Transport Interfaces Programming Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94043-1100
U.S.A.

Part No: 805–4041–10
October 1998

Contents

Preface

This manual describes the programmatic interfaces to transport services in the Solaris operating environment.

In this guide, the terms SunOS™ and Solaris™ are used interchangeably because the interfaces described in this manual are common to both. Solaris 7, the distributed computing operating environment for SunSoft™, is a superset of SunOS. It consists of SunOS release 5.7 with ONC+™, OpenWindows™, ToolTalk™, DeskSet™, OPEN LOOK, and other utilities. This release of Solaris is fully compatible with System V, Release 4 (SVR4) of UNIX® and conforms to the third edition of the System V Interface Description (SVID). It supports all System V network services.

Who Should Use This Book

The guide assists you in developing a networked, distributed application in the Solaris operating environment.

Use of this guide assumes basic competence in programming, a working familiarity with the C programming language, and a working familiarity with the UNIX operating system. Previous experience in network programming is helpful, but is not required to use this manual.

How This Book Is Organized

Chapter 1 gives a high-level introduction to networking concepts and the topics covered in this book.

Chapter 2 describes the socket interface at the transport layer.

Chapter 3 describes the X/Open Transport Interface (XTI) and UNIX System V Transport Layer Interface (TLI).

Chapter 4 describes the network selection mechanisms used by applications in selecting a network transport and its configuration.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

- For a list of documents and how to order them, see the catalog section of SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Related Books

The following online System AnswerBook™ products cover related network programming topics:

- *Solaris 7 Reference Manual Collection*
- *Solaris 7 Software Developer Collection*

The following third-party books are excellent sources on network programming topics:

- Stevens, Richard W.*UNIX Network Programming*. Prentice Hall Software Series, 1990.
- Rago, Stephen A. *System V Network Programming*. Addison-Wesley, 1993.
- Stevens, Richar W. *TCP/IP Illustrated, Volume I*. Addison-Wesley, 1994.
- Padovano, Michael. *Networking Applications on UNIX System V Release 4*. Prentice Hall, Inc., 1993
- Comer, Douglas E. and Srevens, David L. *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*, 2nd Edition. Prentice Hall, Inc., 1991.
- Comer, Douglas E. and Stevens, David L. *Internetworking with TCP/IP, Volume II: Design, Implementation, and Internals*. Prentice Hall, Inc., 1991.
- Comer, Douglas E. and Stenens, David L. *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications*, BSD Sockets Version. Prentice Hall, Inc., 1993.

- Comer, Douglas E. and Stevens, David L. *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications*, AT&T TLI Version. Prentice Hall, Inc., 1994.

What Typographic Changes and Symbols Mean

Table P–1 describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide.* These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

Table P–2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to Network Programming Interfaces

This chapter is a high-level introduction to this book. It is most helpful to those who are new to network programming and for those who would like a brief overview of network programming in the Solaris environment.

Note - Because this chapter briefly introduces these topics, you might find the reference books listed in "Related Books" on page viii to be helpful.

The Client-Server Model

The client-server model is a common method of implementing distributed applications. Figure 1–1 shows a typical networked environment where different services are provided and used by client and server processes.

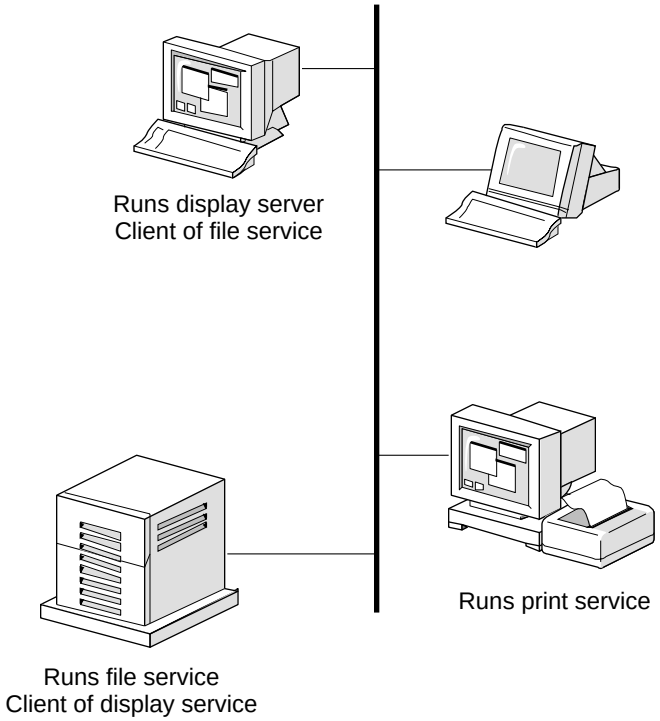

Figure 1–1 Client-Server Model

A *server* is a process that provides a service that can be used by other processes. Servers accept requests, perform their service, and return the results to the requester. Some examples of servers are:

- A file service such as the NFS™ file system, which provides access to files and directories to other processes or systems
- A display service, such as the X Window System™ environment, which provides access to a high resolution display device
- A time-of-day server that returns the current time whenever a client requests it

A server process normally listens at a well-known address for service requests. When a request is received, the server is unblocked and processes the client's request. Multiple servers can offer the same service, and they execute on the same machine or on multiple machines. It is common to replicate copies of a given server onto physically independent machines to increase reliability or improve performance. If a machine's primary purpose is to support a particular server program, the term "server" can be applied to the machine as well as to the server program. Thus, you hear statements such as "Mosey is our mail server."

A *client* is a process that makes use of a service, or services, provided by other processes and waits for a response. An individual system might be both a client and a server for different services, or even for the same service. For example, a print

server receives print requests from a client, but might need to issue a client request to a file server to access a file.

Network Services in the Solaris Environment

The Solaris environment provides a large number of networking services based upon the Internet protocol suite (also loosely referred to as the TCP/IP protocol suite, described on “TCP/IP Internet Protocol Suite” on page 7). These services are listed in Table 1–1.

TABLE 1–1 TCP/IP Services

Service	Service Description
ARP	Address Resolution Protocol. Used to obtain the hardware network address corresponding to an IP address.
DHCP	Dynamic Host Configuration Protocol. Allows a host to get an Internet Protocol (IP) address and other Internet configuration parameters without any need for preconfiguration by the user.
BOOTP	Boot Protocol. Allows diskless systems to boot from a remote server.
DNS	Domain Name System. Name service used by the Internet. Uses both TCP and UDP protocols.
FTP	File Transfer Protocol. Reliable file transfer. Allows interactive transfer of ASCII and binary files.
ICMP	Internet Control Message Protocol. Used to relay error and control information. Used by TCP for flow control.
IP	Internet Protocol. The core protocol of the TCP/IP protocol suite.
NTP	Network Time Protocol. Synchronizes the system clock of your host with the system clock of another computer or time source.
RARP	Reverse Address Resolution Protocol. Used primarily in diskless clients systems that have a hardware address but need to find out their IP address.
SMTP	Simple Mail Transfer Protocol. Electronic mail delivery protocol.

TABLE 1–1 TCP / IP Services *(continued)*

Service	Service Description
SNMP	Simple Network Management Protocol. Basis of many network management packages. Allows monitoring of activity throughout a network.
TCP	Transmission Control Protocol. Reliable connection-oriented byte stream transport.
TELNET	Terminal emulation. Enables login and interactive session on a remote system.
TFTP	Trivial File Transfer Protocol. Simpler but less secure version of FTP.
UDP	User Datagram Protocol. Unreliable connectionless datagram transport.

In addition to the base protocols and services, the protocol suite also provides some commonly used utility applications (such as `rcp`, `rsh`, and `rlogin`) built on top of the Internet protocol suite.

The Solaris computing environment also provides heterogeneous distributed computing facilities in its ONC+ architecture. The ONC+ architecture is a set of services built on top of Sun's remote procedure call (RPC) protocol. The programming interfaces available in the ONC+ platform are described in the *ONC+ Developer's Guide*.

Layered Protocols

A *protocol* is a set of rules and conventions that describe how information is to be exchanged between two entities. Networking tasks often require more than one protocol to perform a task, such as file transfer.

These protocols are often conceptualized in a model consisting of a series of layers, each of which deals with one functional aspect of the communication. Each layer has a well-defined interface to the layer immediately above and below it. The left side of Figure 1–2 shows that data is passed down through the interface to the layer below. Each layer adds the necessary information to the data so that the receiving system understands how to handle the data and is able to route the data. At the bottom layer on the sending side, the data is physically transmitted across some medium to the receiving system. It is passed up through the layers on the right side of Figure 1–2, with each layer removing the information added by the corresponding layers on

the sending system. A set of protocols layered in this way is called a *protocol stack*. A layer can have more than one protocol defined for it.

