cli_addr.sin_family      = AF_INET;
    cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    cli_addr.sin_port        = htons(0);

    if (bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
        error.system("client: can't bind local address");


    // do something...

    do_it(argv[0], sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

    close(sockfd);

    return(EXIT_SUCCESS);
}
```

A.5.19 performance/udpsserver.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: udpsserver.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: measure performance of UDP (this file: UDP server) (socket version)
//          needs udpsclient(.c) as UDP client
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/01 17:57:19 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#include <string.h>
#ifndef sun
#include <strings.h>
#endif

// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
#include "udps.h"



// Macros ///////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects ////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //

/*
 * read (and discard) all incoming data, send a byte as reply back
 *
 * in: sockfd   : fd to use
 *     pcli_addr: ptr to appropriate sockaddr_XX structure
 *     maxclilen: sizeof(*pcli_addr)
 */

void do_it(int sockfd, struct sockaddr *pcli_addr, int maxclilen)
{
    int     n, clilen;

    char *mesg;
    mesg = new char[MAX_BUFFER];

    for ( ; ; )
    {
        clilen = maxclilen;
        n = recvfrom(sockfd, mesg, MAX_BUFFER, 0, pcli_addr, &clilen);
```

```
        if (n < 0)
            error.system("recvfrom error");

        // send a byte as acknowledgment back:

        if (sendto(sockfd, mesg, 1, 0, pcli_addr, clilen) != 1)
            error.system("sendto error");
    }
}




// Main /////////////////////////////////////////////////////////////// Main //

/*
 * Example of server using the UDP protocol.
 *
 * paramteters:
 *
 *   argv[1]: port number to bind() & listen() to
 */

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    if(argc!=2)
    {
        cerr << "Usage: " << argv[0] << " port" << endl;
        exit(EXIT_FAILURE);
    }

    int sockfd;
    struct sockaddr_in  serv_addr, cli_addr;
    int portnumber = -1;


    // get port number

    portnumber = atoi(argv[1]);
    if(portnumber <1)
    {
        cerr << "illegal port number" << endl;
        exit(EXIT_FAILURE);
    }


    // create a UDP socket (an Internet datagram socket)

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        error.system("server: can't create datagram socket");


    // bind local address so that the client can send to us.

    bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(portnumber);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error.system("server: can't bind local address");

    // do something...

    do_it(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));

    return(EXIT_SUCCESS);
}
```

A.6 Program Environment

A program often has to find out in what kind of environment it runs or which facilities the operating system has. The programs in the `environment` directory (see Table 36) show how to find out various things about a used operating system / computer. These programs were used for getting information about the used systems. Most of the programs are not described directly in the main document. The WWW version of this document contains hyper-links to the results of the programs on the three used systems.

File Name	**Purpose**	**Page**
`Makefile`	Makefile for this directory	286
`confstrtest.c`	shows how to get configuration dependent string variables	287
`getpipecapacity.c`	get capacity of pipes	288
`hostadress.c`	print (internet) addresses and aliases of current host	291
`systemtype.c`	prints information about the hardware and software the program is running on	293
`sysconftest.c`	print all constants that are available via `sysconf()`	295
`sysvipclimits.c`	obtain the maximum nuber of System V IPC objects per process	295

Table 36: Purpose of Files in the `environment` Directory

A.6.1 environment/Makefile

```
include ../Makefile.def

ALL = systemtype confstrtest hostadress getpipecapacity sysconftest \
      sysvipclimits

all: $(ALL)

# get environment data

dataget:
    systemtype >systemtype.log
    confstrtest >confstrtest.log
    hostadress >hostadress.log
    getpipecapacity >getpipecapacity.log
    sysconftest >sysconftest.log
```

```
    sysvipclimits >sysvipclimits.log

# rename files

datamove:
    echo $(HOSTTYPE) $(OSTYPE)
    # move data files
    mv confstrtest.log confstrtest.log.$(OSTYPE)
    mv getpipecapacity.log getpipecapacity.log.$(OSTYPE)
    mv hostadress.log hostadress.log.$(OSTYPE)
    mv sysconftest.log sysconftest.log.$(OSTYPE)
    mv systemtype.log systemtype.log.$(OSTYPE)
    mv sysvipclimits.log sysvipclimits .log.$(OSTYPE)

include ../Makefile.def2
```

A.6.2 environment/confstrtest.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: confstrtest.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: get configuration dependent string variables
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 02:01:07 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //

#define __USE_POSIX_2



// System headers /////////////////////////////////////////// System headers //

#include <unistd.h>
#include <stdlib.h>
#include <iostream.h>




// Local headers ///////////////////////////////////////////// Local headers //
    /* NONE */



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */
```

```
// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //
    /* NONE */


int
main(void)
{
#ifdef sun
    cout << "confstr not supported by sun" << endl;
#else
    char *pathbuf; size_t n;

    n = confstr(_CS_PATH,NULL,(size_t)0);

    if ((pathbuf = (char *)malloc(n)) == NULL) abort();
        confstr(_CS_PATH, pathbuf, n);

    cout << pathbuf << endl;