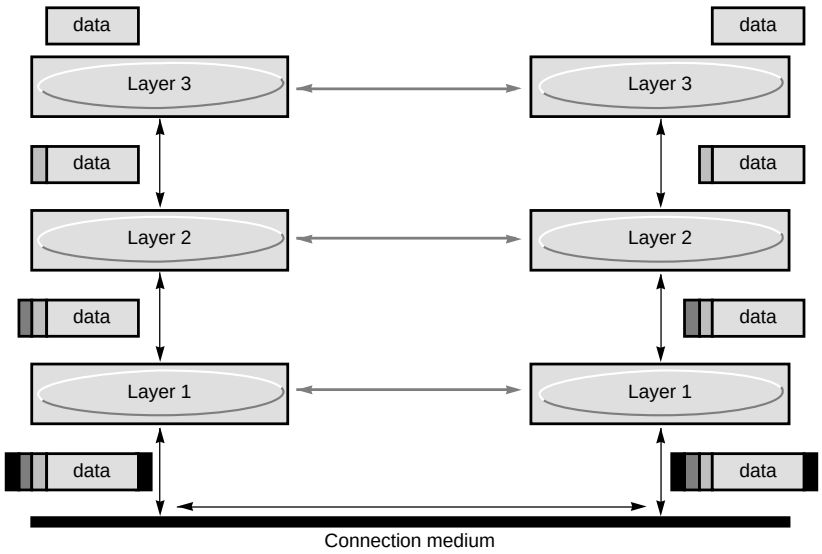

Figure 1–2 Layered Protocols

Two well-known reference models are discussed in the following sections: open systems interconnection (OSI) reference model and Internet (TCP/IP) protocol suite.

Open Systems Interconnection (OSI) Reference Model

The OSI reference model is used to conceptualize network service architectures and as a convenient framework for explaining networking concepts. It is not the basis for the Internet protocol suite, but the Internet protocol's four-layer model can be mapped to the more general OSI reference model. The OSI protocol suite follows the OSI reference model closely.

The OSI reference model divides networking functions into seven layers, as shown in Figure 1–3. Each protocol layer performs services for the layer above it. The ISO definition of the protocol layers gives designers considerable freedom in implementation. For example, some applications skip the presentation and session layers (layers 5 and 6) to interface directly with the transport layer. In this case, the application performs any needed presentation and session services.

Industry standards have been or are being defined for each layer of the reference model.

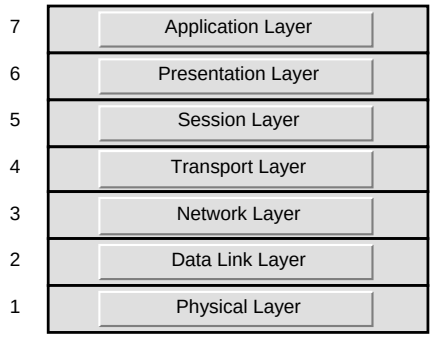

Figure 1–3 OSI Reference Model

OSI Reference Model Description

The following section explains each layer the OSI reference model.

Layer 1: Physical Layer

This layer specifies the physical media connecting hosts and networks, and the procedures used to transfer data between machines using a specified media. This layer is commonly referred to as the hardware layer of the model.

Layer 2: Data Link Layer

This layer manages the reliable delivery of data across the physical network. For example, it provides the abstraction of a reliable connection over the potentially unreliable physical layer.

Layer 3: Network Layer

This layer is responsible for routing machine-to-machine communications. It determines the path a transmission must take, based upon the destination machine's address. This layer must also respond to network congestion problems.

Layer 4: Transport Layer

This layer provides end-to-end sequenced delivery of data. It is the lowest layer that provides applications and higher layers with end-to-end service. This layer hides the topology and characteristics of the underlying network from users. It provides reliable end-to-end data delivery if the service characteristics require it.

Layer 5: Session Layer

This layer manages sessions between cooperating applications.

Layer 6: Presentation Layer

This layer performs the translation between the data representation local to the computer and the processor-independent format that is sent across the network. It can also negotiate the transfer formats in some protocol suites. Typical examples include standard routines that compress text or convert graphic images into bit streams for transmission across a network.

Layer 7: Application Layer

This layer consists of the user-level programs and network services. Some examples are `telnet`, `ftp`, and `tftp`.

TCP/IP Internet Protocol Suite

TCP/IP is a widely used protocol suite for internetworking, a term that refers to the connection of various physical networks to form one large virtual network. Any system connected to a TCP/IP internetwork should be able to communicate with any other system within the internetwork, regardless of the physical network on which the systems actually reside. Networks are linked together by a system that functions as a gateway between systems.

While TCP/IP has a closely associated history with UNIX systems, the TCP/IP protocols themselves are independent of the operating system, the network topology, and the connection medium. TCP/IP operates on Ethernet and Token Ring local area networks (LANs), across wide area links such as X.25, and serial connections. Support for TCP/IP networking has been an integral part of SunOS in all versions of the operating system.

TCP/IP Protocol Stack

The TCP/IP protocol suite can be described using a reference model similar to the OSI reference model. Figure 1–4 shows the corresponding OSI layers and some example services at each layer. TCP/IP does not delineate the presentation and session layers as the OSI model does; application code provides the necessary presentation or session functionality.

The TCP/IP protocols are defined in documents called *Requests for Comments* (RFCs). RFCs are maintained by the Network Information Center (NIC), the organization that handles address registration for the Internet.

RFCs define a number of applications, the most widely used being `telnet`, a terminal emulation service on remote hosts, and `ftp`, which allows files to be transferred between systems.

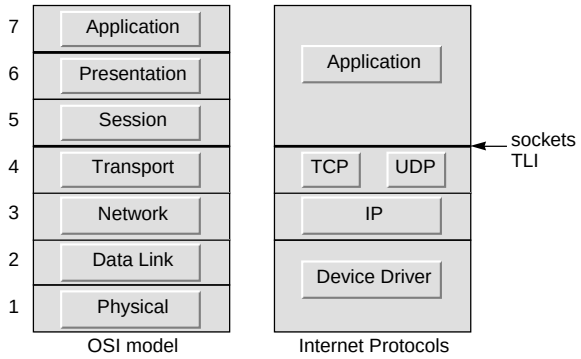

Figure 1–4 TCP/IP Protocol Stack

TCP/IP Protocol Stack Description

The following sections describes the parts of the TCP/IP protocol stack.

Device Drivers

The device driver layer (also called the Network Interface) is the lowest TCP/IP layer and is responsible for accepting packets and transmitting them over a specific network. A network interface might consist of a device driver or a complex subsystem that uses its own data link protocol.

Internet Protocol (IP) Layer

The Internet Protocol layer handles communication from one machine to another. It accepts requests to send data from the transport layer along with an identification of the machine to which the data is to be sent. It encapsulates the data into an IP datagram, fills in the datagram header, uses the routing algorithm to determine how to deliver the datagram, and passes the datagram to the appropriate device driver for transmission.

The IP layer corresponds to the network layer in the OSI reference model. IP provides a connectionless, “unreliable” packet-forwarding service that routes packets from one system to another.

Transport Layer

The primary purpose of the transport layer is to provide communication from one application program to another. The transport software divides the stream of data being transmitted into smaller pieces called packets in the ISO terminology and passes each packet along with the destination information to the next layer for transmission.

This layer consists of Transport Control Protocol (TCP), a connection-oriented transport service (COTS), and the user datagram protocol (UDP), a connectionless transport service (CLTS).

Application Layer

The application layer consists of user-invoked application programs that access services available across a TCP/IP Internet. The application program passes data in the required form to the transport layer for delivery.

Connection-Oriented and Connectionless Protocols

A number of characteristics can be used to describe communications protocols. The most important is the distinction between *connection-oriented transport* services (COTS) and *connectionless transport* services (CLTS).

Connection-Oriented Protocols

TCP is an example of a connection-oriented protocol. It requires a logical connection to be established between the two processes before data is exchanged. The connection must be maintained during the entire time that communication is taking place, then released afterwards. The process is much like a telephone call, where a virtual circuit is established—the caller must know the person's telephone number and the phone must be answered—before the message can be delivered.

TCP/IP is also a connection-oriented transport with orderly release. With orderly release, any data remaining in the buffer is sent before the connection is terminated. The release is accomplished in a three-way handshake between client and server processes. The connection-oriented protocols in the OSI protocol suite, on the other hand, do not support orderly release. Applications perform any handshake necessary for ensuring orderly release.

Examples of services that use connection-oriented transport services are `telnet`, `rlogin`, and `ftp`.

Connectionless Protocols

Connectionless protocols, in contrast, allow data to be exchanged without setting up a link between processes. Each unit of data, with all the necessary information to route it to the intended destination, is transferred independent of other data packets and can travel over different paths to reach the final destination. Some data packets might be lost in transmission or might arrive out of sequence to other data packets.

UDP is a connectionless protocol. It is known as a datagram protocol because it is analogous to sending a letter where you don't acknowledge receipt.

Examples of applications that use connectionless transport services are broadcasting and `tftp`. Early implementations of NFS used UDP, whereas newer implementations prefer to use TCP.

Choosing Between COTS and CLTS

The application developer must decide which type of protocol works best for the particular application. Some questions to ask are:

- How reliable must the connection be?
- Must the data arrive in the same order as it was sent?
- Must the connection be able to handle duplicate data packets?
- Must the connection have flow control?
- Must the connection acknowledge the messages it receives?
- What kind of service can the application live with?
- What level of performance is required?

If reliability is paramount, then connection-oriented transport services (COTS) is the better choice.

CHAPTER 2

Programming With Sockets

This chapter presents the socket interface and illustrates it with sample programs. The programs demonstrate the Internet domain sockets.