    free(pathbuf);
#endif
    return 0;
}
```

A.6.3 environment/getpipecapacity.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: getpipecapacity.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: get capacity of a pipe
```

```
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/02 03:20:31 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //

#ifndef hpux
#define USE_SELECT 1
#else
#define USE_POLL 1
#endif



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <string.h>

#ifdef USE_POLL
#include <poll.h>
// #include <stropts.h>
#endif




// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */
```

```
// Signal catching functions /////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    int i, n, fd[2];

#ifdef USE_SELECT
    fd_set writeset;
    struct timeval  tv;

    if (pipe(fd) < 0)
        error.system("pipe error");

    FD_ZERO(&writeset);

    for (n = 0; ; n++)
    {
        // write 1 byte at a time until pipe is full

        FD_SET(fd[1], &writeset);

        tv.tv_sec = tv.tv_usec = 0;     // don't wait at all

        if ( (i = select(fd[1]+1, NULL, &writeset, NULL, &tv)) < 0)
            error.system("select error");
        else if (i == 0)
            break;

        if (write(fd[1], "X", 1) != 1)
            error.system("write error");
    }
#endif


#ifdef USE_POLL
    struct pollfd fdarray[1];

    if (pipe(fd) < 0)
        error.system("pipe error");
```

```
    for (n = 0; true; n++)
    {
        fdarray[0].fd = fd[1];
        fdarray[0].events = POLLOUT;

        if ( (i = poll(fdarray, 1, 0)) < 0)
            error.system("poll error");
        else if (i == 0)
            break;

        if (write(fd[1], "X", 1) != 1)
            error.system("write error");
    }
#endif


    cout << "pipe capacity = " << n << endl;

    exit(EXIT_SUCCESS);
}
```

A.6.4 environment/hostadress.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: hostadress.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: get some informations about host where program is running
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 02:04:08 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/param.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"
```

```
// Macros ///////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int maxhostnamelen = MAXHOSTNAMELEN+1;



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ///////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    char name[maxhostnamelen];
    struct hostent *entry;
    int i=0;


    // get name of host

    if (gethostname(name, maxhostnamelen) != 0)
        error.system("unknown name!");

    cout << "host name: \"" << name << "\"" << endl;



    // get more info with the help of our name

    if ((entry = gethostbyname(name)) == NULL)
        error.system("no host name info");
```

```
    cout << "official name: \"" << entry->h_name << "\"" << endl;

    // print aliases as well

    char **aliases=entry->h_aliases;
    for(i=1; aliases[i]; i++)
        cout << "also known as: \"" << aliases[i] << "\""<< endl;


    // and numerical...

    switch(entry->h_addrtype)
    {
        case AF_INET: {
                        in_addr ia;
                        ia.s_addr=*((unsigned long *)(entry->h_addr_list[0]));

                        cout << "address: " << inet_ntoa(ia) << endl;

                        for(unsigned long **aliases=
                            (unsigned long **)entry->h_addr_list, i=1;
                            aliases[i]; i++)
                        {
                            ia.s_addr=*aliases[i];
                            cout << "or: " << inet_ntoa(ia) << endl;
                        }
                      }
                      break;

        default:
                      error.warning("unknown address type");
                      break;

    }

    return(EXIT_SUCCESS);
}
```

A.6.5 environment/systemtype.c

```
///////////////////////////////////////////////////////////////////////////////
// Filename: systemtype.c
///////////////////////////////////////////////////////////////////////////////
// Purpose: get some information over system where program is running
///////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/06 02:08:54 muellerg: created
//
///////////////////////////////////////////////////////////////////////////////
```

```
// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */


// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/utsname.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //
    /* NONE */



// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions ///////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions /////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main /////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    struct utsname un;

    if ( uname( &un ) == -1 ) // if -1, an error occurred
        error.system("uname failed");
```

```
    // no errors occurred, so display the fields

    cout    << "System name:\t" << un.sysname << endl
            << "Node name:\t"   << un.nodename << endl
            << "Release:\t"     << un.release << endl
            << "Version:\t"     << un.version << endl
            << "Machine:\t"     << un.machine << endl

#ifdef _HPUX_SOURCE
            << "ID Number:\t"   << un.__idnumber << endl
#endif

            << endl;

    return(EXIT_SUCCESS);
}
```

A.6.6 environment/sysconftest.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: sysconftest.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: get all constansts that are available via sysconf()
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/02/29 02:41:48 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches //////////////////////////// Feature test switches //

// #define _POSIX_SOURCE 1
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <unistd.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */
```

```
// File scope objects ////////////////////////////////// File scope objects //


struct testentry
{
    int name;
    char *argname;
    char *description;
};

struct testentry tests[] =
{
   {   _SC_ARG_MAX,
       "_SC_ARG_MAX",
              "The  maximum  length of the arguments to the exec()\n"
              "family of functions;  the  corresponding  macro  is\n"
              "ARG_MAX.\n"
   },
   {   _SC_CHILD_MAX,
       "_SC_CHILD_MAX",
              "The  number  of simultaneous processes per user id,\n"
              "the corresponding macro is _POSIX_CHILD_MAX.\n"
   },
   {   _SC_CLK_TCK,
       "_SC_CLK_TCK",
              "The number of clock ticks per  second;  the  corre-\n"
              "sponding macro is CLK_TCK.\n"
   },
   {   _SC_STREAM_MAX,
       "_SC_STREAM_MAX",
              "The  maximum  number  of streams that a process can\n"
              "have open at any  time.   The  corresponding  POSIX\n"
              "macro  is  STREAM_MAX, the corresponding standard C\n"
              "macro is FOPEN_MAX.\n"
   },
   {   _SC_TZNAME_MAX,
       "_SC_TZNAME_MAX",
              "The maximum number of bytes in a timezone name, the\n"
              "corresponding macro is TZNAME_MAX.\n"
   },
   {   _SC_OPEN_MAX,
       "_SC_OPEN_MAX",
              "The maximum number of files that a process can have\n"
              "open  at  any  time,  the  corresponding  macro  is\n"
              "_POSIX_OPEN_MAX.\n"
   },
   {   _SC_JOB_CONTROL,
       "_SC_JOB_CONTROL",
              "This indicates whether POSIX - style job control is\n"
              "supported,    the    corresponding    macro     is\n"
              "_POSIX_JOB_CONTROL.\n"
   },
   {   _SC_SAVED_IDS,
       "_SC_SAVED_IDS",
              "This  indicates  whether a process has a saved set-\n"
              "user-ID and a saved set-group-ID; the corresponding\n"
              "macro is _POSIX_SAVED_IDS.\n"
   },
```

```
    {   _SC_VERSION,
        "_SC_VERSION",
               "indicates  the  year and month the POSIX.1 standard\n"
               "was  approved  in  the  format  YYYYMML;the   value\n"
               "199009L indicates the most recent revision, 1990.\n"
    },
//       Next, the POSIX.2 values:

    {   _SC_BC_BASE_MAX,
        "_SC_BC_BASE_MAX",
               "indicates  the  maximum obase value accepted by the\n"
               "bc(1)   utility;   the   corresponding   macro   is\n"
               "BC_BASE_MAX.\n"
    },
    {   _SC_BC_DIM_MAX,
        "_SC_BC_DIM_MAX",
               "indicates  the  maximum value of elements permitted\n"
               "in an array by bc(1); the  corresponding  macro  is\n"
               "BC_DIM_MAX.\n"
    },
    {   _SC_BC_SCALE_MAX,
        "_SC_BC_SCALE_MAX",
               "indicates the maximum scale value allowed by bc(1);\n"
               "the corresponding macro is BC_SCALE_MAX.\n"
    },
    {   _SC_BC_STRING_MAX,
        "_SC_BC_STRING_MAX",
               "indicates the maximum length of a  string  accepted\n"
               "by bc(1); the corresponding macro is BC_STRING_MAX.\n"
    },
    {   _SC_COLL_WEIGHTS_MAX,
        "_SC_COLL_WEIGHTS_MAX",
               "indicates the maximum numbers of weights  that  can\n"
               "be  assigned  to  an  entry of the LC_COLLATE order\n"
               "keyword in the locale definition file;  the  corre-\n"
               "sponding macro is COLL_WEIGHTS_MAX.\n"
    },
    {   _SC_EXPR_NEST_MAX,
        "_SC_EXPR_NEST_MAX",
               "is  the  maximum number of expressions which can be\n"
               "nested within parentheses by expr(1).   The  corre-\n"
               "sponding macro is EXPR_NEST_MAX.\n"
    },
    {   _SC_LINE_MAX,
        "_SC_LINE_MAX",
               "The  maximum  length  of  a  utility's  input  line\n"
               "length, either from standard input or from a  file.\n"
               "This  includes  length for a trailing newline.  The\n"
               "corresponding macro is LINE_MAX.\n"
    },
    {   _SC_RE_DUP_MAX,
        "_SC_RE_DUP_MAX",
               "The maximum number of  repeated  occurrences  of  a\n"
               "regular   expression  when  the  interval  notation\n"
               "{m,n} is used. The  value  of  the  corresponding\n"
               "macro is RE_DUP_MAX.\n"
    },
    {   _SC_2_VERSION,
```

```
       "_SC_2_VERSION",
              "indicates  the  version  of the POSIX.2 standard in\n"
              "the format of YYYYMML.  The corresponding macro  is\n"
              "POSIX2_VERSION.\n"
   },
#ifndef linux
#ifndef sun
#ifndef hpux
   {   _SC_2_DEV,
       "_SC_2_DEV",
              "indicates  whether  the POSIX.2 C language develop-\n"
              "ment facilities are supported.   The  corresponding\n"
              "macro is POSIX2_C_DEV.\n"
   },
#endif
#endif
#endif
   {   _SC_2_FORT_DEV,
       "_SC_2_FORT_DEV",
              "indicates  whether  the POSIX.2 FORTRAN development\n"
              "utilities are supported.  The  corresponding  macro\n"
              "is POSIX2_FORT_RUN.\n"
   },
   {   _SC_2_FORT_RUN,
       "_SC_2_FORT_RUN",
              "indicates whether the POSIX.2 FORTRAN runtime util-\n"
              "ities are supported.  The  corresponding  macro  is\n"
              "POSIX2_FORT_RUN.\n"
   },
#ifndef linux
#ifndef sun
#ifndef hpux
   {   POSIX2_LOCALEDEF,
       "POSIX2_LOCALEDEF",
              "indicates  whether  the POSIX.2 creation of locates\n"
              "via  locale(1)  is  supported.   The  corresponding\n"
              "macro is POSIX2_LOCALEDEF.\n"
   },
#endif
#endif
#endif
   {   _SC_2_SW_DEV,
       "_SC_2_SW_DEV",
              "indicates  whether the POSIX.2 software development\n"
              "utilities option is supported.   The  corresponding\n"
              "macro is POSIX2_SW_DEV.\n"
   },
// end entry
   {
            0,
            NULL,
            NULL
   }
};
// RETURN VALUE
//       The  value returned is the value of the system resource, 1
//       if a queried option is available, 0 if it is not, or -1 on
//       error.  The variable errno is not set.
```

```
// External variables, functions, and classes ///////////// External objects //
    /* NONE */



// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //
    /* NONE */



// Functions and class implementation /// Functions and class implementation //
    /* NONE */




// Main //////////////////////////////////////////////////////////////// Main //

int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    cout << "This program shows all values for system limits that can" << endl
         << "be obtained via sysconf()" << endl << endl;

    for(unsigned int i=0; tests[i].argname; i++)
    {
        long result = 0;

        result = sysconf(tests[i].name);

        cout << tests[i].argname << ": ";

        if(result == -1)
            cout << "not supported";
        else
        if(result == 1)
            cout << "supported";
        else
            cout << result;

        cout << endl << "explanation: " << endl;
        cout << tests[i].description << endl << endl;
    }


    return(EXIT_SUCCESS);
}
```