Sockets are Multithread Safe

The interface described in this chapter is multithread safe. Applications that contain socket function calls can be used freely in a multithreaded application. Note, however, that the degree of concurrency available to applications is not specified.

SunOS 4 Binary Compatibility

Two major changes from SunOS 4 hold true for SunOS 5 releases. The binary compatibility package allows SunOS 4–based dynamically linked socket applications to run on SunOS 5.

1. You must explicitly specify the socket library (`-lsocket` or `libsocket`) on the compilation line.

2. You may need to link with `libnsl` as well (use `-lsocket -lnsl`, not `-lnsl -lsocket`).
3. You must recompile all SunOS 4 socket-based applications with the socket library to run under SunOS 5.

What Are Sockets?

Sockets are the Berkeley UNIX interface to network protocols. They have been an integral part of SunOS releases since 1981. They are commonly referred to as Berkeley sockets or BSD sockets. Beginning in Solaris 7, the XNS 5 (Unix98) Socket interfaces (which differ slightly from the BSD sockets) are also available.

The XNS 5 (Unix98) Socket interfaces are documented in the following man pages: **accept**(3XN), **bind**(3XN), **connect**(3XN), **endhostent**(3XN), **endnetent**(3XN), **endprotoent**(3XN), **endservent**(3XN), **gethostbyaddr**(3XN), **gethostbyname**(3XN), **gethostent**(3XN), **gethostname**(3XN), **getnetbyaddr**(3XN), **getnetbyname**(3XN), **getnetent**(3XN), **getpeername**(3XN), **getprotobyname**(3XN), **getprotobynumber**(3XN), **getprotoent**(3XN), **getservbyname**(3XN), **getservbyport**(3XN), **getservent**(3XN), **getsockname**(3XN), **getsockopt**(3XN), **htonl**(3XN), **htons**(3XN), **inet_addr**(3XN), **inet_lnaof**(3XN), **inet_makeaddr**(3XN), **inet_netof**(3XN), **inet_network**(3XN), **inet_ntoa**(3XN), **listen**(3XN), **ntohl**(3XN), **ntohs**(3XN), **recv**(3XN), **recvfrom**(3XN), **recvmsg**(3XN), **send**(3XN), **sendmsg**(3XN), **sendto**(3XN), **sethostent**(3XN), **setnetent**(3XN), **setprotoent**(3XN), **setservent**(3XN), **setsockopt**(3XN), **shutdown**(3XN), **socket**(3XN), and **socketpair**(3XN). The traditional SunOS 5 BSD Socket behaviour is documented in the corresponding 3N man pages. See the **standards**(5) man page for information on building applications that use the XNS 5 (Unix98) socket interface.

Since the days of early UNIX, applications have used the file system model of input/output to access devices and files. The file system model is sometimes called *open-close-read-write* after the basic function calls used in this model. However, the interaction between user processes and network protocols are more complex than the interaction between user processes and I/O devices.

A socket is an endpoint of communication to which a name can be bound. A socket has a *type* and one associated process. Sockets were designed to implement the client-server model for interprocess communication where:

- The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, Xerox internet protocols (XNS), and UNIX domain.
- The interface to network protocols needs to accommodate server code that waits for connections and client code that initiates connections.

- It also needs to operate differently, depending on whether communication is connection-oriented or connectionless.
- Application programs might want to specify the destination address of the datagrams it delivers instead of binding the address with the `open( )` call.

To address these issues and others, sockets are designed to accommodate network protocols, while still behaving like UNIX files or devices whenever it makes sense. Applications create sockets when they are needed. Sockets work with the `open( )`, `close( )`, `read( )`, and `write( )` function calls, and the operating system can differentiate between the file descriptors for files, and file descriptors for sockets.

UNIX domain sockets are named with UNIX paths. For example, a socket might be named `/tmp/foo`. UNIX domain sockets communicate only between processes on a single host. Sockets in the UNIX domain are not considered part of the network protocols because they can only be used to communicate with processes within the same UNIX system. They are rarely used today and are only briefly covered in this manual.

Socket Libraries

The socket interface routines are in a library that must be linked with the application. The libraries `libsocket.so` and `libsocket.a` are contained in `/usr/lib` with the rest of the system service libraries. The difference is that `libsocket.so` is used for dynamic linking, whereas `libsocket.a` is used for static linking.

Note - Static linking is *strongly* discouraged.

Socket Types

Socket types define the communication properties visible to a user. The Internet domain sockets provide access to the TCP/IP transport protocols. The Internet domain is identified by the value `AF_INET`. Sockets exchange data only with sockets in the same domain.

Three types of sockets are supported:

1. Stream sockets allow processes to communicate using TCP. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is `SOCK_STREAM`.
2. Datagram sockets allow processes to use UDP to communicate. A datagram socket supports bidirectional flow of messages. A process on a datagram socket can receive messages in a different order from the sending sequence and can

receive duplicate messages. Record boundaries in the data are preserved. The socket type is SOCK_DGRAM.

3. Raw sockets provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is SOCK_RAW.

See "Selecting Specific Protocols" on page 45 for further information.

Socket Tutorial

This section covers the basic methodologies of using sockets.

Socket Creation

The socket( ) call creates a socket in the specified domain and of the specified type.

```
s = socket(domain, type, protocol);
```

If the protocol is unspecified (a value of 0), the system selects a protocol that supports the requested socket type. The socket handle (a file descriptor) is returned.

The domain is specified by one of the constants defined in sys/socket.h. Constants named AF_*suite* specify the address format to use in interpreting names as shown in Table 2–1.

TABLE 2–1 Protocol Family

AF_APPLETALK	Apple Computer Inc. Appletalk network
AF_INET	Internet domain
AF_PUP	Xerox Corporation PUP internet
AF_UNIX	Unix file system

Socket types are defined in `sys/socket.h`. These types—`SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`—are supported by `AF_INET` and `AF_UNIX`. The following creates a stream socket in the Internet domain:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket with the TCP protocol providing the underlying communication. The following creates a datagram socket for intramachine use:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Use the default protocol (the *protocol* argument is `0`) in most situations. You can specify a protocol other than the default, as described in “Advanced Topics” on page 39.

Binding Local Names

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to it. Communicating processes are connected through addresses. In the Internet domain, a connection is composed of local and remote addresses, and local and remote ports. In the UNIX domain, a connection is composed of (usually) one or two path names. In most domains, connections must be unique.

In the Internet domain, there can never be duplicate ordered sets, such as: `protocol`, `local address`, `local port`, `foreign address`, `foreign port`. UNIX domain sockets need not always be bound to a name, but, when bound, there can never be duplicate ordered sets such as: `local pathname` or `foreign pathname`. The path names cannot refer to existing files.

The `bind( )` call allows a process to specify the local address of the socket. This forms the set `local address`, `local port` (or `local pathname`) while `connect( )` and `accept( )` complete a socket’s association by fixing the remote half of the address tuple. The `bind( )` function call is used as follows:

```
bind (s, name, namelen);
```

The socket handle is *s*. The bound name is a byte string that is interpreted by the supporting protocol(s). Internet domain names contain an Internet address and port number. UNIX domain names contain a path name and a family. Code Example 2–1 shows binding the name `/tmp/foo` to a UNIX domain socket.

CODE EXAMPLE 2–1 Bind Name to Socket

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
  strlen(addr.sun_path) + sizeof (addr.sun_family));
```

When determining the size of an AF_UNIX socket address, null bytes are not counted, which is why `strlen()` use is fine.

The file name referred to in `addr.sun_path` is created as a socket in the system file name space. The caller must have write permission in the directory where `addr.sun_path` is created. The file should be deleted by the caller when it is no longer needed. AF_UNIX sockets can be deleted with `unlink()`.

Binding an Internet address is more complicated but the call is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
  s = socket(AF_INET, SOCK_STREAM, 0);
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = htonl(INADDR_ANY);
  sin.sin_port = htons(MYPORT);
  bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The content of the address `sin` is described in “Address Binding” on page 46, where Internet address bindings are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server’s socket. On the client side, the `connect()` call initiates a connection. In the UNIX domain, this might appear as:

```
struct sockaddr_un server;
server.sun.family = AF_UNIX;
 ...
connect(s, (struct sockaddr *)&server,
    strlen(server.sun_path) + sizeof (server.sun_family));
```

In the Internet domain it might appear as:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. See "Signals and Process Group ID" on page 44. This is the usual way that local addresses are bound to a socket on the client side.

In the examples that follow, only `AF_INET` sockets are described.

To receive a client's connection, a server must perform two steps after binding its socket. The first is to indicate how many connection requests can be queued. The second step is to accept a connection:

```
struct sockaddr_in from;
 ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

The socket handle *s* is the socket bound to the address to which the connection request is sent. The second parameter of `listen()` specifies the maximum number of outstanding connections that might be queued. `from` is a structure that is filled with the address of the client. A `NULL` pointer might be passed. *fromlen* is the length of the structure. (In the UNIX domain, `from` is declared a `struct sockaddr_un`.)

`accept()` normally blocks. `accept()` returns a new socket descriptor that is connected to the requesting client. The value of *fromlen* is changed to the actual size of the address.

A server cannot indicate that it accepts connections only from specific addresses. The server can check the from address returned by `accept()` and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the accept call. These techniques are presented in "Advanced Topics" on page 39.

Connection Errors

An error is returned if the connection is unsuccessful (however, an address bound by the system remains). Otherwise, the socket is associated with the server and data transfer can begin.

Table 2–2 lists some of the more common errors returned when a connection attempt fails.

TABLE 2–2 Socket Connection Errors

Socket Errors	Error Description
ENOBUFS	Lack of memory available to support the call.
EPROTONOSUPPORT	Request for an unknown protocol.
EPROTOTYPE	Request for an unsupported type of socket.
ETIMEDOUT	No connection established in specified time. This happens when the destination host is down or when problems in the network result in lost transmissions.
ECONNREFUSED	The host refused service. This happens when a server process is not present at the requested address.
ENETDOWN or EHOSTDOWN	These errors are caused by status information delivered by the underlying communication interface.
ENETUNREACH or EHOSTUNREACH	These operational errors can occur either because there is no route to the network or host, or because of status information returned by intermediate gateways or switching nodes. The status returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the functions to send and receive data. You can send or receive a message with the normal `read()` and `write()` function calls:

```
write(s, buf, sizeof buf);
```

(continued)

(Continuation)

```
read(s,  buf, sizeof buf);
```

Or the calls `send()` and `recv()` can be used:

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send()` and `recv()` are very similar to `read()` and `write()`, but the `flags` argument is important. The flags, defined in `sys/socket.h`, can be specified as a nonzero value if one or more of the following is required:

`MSG_OOB`	Send and receive out-of-band data
`MSG_PEEK`	Look at data without reading
`MSG_DONTROUTE`	Send data without routing packets

Out-of-band data is specific to stream sockets. When `MSG_PEEK` is specified with a `recv()` call, any data present is returned to the user but treated as still unread. The next `read()` or `recv()` call on the socket returns the same data. The option to send data without routing packets applied to the outgoing packets is currently used only by the routing table management process and is unlikely to be interesting to most users.

Closing Sockets

A `SOCK_STREAM` socket can be discarded by a `close()` function call. If data is queued to a socket that promises reliable delivery after a `close()`, the protocol continues to try to transfer the data. If the data is still undelivered after an arbitrary period, it is discarded.

A `shutdown()` closes `SOCK_STREAM` sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

Where `how` is defined as:

0 Disallows further receives

1 Disallows further sends

2 Disallows both further sends and receives

Connecting Stream Sockets

Figure 2–1 and the next two examples illustrate initiating and accepting an Internet domain stream connection.

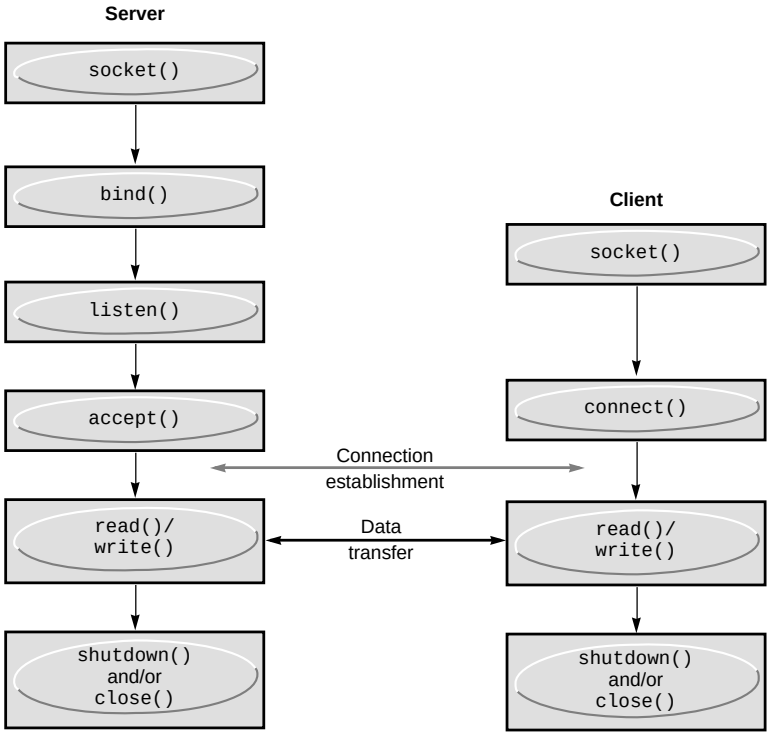

Figure 2–1 Connection-Oriented Communication Using Stream Sockets

The program in Code Example 2–2 is a server. It creates a socket and binds a name to it, then displays the port number. The program calls `listen()` to mark the socket ready to accept connection requests and initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes it. The use of `INADDR_ANY` is explained in "Address Binding" on page 46.

CODE EXAMPLE 2–2 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
* This program creates a socket and then begins an infinite loop.
* Each time through the loop it accepts a connection and prints
* data from it. When the connection breaks, or the client closes
* the connection, the program accepts a new connection.
*/

main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;

   /* Create socket. */
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock == –1) {
      perror("opening stream socket");
      exit(1);
   }
   /* Bind socket using wildcards.*/
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = 0;
   if (bind(sock, (struct sockaddr *) &server, sizeof server)
         == –1) {
      perror("binding stream socket");
      exit(1);
   }
   /* Find out assigned port number and print it out. */
   length = sizeof server;
   if (getsockname(sock,(struct sockaddr *) &server,&length)
         == –1) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(server.sin_port));
   /* Start accepting connections. */
   listen(sock, 5);
   do {
      msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
      if (msgsock == –1
         perror("accept");
      else do {
         memset(buf, 0, sizeof buf);
         if ((rval = read(msgsock,buf, 1024)) == –1)
```

(continued)

(Continuation)

```
            perror("reading stream message");
         if (rval == 0)
            printf("Ending connection\n");
         else
            /* assumes the data is printable */
            printf("-->%s\n", buf);
      } while (rval > 0);
      close(msgsock);
   } while(TRUE);
   /*
    * Since this program has an infinite loop, the socket "sock" is
    * never explicitly closed. However, all sockets will be closed
    * automatically when a process is killed or terminates normally.
    */
   exit(0);
}
```

To initiate a connection, the client program in Code Example 2–3 creates a stream socket and calls `connect( )`, specifying the address of the socket for connection. If the target socket exists and the request is accepted, the connection is complete and the program can send data. Data are delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines, such as `ntohl( )`, `ntohs( )`, `htons( )`, and `htonl( )`, in this program, see the **`byteorder`**(3N) man page.

CODE EXAMPLE 2–3 Internet Domain Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(argc, argv)
  int argc;
  char *argv[];
{
  int sock;
   struct sockaddr_in server;
```

(continued)

(Continuation)

```
   struct hostent *hp, *gethostbyname();
   char buf[1024];

   /* Create socket. */
   sock = socket( AF_INET, SOCK_STREAM, 0);
   if (sock == –1) {
    perror("opening stream socket");
    exit(1);
   }
   /* Connect socket using name specified by command line. */
   server.sin_family = AF_INET;
   hp = gethostbyname(argv[1] );
/*
 * gethostbyname returns a structure including the network address
 * of the specified host.
 */
  if (hp == (struct hostent *) 0) {
   fprintf(stderr, "%s: unknown host\n", argv[1]);
   exit(2);
  }
  memcpy((char *) &server.sin_addr, (char *) hp->h_addr,
   hp->h_length);
  server.sin_port = htons(atoi(argv[2]));
  if (connect(sock, (struct sockaddr *) &server, sizeof server)
    == –1) {
   perror("connecting stream socket");
   exit(1);
  }
  if (write( sock, DATA, sizeof DATA) == –1)
   perror("writing on stream socket");
  close(sock);
  exit(0);
}
```

Datagram Sockets

A datagram socket provides a symmetric data exchange interface. There is no requirement for connection establishment. Each message carries the destination address. Figure 2–2 shows the flow of communication between server and client.

Note - The `bind()` step shown below for the server is optional.

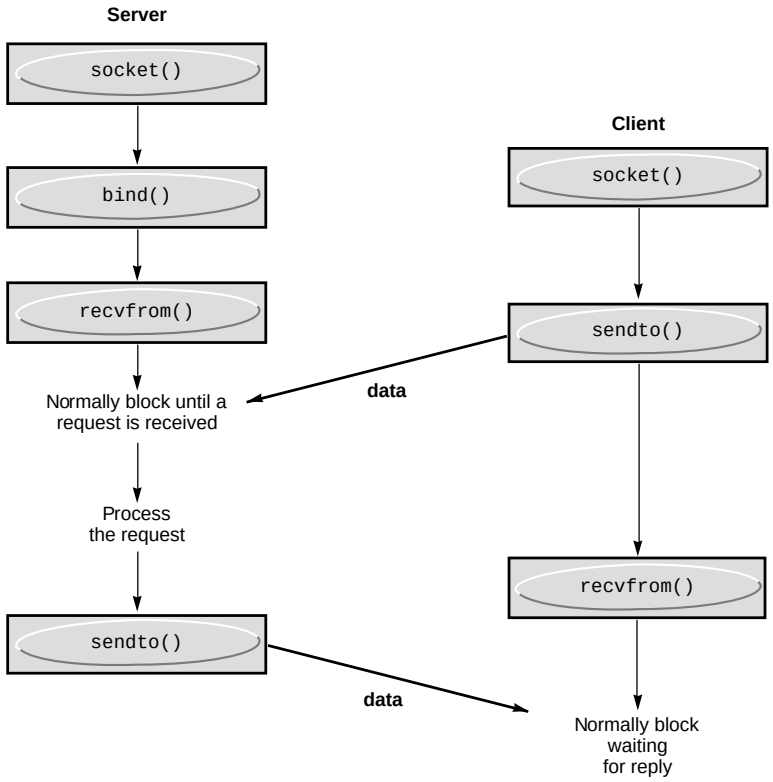

Figure 2–2 Connectionless Communication Using Datagram Sockets

Datagram sockets are created as described in "Socket Creation" on page 14. If a particular local address is needed, the `bind()` operation must precede the first data transmission. Otherwise, the system sets the local address and/or port when data is first sent. To send data, the `sendto()` call is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are the same as in connection-oriented sockets. The *to* and *tolen* values indicate the address of the intended recipient of the message. A locally detected error condition (such as an unreachable network) causes a return of `–1` and *errno* to be set to the error number.