A.6.7 environment/sysvipclimits.c

```
////////////////////////////////////////////////////////////////////////////////
// Filename: sysvipclimits.c
////////////////////////////////////////////////////////////////////////////////
// Purpose: obtain the maximum nuber of System V IPC objects per process
////////////////////////////////////////////////////////////////////////////////
// History:
// ========
//
// Date     Time     Name      Description
// -------- -------- --------  ------------------------------------------------
// 96/03/18 04:42:38 muellerg: created
//
////////////////////////////////////////////////////////////////////////////////


// Feature test switches ///////////////////////////// Feature test switches //
    /* NONE */



// System headers /////////////////////////////////////////// System headers //

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <sys/sem.h>


// Local headers ///////////////////////////////////////////// Local headers //

#include "../common.h"



// Macros /////////////////////////////////////////////////////////// Macros //
    /* NONE */



// File scope objects /////////////////////////////////// File scope objects //

const int max_try = 1000;       // how many IPC objects at best?

const int SHM_MODE = (SHM_R | SHM_W);                   // read and write
const int MSG_MODE = 0666;      // MSG_R | MSG_W;       // read and write
const int SEM_MODE = 0666;      // SEM_R | SEM_A;       // read and alter

const int SHM_SIZE = 4096;      // size of shared memory segemnt

// External variables, functions, and classes ///////////// External objects //
    /* NONE */
```

```
// Signal catching functions //////////////////// Signal catching functions //
    /* NONE */



// Structures, unions, and class definitions ////////////////// Definitions //

struct sysvinfo
{
    int id;             // key for System V IPC object

    // this is only needed for shared memory:
    char *shmptr;       // pointer to shared memory
};



// Functions and class implementation /// Functions and class implementation //
    /* NONE */



// Main ////////////////////////////////////////////////////////////// Main //


int
main(int argc, char *argv[])
{
    error.set_program_name(argv[0]);

    cout << "This program tries to find out various things about System"
            " V IPC" << endl << endl;


    // display information from include files

    cout << "Informations from the include files:" << endl;


    /* shared memory */

    cout
         << "Shared memory:" << endl

#ifdef SHMMAX
         << "maximum shared segment size in bytes: " << SHMMAX << endl
#endif
#ifdef SHMMIN
         << "minimum shared segment size in bytes: " << SHMMIN << endl
#endif
#ifdef SHMMNI
         << "maximum number of shared memory segments system wide: "
         << SHMMNI << endl
#endif
#ifdef SHMALL
         << "maximum number of shared memory pages system wide: "
         << SHMALL << endl
#endif
```

```
#ifdef SHMSEG
        << "maximum number of shared segsments per process: "
        << SHMSEG << endl
#endif
        << endl


    /* message queues */

        << "Message queues:" << endl

#ifdef MSGMAX
        << "size in bytes of the largest message we can send: "
        << MSGMAX << endl
#endif
#ifdef MSGMNB
        << "maximum size in bytes of a particular queue: "
        << MSGMNB << endl
#endif
#ifdef MSGMNI
        << "maximum number of message queues, systemwide: "
        << MSGMNI << endl
#endif
#ifdef MSGTQL
        << "maximum number of messages, systemwide: "
        << MSGTQL << endl
#endif
        << endl


    /* semaphores */

        << "Semaphores:" << endl

#ifdef SEMVMX
        << "maximum value of any semaphore: " << SEMVMX << endl
#endif
#ifdef SEMAEM
        << "maximum value of any semaphore's adjust-on-exit value: "
        << SEMAEM << endl
#endif
#ifdef SEMMNI
        << "maximum number of semaphore sets, systemwide: "
        << SEMMNI << endl
#endif
#ifdef SEMMSL
        << "maximum number of semaphores per semaphore set: "
        << SEMMSL << endl
#endif
#ifdef SEMMNU
        << "maximum number of undo structures, systemwide: "
        << SEMMNU << endl
#endif
#ifdef SEMUME
        << "maximum number of undo entries per undo structure: "
        << SEMUME << endl
#endif
#ifdef SEMOPM
```

```
         << "maximum number of operations per semop call: "
         << SEMOPM << endl
#endif
         << endl;




    // try to get as much shared memory segments as possible

    sysvinfo *data;

    data = new sysvinfo[max_try];
    if(!data)
        error.panic("could not get memory!");

    bool ok = true;

    int i = 0;

    while(ok && i < max_try)
    {
        data[i].id = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE);

        if(data[i].id != -1)
        {
            data[i].shmptr = (char *)shmat(data[i].id, 0, 0);
            if(data[i].shmptr != (char *)-1)
            {
                /* everything is ok */

                i++;
            }
            else
            {
                // what is opposite of shmget?

                shmctl(data[i].id, IPC_RMID, 0);

                ok = false;
            }
        }
        else
            ok=false;
    }


    cout << "This process was able to get " << i << " shared memory"
            " segments\n" << endl;


    int max = i;

    // free segments

    for(i=0; i < max; i++)
    {
        if (shmdt(data[i].shmptr) == -1)
```

```
        error.system("shmdt error");

    if (shmctl(data[i].id, IPC_RMID, 0) == -1)
        error.system("shmctl error");
}



// create message queues


ok = true;

i = 0;

while(ok && i < max_try)
{
    data[i].id = msgget(IPC_PRIVATE, MSG_MODE);

    if(data[i].id != -1)
    {
        /* everything is ok */

        i++;
    }
    else
        ok = false;
}


cout << "This process was able to create " << i << " message queues\n"
     << endl;

max = i;

// free message queues

for(i=0; i < max; i++)
{
    // destroy message queue

    if(msgctl(data[i].id, IPC_RMID, NULL) == -1)
        error.system("msgctl error");
}



// create semaphore

ok = true;

i = 0;

while(ok && i < max_try)
{
    data[i].id = semget(IPC_PRIVATE, 1, SEM_MODE);
```

```
        if(data[i].id != -1)
        {
            /* everything is ok */

            i++;
        }
        else
            ok = false;
    }


    cout << "This process was able to create " << i << " semaphore sets\n"
         << endl;

    max = i;

    // free semaphore sets

    for(i=0; i < max; i++)
    {
        // destroy semaphore set

#ifdef linux
        union semun nop;
#else
        int nop = 0;
#endif

        if(semctl(data[i].id, 0, IPC_RMID, nop) == -1)
            error.system("semctl error");
    }

    return(EXIT_SUCCESS);
}
```

A.7 Remote Procedure Calls

The directory rpc contains a modified RPC example source code from [Stev90] that demonstrates how to use RPC.

File Name	Purpose	Page
Makefile	Makefile for this directory	306
date.x	Specification of remote date and time service	306
dateproc.c	remote procedures, called by server stub	307
rdate.c	client program for remote date service	307

Table 37: Purpose of Files in the rpc Directory

A.7.1 rpc/Makefile

```
BIN = date_svc rdate
GEN = date_clnt.c date_svc.c date.h
RPCCOM = rpcgen
CFLAGS = -I.. -O
CC = cc

all: $(BIN)

$(GEN): date.x
	$(RPCCOM) date.x

date_svc: dateproc.o date_svc.o
	$(CC) -o $@ dateproc.o date_svc.o $(LIB)

rdate: rdate.o date_clnt.o
	$(CC) -o $@ rdate.o date_clnt.o $(LIB)

rdate.o: rdate.c date.h

dateproc.o: dateproc.c date.h

clean:
	rm -f $(GEN) *.o *~

purge:
	rm -f $(BIN)
```

A.7.2 rpc/date.x

```
/*
 * date.x - Specification of remote date and time service.
 */
```

```
/*
 * Define 2 procedures:
 *  bin_date_1() returns the binary time and date (no arguments).
 *  str_date_1() takes a binary time and returns a human-readable string.
 */

program DATE_PROG {
    version DATE_VERS {
    long    BIN_DATE(void) = 1; /* procedure number = 1 */
    string  STR_DATE(long) = 2; /* procedure number = 2 */
    } = 1;              /* version number = 1 */
} = 0x31234567;             /* program number = 0x31234567 */
```

A.7.3 rpc/dateproc.c

```
/*
 * dateproc.c - remote procedures; called by server stub.
 */

#include    <rpc/rpc.h> /* standard RPC include file */
#include    <time.h>    /* for timing functions */

/*
 * Return the binary date and time.
 */

long *
bin_date_1_svc(void)
{
    static long timeval;    /* must be static */

    timeval = time((long *) 0);

    return(&timeval);
}

/*
 * Convert a binary time and return a human readable string.
 */

char **
str_date_1_svc(long *bintime)
{
    static char *ptr;       /* must be static */

    ptr = ctime(bintime);       /* convert to local time */

    return(&ptr);       /* return the address of pointer */
}
```

A.7.4 rpc/rdate.c

```
/*
 * rdate.c - client program for remote date service.
 */
```

```
#include    <stdio.h>
#include    <rpc/rpc.h> /* standard RPC include file */
#include    "date.h"    /* this file is generated by rpcgen */

main(argc, argv)
int argc;
char    *argv[];
{
    CLIENT      *cl;        /* RPC handle */
    char        *server;
    long        *lresult;   /* return value from bin_date_1() */
    char        **sresult;  /* return value from str_date_1() */

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }
    server = argv[1];

    /*
     * Create the client "handle."
     */

    if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
        /*
         * Couldn't establish connection with server.
         */

        clnt_pcreateerror(server);
        exit(2);
    }

    /*
     * First call the remote procedure "bin_date".
     */

    if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(3);
    }
    printf("time on host %s = %ld\n", server, *lresult);

    /*
     * Now call the remote procedure "str_date".
     */

    if ( (sresult = str_date_1(lresult, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(4);
    }
    printf("time on host %s = %s", server, *sresult);

    clnt_destroy(cl);       /* done with the handle */
    exit(0);
}
```

B Additional Information Resources

B.1 Internet Request For Comments

The Internet Request For Comments (or RFC) documents are the written definitions of the protocols and policies of the Internet.

The Web-page `http://www.cis.ohio-state.edu/hypertext/information/rfc.html` provides entry points to get at the RFCs.

As the time of writing (December 1995) RFC 1880 describes the state of standardization of protocols used in the Internet as determined by the Internet Architecture Board (IAB).

The University of Newcastle upon Tyne has an own RFC-index:
`http://net-www.ncl.ac.uk:80/general/rfc/index.html`

B.2 Periodicals

UNIX and networking is covered in the following **periodicals**:

- *Journal of Internetworking: Research and Experience*
  reports on internetworking research, with emphasis on experimental validation of ideas (source: [Com95, page 15])
- *ConneXions*
  contains articles about TCP/IP and the Internet as well as official statements of policy from the IAB (source: [Com95, page 15])
- *Computer Communication Review*
  is the SIGCOMM quarterly publication

B.3 Additional Web Pages

Here a collection of probably interesting **Web pages**:

- Available C++ libraries FAQ:
  `http://www.cis.ohio-state.edu/hypertext/faq/usenet/C++-faq/libraries/faq.html`