To receive messages on a datagram socket, the `recvfrom()` call is used:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

Before the call, *fromlen* is set to the size of the *from* buffer. On return, it is set to the size of the address from which the datagram was received.

Datagram sockets can also use the `connect()` call to associate a socket with a specific destination address. It can then use the `send()` call. Any data sent on the socket without explicitly specifying a destination address is addressed to the connected peer, and only data received from that peer is delivered. Only one connected address is permitted for one socket at a time. A second `connect()` call changes the destination address. Connect requests on datagram sockets return immediately. The system records the peer's address. `accept()`, and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from previous `send()` calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket, or an option of `getsockopt()`, `SO_ERROR`, can be used to interrogate the error status.

Code Example 2–4 shows how to send an Internet call by creating a socket, binding a name to the socket, and sending the message to the socket.

CODE EXAMPLE 2–4 Sending an Internet Domain Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_in name;
   struct hostent *hp, *gethostbyname();

   /* Create socket on which to send. */
   sock = socket(AF_INET,SOCK_DGRAM, 0);
   if (sock == –1) {
      perror("opening datagram socket");
      exit(1);
   }
   /*
    * Construct name, with no wildcards, of the socket to ‘‘send’’
    * to. gethostbyname returns a structure including the network
    * address of the specified host. The port number is taken from
    * the command line.
    */
   hp = gethostbyname(argv[1]);
   if (hp == (struct hostent *) 0) {
```

(continued)

(Continuation)

```
      fprintf(stderr, "%s: unknown host\n", argv[1]);
      exit(2);
   }
   memcpy((char *) &name.sin_addr, (char *) hp->h_addr,
      hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));
   /* Send message. */
   if (sendto(sock,DATA, sizeof DATA ,0,
      (struct sockaddr *) &name,sizeof name) == -1)
      perror("sending datagram message");
   close(sock);
   exit(0);
}
```

Code Example 2–5 shows how to read an Internet call by creating a socket, binding a name to the socket, and then reading from the socket.

CODE EXAMPLE 2–5 Reading Internet Domain Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
* The include file <netinet/in.h> defines sockaddr_in as:
 * struct sockaddr_in {
 *  short   sin_family;
 *  u_short   sin_port;
 *  struct   in_addr sin_addr;
 *  char   sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */

main()
{
   int sock, length;
   struct sockaddr_in name;
   char buf[1024];

   /* Create socket from which to read. */
   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock == -1) {
      perror("opening datagram socket");
      exit(1);
   }
   /* Create name with wildcards. */
```

(continued)

(Continuation)

```
   name.sin_family = AF_INET;
   name.sin_addr.s_addr = INADDR_ANY;
   name.sin_port = 0;
   if (bind(sock,(struct sockaddr *)&name, sizeof name) == –1) {
      perror("binding datagram socket");
      exit(1);
   }
   /* Find assigned port value and print it out. */
   length = sizeof(name);
   if (getsockname(sock,(struct sockaddr *) &name, &length)
         == –1)  {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(name.sin_port));
   /* Read from the socket. */
   if (read(sock, buf, 1024) == –1 )
      perror("receiving datagram packet");
   /* Assumes the data is printable */
   printf("-->%s\n", buf);
   close(sock);
   exit(0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or files. Use the `select()` call to do this:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of `select()` is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of `select()` are pointers to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. Any of these pointers can be a properly cast null. Each set is a structure containing an array of long integer bit masks. The size of the array is set by

FD_SETSIZE (defined in select.h). The array is long enough to hold one bit for each FD_SETSIZE file descriptor.

The macros FD_SET(*fd*, &*mask*) and FD_CLR(*fd*, &*mask*) add and delete, respectively, the file descriptor *fd* in the set mask. The set should be zeroed before use, and the macro FD_ZERO(&*mask*) clears the set mask.

The fifth argument of select( ) allows a time-out value to be specified. If the timeout pointer is NULL, select( ) blocks until a descriptor is selectable, or until a signal is received. If the fields in timeout are set to 0, select( ) polls and returns immediately.

select( ) normally returns the number of file descriptors selected. select( ) returns a 0 if the time-out has expired. select( ) returns –1 for an error or interrupt with the error number in *errno* and the file descriptor masks unchanged. For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

You should test the status of a file descriptor in a select mask with the FD_ISSET(*fd*, &*mask*) macro. It returns a nonzero value if *fd* is in the set mask, and 0 if it is not. Use select( ) followed by a FD_ISSET(*fd*, &*mask*) macro on the read set to check for queued connect requests on a socket.

Code Example 2–6 shows how to select on a “listening” socket for readability to determine when a new connection can be picked up with a call to accept( ). The program accepts connection requests, reads data, and disconnects on a single socket.

CODE EXAMPLE 2–6 Using select( ) to Check for Pending Connections

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */

main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;
   fd_set ready;
   struct timeval to;
```

(continued)

(Continuation)

```
   /* Open a socket and bind it as in previous examples. */

   /* Start accepting connections. */
   listen(sock, 5);
   do {
      FD_ZERO(&ready);
      FD_SET(sock, &ready);
      to.tv_sec = 5;
      to.tv_usec = 0;
      if (select(sock + 1, &ready, (fd_set *)0, (fd_set *)0, &to) == –1) {
         perror("select");
         continue;
      }
      if (FD_ISSET(sock, &ready)) {
         msgsock = accept(sock, (struct sockaddr *)0,
            (int *)0);
         if (msgsock == –1)
            perror("accept");
         else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock, buf, 1024)) == –1)
               perror("reading stream message");
            else if (rval == 0)
               printf("Ending connection\n");
            else
               printf("-->%s\n", buf);
         } while (rval > 0);
         close(msgsock);
      } else
         printf("Do something else\n");
   } while (TRUE);
    exit(0);
}
```

In previous versions of the `select()` routine, its arguments were pointers to integers instead of pointers to `fd_sets`. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

`select()` provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals described in “Advanced Topics” on page 39 provide asynchronous notification of output completion, input availability, and exceptional conditions.

Standard Routines

You might need to locate and construct network addresses. This section describes the routines that manipulate network addresses. Unless otherwise stated, functions presented in this section apply only to the Internet domain.

Locating a service on a remote host requires many levels of mapping before client and server communicate. A service has a name for human use. The service and host names must be translated to network addresses. Finally, the address is used to locate and route to the host. The specifics of the mappings can vary between network architectures. Preferably, a network does not require that hosts be named, thus protecting the identity of their physical locations. It is more flexible to discover the location of the host when it is addressed.

Standard routines map host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers, and the appropriate protocol to use in communicating with the server process. The file `netdb.h` must be included when using any of these routines.

Host Names

An Internet host-name-to-address mapping is represented by the `hostent` structure:

```
struct hostent {
    char  *h_name;            /* official name of host */
    char  **h_aliases;        /* alias list */
    int   h_addrtype;         /* hostaddrtype(e.g.,AF_INET) */
    int   h_length;           /* length of address */
    char  **h_addr_list;      /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

`gethostbyname()` maps an Internet host name to a `hostent` structure, `gethostbyaddr()` maps an Internet host address to a `hostent` structure, and `inet_ntoa()` maps an Internet host address to a displayable string.

The routines return a `hostent` structure containing the name of the host, its aliases, the address type (address family), and a `NULL`-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The `h_addr` definition is for backward compatibility, and is the first address in the list of addresses in the `hostent` structure.

Network Names

The routines to map network names to numbers, and back return a `netent` structure:

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
   char     *n_name;      /* official name of net */
   char     **n_aliases;  /* alias list */
   int      n_addrtype;   /* net address type */
   int      n_net;        /* net number, host byte order */
};
```

`getnetbyname( )`, `getnetbyaddr( )`, and `getnetent( )` are the network counterparts to the host routines described above.

Protocol Names

The `protoent` structure defines the protocol-name mapping used with `getprotobyname( )`, `getprotobynumber( )`, and `getprotoent( )`:

```
struct protoent {
   char     *p_name;          /* official protocol name */
   char     **p_aliases       /* alias list */
   int      p_proto;          /* protocol number */
};
```

In the UNIX domain, no protocol database exists.