- Client Server Computing - Unix Sockets and RPC:
  `http://pandonia.canberra.edu.au/ClientServer/socket.html`
- UNIX Reference Desk:
  `http://www.eecs.nwu.edu/unix.html`
- Jumbo! - Programming: Unix:
  `http://www.jumbo.com/prog/unix/`
- RFC Index Search Form:
  `http://pubweb.nexor.co.uk/public/rfc/index/rfc.html`
- LINUX.ORG.UK:
  `http://www.linux.org.uk`
- Index - Well Known TCP/IP Ports - information about protocols and IP ports:
  `http://www.con.wesleyan.edu/~triemer/network/docservs.html`

Some networking & UNIX programming related **newsgroups**:

- comp.unix.programmer:
  `news:comp.unix.programmer`
- comp.protocols.tcp-ip:
  `news:comp.protocols.tcp-ip`
- comp.os.linux.networking:
  `news:comp.os.linux.networking`

Some **META-FAQ**s:

- USENET FAQs:
  `http://www.cis.ohio-state.edu/hypertext/faq/usenet/top.html`
- Computer and Communication Usenet related Newsgroups and FAQs:
  `http://www.cmpcmm.com/cc/usenet.html`

Some Pointers to **FAQ**s:

- TCP-IP FAQ:
  `ftp://ftp.netcom.com/ftp/pub/gnn/tcp-ip.faq`
- C++ FAQ:
  `http://www.cis.ohio-state.edu/hypertext/faq/usenet/C++-faq/top.html`

- comp.protocols.tcp-ip FAQ:
  `http://www.cis.ohio-state.edu/hypertext/faq/bngusenet/comp/protocols/tcp-ip/top.html`

Man-Pages:

- BSDI:
  `http://www.bsdi.com/bsdi-man/`
- HP-UX:
  `http://www.cis.ohio-state.edu/man/hpux.section_top.html`

C Glossary

Address family:
The address format used to interpret addresses specified in socket operations. Examples are the internet address family/domain (AF_INET) and the UNIX address family/domain (AF_UNIX).

Advanced Research Projects Agency:
A U.S. government research agency that was instrumental in developing and using the original ARPA Services on the ARPANET. Later it founded global Internet.

Alias:
A term used to refer to alternate names for networks, hosts and protocols. This is also an internetwork mailing term that refers an alternate name for a recipient or list of recipients (a mailing list).

ANSI:
American National Standards Institute. A group that defines U.S. standards for information processing industry including ASCII, FDDI, SGML, programming languages, SQL etc.

API:
Application Programming Interface

ARPA:
See *Advanced Research Projects Agency.*

ARPA/Berkeley Services:
The set of services originally developed for use on the ARPANET (i.e., `telnet`(1)) or distributed with the Berkeley Software Distribution of UNIX, version 4.2 (i.e., `rlogin`(1)).

ARPANET:
The Advanced Research Projects Agency Network, founded by ARPA. It served a long time as the basis for networking research and as a central backbone during development of Internet.

ASCII:
American Standard Code for Information Interchange.

Association:
An Interprocess Communication connection (e.g., a socket) is defined by an association. An association contains the (protocol, local address, local port, remote address, remote port)-tuple. Associations must be unique; duplicate associations on the same system may not exist.

Berkeley Software Distribution UNIX:
A version of UNIX software released by the University of California at Berkeley, or one of the commercial versions derived from it. BSD UNIX was the first UNIX which first included TCP/IP protocols.

Binding:
Establishing the address of a socket which allows other sockets to connect to it or to send data to it.

BSD UNIX:
See *Berkeley Software Distribution UNIX*.

Channel:
A communication path created by establishing a connection between sockets.

Client:
A process that is requesting some service from another process.

Client host:
The host on which a client process is running.

client/server:
A model of interaction in distributed systems in which a program on one site sends a request to a program (at probably another site) and waits for a response. The requesting program is called *client*, the program that responds to that request *server*.

Connection:
An abstraction that is provided by protocol software.

Daemon:
A software process that runs continuously and provides services on request. In the UNIX *client/server* model the *server* is normally a daemon.

DARPA:
See *Defense Advanced Research Projects Agency.*

Datagram sockets:
A socket that maintains record boundaries and treats data as individual messages rather than a stream of bytes. Messages may be sent to and received from many other datagram sockets. Datagram sockets do not support the concept of a connection. Messages could be lost or duplicated and may not arrive in the same sequence sent. Datagram sockets of the Internet domain use the User Datagram Protocol.

Defense Advanced Research Projects Agency:
The military arm of the Advanced Research Projects Agency. Former name of *ARPA*.

Domain:
A set of allowable names or values.

File Transfer Protocol:
The TCP/IP high-level file transfer protocol for transferring files from one machine to another. The `ftp` command uses the FTP protocol.

Free Software Foundation:
The Free Software Foundation is dedicated to eliminating restrictions on people's right to use, copy, modify, and redistribute computer programs. They do this by promoting the development and use of free software. Specifically, they are putting together a complete, integrated software system named "GNU" that will be upwardly compatible with UNIX. Most parts of this system are already being used and distributed.

FSF:
See *Free Software Foundation.*

FTP:
See *File Transfer Protocol.*

Gateway:

Former: a node that connects two or more networks together and routes packets between those networks. Now: application program that interconnects two or more services (e.g. e-mail gateway between two different net types).

GNU:

GNU, which stands for *Gnu's Not Unix*, is the name for the complete UNIX-compatible software system. See also *Free Software Foundation*.

GOSIP:

See *Government OSI Profile*.

Government OSI Profile:

Governments in many countries have adopted selected subsets of OSI, known as GOSIP, and have made them a requirement for interconnectivity and procurement activities.

Host:

A node that has primary functions other than switching data for the network (normally end-user computer system).

ICMP:

Internet Control Message Protocol, part of the Internet Protocol. The ICMP is used from gateways to hosts and between hosts to report errors and make routing suggestions.