Service Names

An Internet domain service resides at a specific, well-known port and uses a particular protocol. A service-name-to-port-number mapping is described by the `servent` structure:

```
struct serven
   char     *s_name;         /* official service name */
   char     **s_aliases;     /* alias list */
   int      s_port;          /* port number, network byte order */
   char     *s_proto;        /* protocol to use */
};
```

`getservbyname( )` maps service names and, optionally, a qualifying protocol to a `servent` structure. The call:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification of a telnet server using any protocol. The call:

```
sp = getservbyname("telnet", "tcp");
```

returns the telnet server that uses the TCP protocol. `getservbyport( )` and `getservent( )` are also provided. `getservbyport( )` has an interface similar to that of `getservbyname( )`; an optional protocol name can be specified to qualify lookups.

Other Routines

In addition to address-related database routines, there are several other routines that simplify manipulating names and addresses. Table 2–3 summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

TABLE 2–3 Runtime Library Routines

Call	Synopsis
`memcmp(`*s1*`, `*s2*`, `*n*`)`	Compares byte-strings; `0` if same, not `0` otherwise
`memcpy(`*s1*`, `*s2*`, `*n*`)`	Copies *n* bytes from *s2* to *s1*
`memset(`*base*`, `*value*`, `*n*`)`	Sets *n* bytes to `value` starting at `base`
`htonl(`*val*`)`	32-bit quantity from host into network byte order
`htons(`*val*`)`	16-bit quantity from host into network byte order

TABLE 2–3 Runtime Library Routines *(continued)*

Call	Synopsis
`ntohl(`*val*`)`	32-bit quantity from network into host byte order
`ntohs(`*val*`)`	16-bit quantity from network into host byte order

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, the host byte ordering is different from network byte order, so programs must sometimes byte-swap values. Routines that return network addresses do so in network order. Byte-swapping problems occur only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On certain machines, where these routines are not needed, they are defined as null macros.

Client-Server Programs

The most common form of distributed application is the client/server model. In this scheme, client processes request services from a server process.

An alternate scheme is a service server that can eliminate dormant server processes. An example is `inetd`, the Internet service daemon. `inetd` listens at a variety of ports, determined at start up by reading a configuration file. When a connection is requested on an `inetd` serviced port, `inetd` spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. `inetd` is described in more detail in “`inetd` Daemon” on page 52.

Servers

Most servers are accessed at well-known Internet port numbers or UNIX domain names. Code Example 2–7 illustrates the main loop of a remote-login server.

CODE EXAMPLE 2–7 Remote Login Server

```
main(argc, argv)
   int argc;
   char *argv[];
{
   int f;
   struct sockaddr_in from;
   struct sockaddr_in sin;
   struct servent *sp;

   sp = getservbyname("login", "tcp");


    if (sp == (struct servent *) NULL) {
      fprintf(stderr, "rlogind: tcp/login: unknown service");
      exit(1);
   }
   ...
   #ifndef DEBUG
   /* Disassociate server from controlling terminal. */
   ...
   #endif
    sin.sin_port = sp->s_port; /* Restricted port */
      sin.sin_addr.s_addr = INADDR_ANY;
      ...
      f = socket(AF_INET, SOCK_STREAM, 0);
      ...
      if (bind( f, (struct sockaddr *) &sin, sizeof sin ) == –1) {
      ...
      }
      ...
      listen(f, 5);
      while (TRUE) {
         int g, len = sizeof from;
         g = accept(f, (struct sockaddr *) &from, &len);
         if (g == –1) {
            if (errno != EINTR)
               syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
         }
         if (fork() == 0) {
            close(f);
            doit(g, &from);
         }
         close(g);
      }
      exit(0);
}
```

Code Example 2–8 shows how the server gets its service definition.

CODE EXAMPLE 2–8 Remote Login Server: Step 1

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
  fprintf(stderr, "rlogind: tcp/login: unknown service\n");
  exit(1);
}
```

The result from `getservbyname()` is used later to define the Internet port at which the program listens for service requests. Some standard port numbers are in `/usr/include/netinet/in.h`.

Code Example 2–9 shows how the server dissociates from the controlling terminal of its invoker in the non-`DEBUG` mode of operation.

CODE EXAMPLE 2–9 Dissociating From the Controlling Terminal

```
   (void) close(0);
   (void) close(1);
   (void) close(2);
   (void) open("/", O_RDONLY);
   (void) dup2(0, 1);
   (void) dup2(0, 2);
   setsid();
```

This prevents the server from receiving signals from the process group of the controlling terminal. After a server has dissociated itself, it cannot send reports of errors to a terminal and must log errors with `syslog()`.

A server next creates a socket and listens for service requests. `bind()` ensures that the server listens at the expected location. (The remote login server listens at a restricted port number, so it runs as super-user.)

Code Example 2–10 illustrates the main body of the loop.

CODE EXAMPLE 2–10 Remote Login Server: Main Body

```
while(TRUE) {
  int g, len = sizeof(from);
  if (g = accept(f, (struct sockaddr *) &from, &len) == –1) {
   if (errno != EINTR)
    syslog(LOG_ERR, "rlogind: accept: %m");
   continue;
  }
  if (fork() == 0) {  /* Child */
   close(f);
   doit(g, &from);
  }
```

(continued)

(Continuation)

```
  close(g);     /* Parent */
}
```

`accept()` blocks messages until a client requests service. `accept()` returns a failure indication if it is interrupted by a signal, such as `SIGCHLD`. The return value from `accept()` is checked and an error is logged with `syslog()` if an error has occurred.

The server then forks a child process and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by `accept()` is closed in the parent. The address of the client is passed to `doit()` for authenticating clients.

Clients

This section describes the steps taken by the client remote login process. As in the server, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
  fprintf(stderr,"rlogin: tcp/login: unknown service");
  exit(1);
}
```

Next, the destination host is looked up with a `gethostbyname()` call:

```
hp = gethostbyname(argv[1]);
if (hp == (struct hostent *) NULL) {
  fprintf(stderr, "rlogin: %s: unknown host", argv[1]);
  exit(2);
}
```

The next step is to connect to the server at the requested host and start the remote login protocol. The address buffer is cleared and filled with the Internet address of the foreign host and the port number at which the login server listens:

```
memset((char *) &server, 0, sizeof server);
memcpy((char*) &server.sin_addr,hp->h_addr,hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. `connect()` implicitly does a `bind()`, since `s` is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
  perror("rlogin: socket");
  exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof server) < 0) {
  perror("rlogin: connect");
  exit(4);
}
```

Connectionless Servers

Some services use datagram sockets. The `rwho` service provides status information on hosts connected to a local area network. (Avoid running `in.rwhod` because it causes heavy network traffic.) This service requires the ability to broadcast information to all hosts connected to a particular network. It is an example of datagram socket use.

A user on a host running the `rwho` server can get the current status of another host with `ruptime`. Typical output is illustrated in Code Example 2–11.

CODE EXAMPLE 2–11 Output of `ruptime` Program

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

Status information is periodically broadcast by the `rwho` server processes on each host. The server process also receives the status information and updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient because a lot of net traffic is generated. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

Code Example 2–12 shows a simplified version of the rwho server. It performs two tasks: receives status information broadcast by other hosts on the network and supplies the status of its host. The first task is done in the main loop of the program: Packets received at the rwho port are checked to be sure they were sent by another rwho server process, and are stamped with the arrival time. They then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs it. This application is prone to error, as a server might be down while a host is up.

CODE EXAMPLE 2–12 rwho Server

```
main()
{
   ...
   sp = getservbyname("who", "udp");
   net = getnetbyname("localnet");
   sin.sin_addr = inet_makeaddr(net->n_net, INADDR_ANY);
   sin.sin_port = sp->s_port;
   ...
   s = socket(AF_INET, SOCK_DGRAM, 0);
   ...
   on = 1;
   if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
         == –1) {
      syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
      exit(1);
   }
   bind(s, (struct sockaddr *) &sin, sizeof sin);
   ...
   signal(SIGALRM, onalrm);
   onalrm();
   while(1) {
      struct whod wd;
       int cc, whod, len = sizeof from;
      cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
         (struct sockaddr *) &from, &len);
      if (cc <= 0) {
      if (cc == –1 && errno != EINTR)
         syslog(LOG_ERR, "rwhod: recv: %m");
      continue;
      }
      if (from.sin_port != sp->s_port) {
         syslog(LOG_ERR, "rwhod: %d: bad from port",
            ntohs(from.sin_port));
         continue;
      }
      ...
      if (!verify( wd.wd_hostname)) {
         syslog(LOG_ERR, "rwhod: bad host name from %x",
            ntohl(from.sin_addr.s_addr));
         continue;
      }
      (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
      whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
      ...
```

(continued)

(Continuation)

```
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *) &wd, cc);
        (void) close(whod);
    }
    exit(0);
}
```

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho` servers to hear. This task is run by a timer and triggered with a signal. Locating the system status information is involved but uninteresting.

Status information is broadcast on the local network. For networks that do not support broadcast, use another scheme.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe problem. The system isolates host-specific data from applications by providing function calls that return the required data. (For example, `uname( )` returns the host's official name.) The `SIOCGIFCONF ioctl( )` call lets you find the networks to which a host is directly connected. A local network broadcasting mechanism has been implemented at the socket level. Combining these two features lets a process broadcast on any directly connected local network that supports broadcasting in a site-independent manner. This solves the problem of deciding how to propagate status with `rwho`, or more generally in broadcasting. Such status is broadcast to connected networks at the socket level, where the connected networks have been obtained through the appropriate `ioctl( )` calls. "Broadcasting and Determining Network Configuration" on page 48 details the specifics of broadcasting.

Advanced Topics

For most programmers, the mechanisms already described are enough to build distributed applications. Others need some of the additional features in this section.

Out-of-Band Data

The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independent of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message can be pending delivery at any time.

For communications protocols that support only in-band signaling (that is, urgent data is delivered in sequence with normal data), the message is extracted from the normal data stream and stored separately. This lets users choose between receiving the urgent data in order and receiving it out of sequence, without having to buffer the intervening data.

You can peek (with `MSG_PEEK`) at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by `SIGURG` with the appropriate `fcntl( )` call, as described in "Interrupt-Driven Socket I/O" on page 43 for `SIGIO`. If multiple sockets have out-of-band data waiting delivery, a `select( )` call for exceptional conditions can be used to determine the sockets with such data pending.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the `MSG_OOB` flag is applied to `send( )` or `sendto( )`. To receive out-of-band data, specify `MSG_OOB` to `recvfrom( )` or `recv( )` (unless out-of-band data is taken in line, in which case the `MSG_OOB` flag is not needed). The `SIOCATMARK` `ioctl` tells whether the read pointer currently points at the mark in the data stream:

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is `1` on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in Code Example 2–13. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process can also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (for example, TCP, the protocol used to provide socket streams in the Internet

domain). With such protocols, the out-of-band byte might not yet have arrived when a `recv()` is done with the `MSG_OOB` flag. In that case, the call returns the error of `EWOULDBLOCK`. Also, there might be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data before the urgent data can be delivered.

CODE EXAMPLE 2–13 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
  int out = FWRITE;
  char waste[BUFSIZ];
  int mark = 0;

  /* flush local terminal output */
  ioctl(1, TIOCFLUSH, (char *) &out);
  while(1) {
   if (ioctl(rem, SIOCATMARK, &mark) == -1) {
    perror("ioctl");
    break;
   }
   if (mark)
    break;
   (void) read(rem, waste, sizeof waste);
  }
  if (recv(rem, &mark, 1, MSG_OOB) == -1) {
   perror("recv");
   ...
  }
  ...
}
```

There is also a facility to retain the position of urgent in-line data in the socket stream. This is available as a socket-level option, `SO_OOBINLINE`. See the **`getsockopt`**(3N) manpage for usage. With this option, the position of urgent data (the mark) is retained, but the urgent data immediately follows the mark in the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Nonblocking Sockets

Some applications require sockets that do not block. For example, requests that cannot complete immediately and would cause the process to be suspended (awaiting completion) are not executed. An error code would be returned. After a

socket is created and any connection to another socket is made, it can be made nonblocking by issuing a `fcntl( )` call, as shown in Code Example 2–14.

CODE EXAMPLE 2–14 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == –1)
  perror("fcntl F_GETFL");
  exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == –1)
  perror("fcntl F_SETFL, FNDELAY");
  exit(1);
}
...
```

When doing I/O on a nonblocking socket, check for the error `EWOULDBLOCK` (in `errno.h`), which occurs when an operation would normally block. `accept( )`, `connect( )`, `send( )`, `recv( )`, `read( )`, and `write( )` can all return `EWOULDBLOCK`. If an operation such as a `send( )` cannot be done in its entirety, but partial writes work (such as when using a stream socket), the data that can be sent immediately are processed, and the return value is the amount actually sent.

Asynchronous Socket I/O

Asynchronous communication between processes is required in applications that handle multiple requests simultaneously. Asynchronous sockets must be `SOCK_STREAM` type. To make a socket asynchronous, you issue a `fcntl( )` call, as shown in Code Example 2–15.

CODE EXAMPLE 2–15 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET, SOCK_STREAM, 0);
...
```

(continued)

(Continuation)

```
if (fileflags = fcntl(s, F_GETFL ) == –1)
  perror("fcntl F_GETFL");
  exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == –1)
  perror("fcntl F_SETFL, FNDELAY | FASYNC");
  exit(1);
}
...
```

After sockets are initialized, connected, and made asynchronous, communication is similar to reading and writing a file asynchronously. A `send( )`, `write( )`, `recv( )`, or `read( )` initiates a data transfer. A data transfer is completed by a signal-driven I/O routine, described in the next section.

Interrupt-Driven Socket I/O

The `SIGIO` signal notifies a process when a socket (actually any file descriptor) has finished a data transfer. The steps in using `SIGIO` are:

- Set up a `SIGIO` signal handler with the `signal( )` or `sigvec( )` calls.
- Use `fcntl( )` to set the process ID or process group ID to route the signal to its own process ID or process group ID (the default process group of a socket is group `0`).
- Convert the socket to asynchronous, as shown in "Asynchronous Socket I/O" on page 42.

Code Example 2–16 shows some sample code to allow a given process to receive information on pending requests as they occur for a socket. With the addition of a handler for `SIGURG( )`, this code can also be used to prepare for receipt of `SIGURG` signals.

CODE EXAMPLE 2–16 Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
  perror("fcntl F_SETOWN");
  exit(1);
```

(continued)

(Continuation)

```
}
```

Signals and Process Group ID

For SIGURG and SIGIO, each socket has a process number and a process group ID. These values are initialized to zero, but can be redefined at a later time with the F_SETOWN fcntl( ), as in the previous example. A positive third argument to fcntl( ) sets the socket's process ID. A negative third argument to fcntl( ) sets the socket's process group ID. The only allowed recipient of SIGURG and SIGIO signals is the calling process. A similar fcntl( ), F_GETOWN, returns the process number of a socket.

Reception of SIGURG and SIGIO can also be enabled by using ioctl( ) to assign the socket to the user's process group:

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
  perror("ioctl: SIOCSPGRP");
}
```

Another signal that is useful in server processes is SIGCHLD. This signal is delivered to a process when any child process changes state. Normally, servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop shown previously can be augmented as shown in Code Example 2–17.

CODE EXAMPLE 2–17 SIGCHLD Signal

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
  int g, len = sizeof from;
  g = accept(f, (struct sockaddr *) &from, &len);
  if (g < 0) {
   if (errno != EINTR)
    syslog(LOG_ERR, "rlogind: accept: %m");
   continue;
  }
  ...
}
```

(continued)

(Continuation)

```
#include <wait.h>

reaper()
{
  int options;
  int error;
  siginfo_t info;

  options = WNOHANG | WEXITED;
  bzero((char *) &info, sizeof(info));
  error = waitid(P_ALL, 0, &info, options);
}
```

If the parent server process fails to reap its children, zombie processes result.

Selecting Specific Protocols

If the third argument of the `socket()` call is `0`, `socket()` selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. When using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, it may be important for the protocol argument to set up de-multiplexing. For example, raw sockets in the Internet domain can be used to implement a new protocol on IP, and the socket receives packets only for the protocol specified. To obtain a particular protocol, determine the protocol number as defined in the protocol domain. For the Internet domain, use one of the library routines discussed in "Standard Routines" on page 30, such as `getprotobyname()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This results in a socket `s` using a stream-based connection, but with protocol type of `newtcp` instead of the default `tcp`.

Address Binding

TCP and UDP use a 4-tuple of *local IP address*, *local port number*, *foreign IP address*, and *foreign port number* to do their addressing. TCP requires these 4-tuples to be unique. UDP does not. It is unrealistic to expect user programs to always know proper values to use for the local address and local port, since a host can reside on multiple networks and the set of allocated port numbers is not directly accessible to a user. To avoid these problems, you can leave parts of the address unspecified and let the system assign the parts appropriately when needed. Various portions of these tuples may be specified by various parts of the sockets API.

`bind()` local address or local port or both

`connect()` foreign address and foreign port

A call to `accept()` retrieves connection information from a foreign client, so it causes the local address and port to be specified to the system (even though the caller of `accept()` didn't specify anything), and the foreign address and port to be returned.

A call to `listen()` can cause a local port to be chosen. If no explicit `bind()` has been done to assign local information, `listen()` causes an ephemeral port number to be assigned.

A service that resides at a particular port, but which does not care what local address is chosen, can `bind()` itself to its port and leave the local address unspecified (set to `INADDR_ANY`, a constant defined in `<netinet/in.h>`). If the local port need not be fixed, a call to `listen()` causes a port to be chosen. Specifying an address of `INADDR_ANY` or a port number of 0 is known as wildcarding.

The wildcard address simplifies local address binding in the Internet domain. The sample code below binds a specific port number, MYPORT, to a socket, and leaves the local address unspecified.