IEEE:

Institute of Electrical and Electronics Engineers, Inc.

International Standards Organization:

International body that drafts, discusses and specifies standards for network protocols. Best known for its 7-layer reference model that describes the conceptual organisation of protocol. Although always used as reference model, it is seldom used in the commercial market.

internet:

Connection of two ore more networks.

Internet:

The collection of networks that connects a large number of networks. Uses the TCP/IP protocol to form a single virtual network. Grows with a phenomenal rate.

Internet address:

A four-byte quantity that is distinct from a link-level address and is the network address of a computer node. This address identifies both which network is on the Internet and which host is on the network.

Internet Protocol:

The TCP/IP standard protocol that defines the IP datagram as the basic unit of information passed across an internet. See [RFC 791] for exact specification.

Internetwork:

A term used to mean among different physical networks.

Interprocess Communication:

A facility that allows a process to communicate with another process on the same host or on a remote host.

IP:

See *Internet Protocol.*

IPng:

Internet Protocol - the Next Generation, a term applied to all activities related to the specification of the next version of IPv6. The current version is IPv4.

IPC:

See *Interprocess Communication.*

ISO:

See *International Standards Organization.*

Node:

A computer system that is attached to or is part of a computer network.

Open Software Foundation:
Consortium of computer companies that are attempting to define a common operating environment for workstations. OSF has the following goals: supply portable, scaleable, and interoperable open systems software technology. Important standards from OSF: OSF/1, DCE, MOTIF.

OSF:
See *Open Software Foundation.*

OSF DCE:
See *OSF Distributed Computing Environment.*

OSF Distributed Computing Environment:
OSF's Distributed Computing Environment provides services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment.

Packet:
A data unit that is transmitted between processes. Sometimes also called frame.

POSIX:
Portable Operating System Interface for uniX, a family of standards developed by the IEEE.

POSIX.1:
IEEE Std. 1003.1-1990 Standard for Information Technology Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API)

Protocol:
A set of conventions for transferring information between computers on a network (e.g., UDP or TCP).

Remote host:
A computer that is accessible through the network or via a gateway.

Remote Procedure Call:
A technology in which a program invokes services across a network by using modified

procedure calls, normally generated with a stub generator.

Request For Comment:
RFCs are documents that define the protocols used in the IP Internet. Some are only suggestions, some are even jokes, and others are published standards. Several sites in the Internet store RFCs and make them available for anonymous ftp.

RFC:
See *Request For Comment.*

RPC:
See *Remote Procedure Call.*

sequence number:
A unique number for every packet on a particular connection maintained by a reliable transport layer service. The sequence number allows the transport layer to see if any packets were lost or delivered out of sequence by the underlying network and data layers.

Server:
A process or host that performs operations that local or remote client hosts request.

Service:
A facility that uses Interprocess Communication to perform remote functions for a user (e.g., `rlogin`(1) or `telnet`(1)).

Socket:
Addressable entities that are at either end of an Interprocess Communication connection. A socket is identified by a socket descriptor. A program can write data to and read data from a socket, just as it writes and reads data to and from files.

Socket address:
The internet address, port address and address family of a socket. The port and internet address combination allows the network to locate a socket.

Stream socket:
A socket that, when connected to another stream socket, passes data as a byte stream

(with no record boundaries). Data is guaranteed to arrive in the sequence sent.

SVID:
See *System V Interface Description.*

System V Interface Description:
A formal document that exactly specifies which facilities a system has to have if it claims to be System V compatible.

TCP:
See *Transmission Control Protocol.*

TCP/IP:
See *TCP/IP Internet Protocol Suite.*

TCP/IP Internet Protocol Suite:
A collection of all TCP/IP protocols, often abbreviated with just TCP/IP, because they are the two fundamental ones.

TLI:
See *Transport Layer Interface.*

Transmission Control Protocol:
A connection-oriented protocol that guarantees that messages are delivered in the order in which they were sent and that all messages are delivered. If a TCP connection cannot deliver a message it closes the connection and informs the entity that created it. This protocol is layered on top of IP.

Transport Layer Interface:
A (not only) network interface defined for System V UNIX.

UDP:
See *User Datagram Protocol.*

Uniform Resource Locator:
A string that gives all information to access a piece of information: it contains the protocol type (e.g. ftp or http), and an identification of specific information (e.g. domain

or path names).

UNIX:
Operating System family. Today there are two major directions, BSD and System V, upon most UNIX flavours are based.UNIX is a trademark of UNIX System Laboratories, Inc.

UNIX Domain Address:
A character string containing the UNIX pathname to a UNIX Domain socket.

UNIX Domain Protocol:
A protocol providing fast communication between processes executing on the same host and using the AF_UNIX socket address family.

URL:
See *Uniform Resource Locator.*

User Datagram Protocol:
A connectionless protocol built on top of IP. It does not provide any guarantees on the ordering or delivery of messages. The important difference between UDP and IP is that UDP includes a protocol port number, allowing the sender to distinguish among multiple application programs on a given remote machine.

Virtual Terminal Protocol:
A protocol that provides terminal access to interactive services on remote hosts (e.g., `telnet`(1)).

World Wide Web:
A very large scale multi-medial information service that allows users to browse information using a WWW-browser like `Netscape` or `Mosaic`.

WWW:
See *World Wide Web.*

D Used UNIX System & Library Calls

The tables below give an overview of all used UNIX system & library calls. The second, third and fourth column indicate whether the function is conforming to 4.4BSD, SVID2 and POSIX.1 respectively. All used systems claim to be be POSIX.1 conforming, therefore they support all the POSIX.1 function calls, even if they are not marked as such. Note that some system calls are used in local and distributed IPC, these are not repeated in the distributed IPC table.

More information about the used system & library calls can be obtained via the '`man` *function*' command, or see page 311 for additional information.

Name	**BSD**	**SVID**	**POSIX**	**Purpose**
kill	•	•	•	send signal to a process
sigaction			•	set up a signal handler with diverse options
sigaddset			•	Adds a signal to asignal set
signal	•	•		set up a signal handler
sigprocmask			•	examines and changes blocked signals
sigsuspend			•	atomically swap out the signal mask while waiting for a signal

Table 38: UNIX Signal Handling Functions

Name	BSD	SVID	POSIX	Purpose
close	•	•	•	close a file descriptor
dup	•	•	•	create a copy of a file descriptor
dup2	•	•	•	create a copy of a file descriptor
mkfifo			•	create a FIFO (named pipe)
open	•	•	•	open and possibly create a file or device
pipe	•	•	•	create pipe (pair of file descriptors, one for reading, one for writing)
read	•	•	•	read from a file descriptor
socketpair	•			create a pair of connected sockets
unlink	•	•	•	delete a name and possibly the file it refers to
write	•	•	•	write to a file descriptor

Table 39: UNIX System Calls for Local IPC

Name	BSD	SVID	POSIX	Purpose
msgctl		•		diverse message control operations
msgget		•		get a message queue identifier
msgrcv		•		receive a message from a message queue
msgsnd		•		send a message to a message queue
semctl		•		diverse semaphore control operations
semget		•		get a semaphore set identifier
semop		•		semaphore operations
shmat		•		attaches a shared memory region to the data segment of the calling process
shmctl		•		diverse shared memory control operations
shmdt		•		detaches a shared memory region from the calling process's data segment
shmget		•		allocates a shared memory region

Table 40: UNIX System V IPC Specific System Calls

Name	BSD	SVID	POSIX	Purpose
accept	•			waits for a client to connect
bind	•			register name for a socket
connect	•			establish a connection with a server
gethostbyname	•			get network host entry
gethostname	•			get host name
getservbyname	•			get service entry
inet_addr	•			converts the Internet host address from numbers-and-dots notation into binary data in network byte order
inet_ntoa	•			converts the Internet host address given in network byte order to a string in standard numbers-and-dots notation
listen	•			used by connection-oriented server to indicate that it is willing to receive connections
recv	•			receive a message from a socket (only if in connected state)
recvfrom	•			receive a message from a socket
rresvport	•			
send	•			transmit a message to another socket (only if in connected state)
sendto	•			transmit a message to another socket
socket	•			create an endpoint for communication

Table 41: UNIX System & Library Calls for BSD's Socket API

Name	BSD	SVID	POSIX	Purpose
t_accept		•		accept a connect request
t_alloc		•		allocate a library structure
t_bind		•		bind an address to a transport endpoint
t_close		•		close a transport endpoint
t_listen		•		listen for a connect request
t_open		•		establish a transport endpoint
t_rcv		•		receive data or expedited data sent over a connection
t_rcvudata		•		receive a data unit
t_snd		•		send data or expedited data over a connection
t_sndudata		•		send a data unit

Table 42: UNIX Library Calls for the System V TLI

Name	BSD	SVID	POSIX	Purpose
sleep	•	•	•	sleep for specified number of seconds
gettimeofday	•			get system time in seconds and microseconds
time	•	•	•	returns the value of time in seconds since 00:00:00 UTC, January 1, 1970
times	•	•	•	get process and child process (user and system) time

Table 43: UNIX Timing System & Library Calls

Name	BSD	SVID	POSIX	Purpose
sprintf	•	•	•	formatted output in a string
strcat	•	•	•	appends a string to another string
strerror	•		•	return string describing error code
strncmp	•	•	•	compare two strings up to n characters
strtol	•	•	•	convert a string to a long integer
vsprintf	•	•	•	formatted output of a varargs argument list in a string

Table 44: UNIX String Library Calls

Name	BSD	SVID	POSIX	Purpose
exit	•	•	•	exit a process normally
fork	•	•	•	create an identical child process
getpid	•	•	•	get process identification of current process
getppid	•	•	•	get process identification of parent process
waitpid	•		•	wait for process termination

Table 45: UNIX Process Related Library and System Calls

Name	BSD	SVID	POSIX	Purpose
bcopy	•			copy byte strings
bzero	•			write zeros to a byte string
free	•	•	•	free dynamically allocated memory
malloc	•	•	•	allocate memory dynamically
memcpy	•	•		copy memory area
memset	•	•		fill memory with a constant byte
select	•			synchronous I/O multiplexing
sysconf	•		•	get configuration information at runtime
umask	•	•	•	set file creation mask
uname	•	•	•	get name and information about current kernel

Table 46: Diverse UNIX System and Library Calls

E About this Document

This document was mainly edited on a i586/90MHz. The used operating system was Linux 1.2.9. The following programs were used to produce this document:

- **Emacs 19.28** to write the LaTeX documentation and the program source code
- **LaTeX 2ε** and **dvipsk 5.58f** to generate the paper version of this document
- **LaTeX**2HTML translator Version 96.1 (Feb 5, 1996) Copyright ©1993, 1994, 1995, 1996, Nikos Drakos, Computer Based Learning Unit, University of Leeds, to generate the `.html`-version of this document
- **XFig 3.1** patchlevel 3.1N to draw all pictures
- **c++2html** (slightly modified) from Stefan Schwarz to convert the source code to `.html`
- **GNU gcc v2.7.0** to compile the programs on Linux
- **CC** of HP-UX to compile the programs on HP-UX
- **GNU gcc v2.7.2** to compile the programs on Solaris 2.5