CODE EXAMPLE 2–18 Bind Port Number to Socket

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
  s = socket(AF_INET, SOCK_STREAM, 0);
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = htonl(INADDR_ANY);
  sin.sin_port = htons(MYPORT);
  bind(s, (struct sockaddr *) &sin, sizeof sin);
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses can receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has two interfaces with addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as in Code Example 2–18, the process can accept connection requests addressed to 128.32.0.4 or 10.0.0.78. To allow only hosts on a specific network to connect to it, a server binds the address of the interface on the appropriate network.

Similarly, a local port number can be left unspecified (specified as `0`), in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
sin.sin_addr.s_addr = inet_addr("127.0.0.1");
sin.sin_family = AF_INET;
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:

- The first is that Internet port numbers less than 1024 (`IPPORT_RESERVED`) are reserved for privileged users (that is, the superuser). Nonprivileged users can use any Internet port number greater than 1024. The largest Internet port number is 65535.
- The second criterion is that the port number is not currently bound to some other socket.

The port number and IP address of the client is found through either `accept()` (the *from* result) or `getpeername()`.

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application. This is because associations are created in a two-step process. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation, the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, you must perform an option call before address binding:

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses already in use can be bound. This does not violate the uniqueness requirement, because the system still verifies at connect time that any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error `EADDRINUSE` is returned.

Broadcasting and Determining Network Configuration

Messages sent by datagram sockets can be broadcast to reach all of the hosts on an attached network. The network must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Broadcasting is usually used for either of two reasons: to find a resource on a local network without having its address, or functions like routing require that information be sent to all accessible neighbors.

To send a broadcast message, create an Internet datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The datagram can be broadcast on only one network by sending to the network's broadcast address. A datagram can also be broadcast on all attached networks by sending to the special address `INADDR_BROADCAST`, defined in `netinet/in.h`.

The system provides a mechanism to determine a number of pieces of information (including the IP address and broadcast address) about the network interfaces on the system. The `SIOCGIFCONF ioctl( )` call returns the interface configuration of a host in a single `ifconf` structure. This structure contains an array of `ifreq` structures, one for each address domain supported by each network interface to which the host is connected. Code Example 2–19 shows these structures defined in `net/if.h`.

CODE EXAMPLE 2–19 `net/if.h` Header File

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
union {
  struct sockaddr ifru_addr;
  struct sockaddr ifru_dstaddr;
  char ifru_oname[IFNAMSIZ]; /* other if name */
  struct sockaddr ifru_broadaddr;
  short ifru_flags;
  int ifru_metric;
  char ifru_data[1]; /* interface dependent data */
  char ifru_enaddr[6];
} ifr_ifru;
```

(continued)

(Continuation)

```
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

The call that obtains the interface configuration is:

```
/*
 * Do SIOCGIFNUM ioctl to find the number of interfaces
 *
 * Allocate space for number of interfaces found
 *
 * Do SIOCGIFCONF with allocated buffer
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
        numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
        perror("ioctl(SIOCGIFCONF)");
        exit(1);
}
...
}
```

After this call, *buf* contains an array of `ifreq` structures, one for each network to which the host is connected. These structures are ordered first by interface name, then by supported address families. `ifc.ifc_len` is set to the number of bytes used by the `ifreq` structures.

Each structure has a set of interface flags that tell whether the corresponding network is up or down, point-to-point or broadcast, and so on. Code Example 2–20 shows the `SIOCGIFFLAGS ioctl()` returning these flags for an interface specified by an `ifreq` structure.

CODE EXAMPLE 2–20 Obtaining Interface Flags

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
   /*
    * Be careful not to use an interface devoted to an address
    * domain other than those intended.
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
      ...
   }
   /* Skip boring cases */
   if ((ifr->ifr_flags & IFF_UP) == 0 ||
      (ifr->ifr_flags & IFF_LOOPBACK) ||
      (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
      continue;
}
```

Code Example 2–21 shows the broadcast of an interface can be obtained with the `SIOGGIFBRDADDR ioctl()`.

CODE EXAMPLE 2–21 Broadcast Address of an Interface

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
  ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
  sizeof ifr->ifr_broadaddr);
```

The `SIOGGIFBRDADDR ioctl()` can also be used to get the destination address of a point-to-point interface.

After the interface broadcast address is obtained, transmit the broadcast datagram with `sendto()`:

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

Use one `sendto()` for each interface to which the host is connected that supports the broadcast or point-to-point addressing.

Zero Copy and Checksum Offload

In Solaris 2.6, the TCP/IP protocol stack has been enhanced to support two new features: zero copy and TCP checksum offload.

- Zero copy uses virtual memory MMU remapping and a copy-on-write technique to move data between the application and the kernel space.
- Checksum offloading relies on special hardware logic to offload the TCP checksum calculation.

Note - Although zero copy and checksum offloading are functionally independent of one another, they have to work together to obtain the optimal performance. Checksum offloading requires hardware support from the network interface and, without this hardware support, zero copy is not enabled.

Zero copy requires that the applications supply page-aligned buffers before VM page remapping can be applied. Applications should use large, circular buffers on the transmit side to avoid expensive copy-on-write faults. A typical buffer allocation is sixteen 8k buffers.

Socket Options

You can set and get several options on sockets through `setsockopt()` and `getsockopt()`; for example by changing the send or receive buffer space. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

Note - In some cases, such as setting the buffer sizes, these are only hints to the operating system. The operating system reserves the right to adjust the values appropriately.

Table 2–4 shows the arguments of the calls.

TABLE 2–4 `setsockopt()` and `getsockopt()` Arguments

Arguments	Description
s	Socket on which the option is to be applied
level	Specifies the protocol level, such as socket level, indicated by the symbolic constant `SOL_SOCKET` in `sys/socket.h`

TABLE 2–4 `setsockopt( )` and `getsockopt( )` Arguments *(continued)*

Arguments	Description
optname	Symbolic constant defined in `sys/socket.h` that specifies the option
optval	Points to the value of the option
optlen	Points to the length of the value of the option

For `getsockopt( )`, *optlen* is a value-result argument, initially set to the size of the storage area pointed to by *optval* and set on return to the length of storage used.

It is sometimes useful to determine the type (for example, stream or datagram) of an existing socket. Programs invoked by `inetd` can do this by using the `SO_TYPE` socket option and the `getsockopt( )` call:

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

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) <0) {
  ...
}
```

After `getsockopt( )`, `type` is set to the value of the socket type, as defined in `sys/socket.h`. For a datagram socket, `type` would be `SOCK_DGRAM`.

`inetd` Daemon

One of the daemons provided with the system is `inetd`. It is invoked at start-up time, and gets the services for which it listens from the `/etc/inetd.conf` file. The daemon creates one socket for each service listed in `/etc/inetd.conf`, binding the appropriate port number to each socket. See the **`inetd`**(1M) man page for details.

`inetd` polls each socket, waiting for a connection request to the service corresponding to that socket. For `SOCK_STREAM` type sockets, `inetd` does an `accept( )` on the listening socket, `fork( )`s, `dup( )`s the new socket to file descriptors `0` and `1` (`stdin` and `stdout`), closes other open file descriptors, and `exec( )`s the appropriate server.

The primary benefit of `inetd` is that services that are not in use are not taking up machine resources. A secondary benefit is that `inetd` does most of the work to

establish a connection. The server started by `inetd` has the socket connected to its client on file descriptors `0` and `1`, and can immediately `read( )`, `write( )`, `send( )`, or `recv( )`. Servers can use buffered I/O as provided by the `stdio` conventions, as long as they use `fflush( )` when appropriate.

`getpeername( )` returns the address of the peer (process) connected to a socket; it is useful in servers started by `inetd`. For example, to log the Internet address in decimal dot notation (such as 128.32.0.4, which is conventional for representing an IP address of a client), an `inetd` server could use the following:

```
struct sockaddr_in name;
int namelen = sizeof name;
 ...
if (getpeername(0, (struct sockaddr *) &name, &namelen) < 0) {
  syslog(LOG_ERR, "getpeername: %m");
  exit(1);
} else
  syslog(LOG_INFO, "Connection from %s",
   inet_ntoa(name.sin_addr));
 ...
```

CHAPTER 3

Programming with XTI and TLI

The X/Open Transport Interface (XTI) and the Transport Layer Interface (TLI) are a set of functions that constitute a network programming interface. XTI is an evolution from the older TLI interface available on SunOS 4. Both interfaces are supported, though XTI represents the future direction of this set of interfaces.

XTI/TLI Is Multithread Safe

The interfaces described in this chapter are multithread safe. This means that applications containing XTI/TLI function calls can be used freely in a multithreaded application. However, the degree of concurrency available to applications is not specified.

XTI/TLI Are Not Asynchronous Safe

The XTI/TLI interface behavior has not been well specified in an asynchronous environment. It is not recommended that these interfaces be used from signal handler routines.

What Are XTI and TLI?

TLI was introduced with AT&T's System V, Release 3 in 1986. It provided a transport layer interface API. TLI was modeled after the ISO Transport Service Definition and provides an API between the OSI transport and session layers. TLI interfaces evolved further in AT&T System V, Release 4 version of Unix and were made available in release of SunOS 5.6 operating system interfaces, too.

XTI interfaces are an evolution of TLI interfaces and represent the future direction of this family of interfaces. Compatibility for applications using TLI interfaces is available. There is no intrinsic need to port TLI applications to XTI immediately. New applications can use the XTI interfaces and older applications can be ported to XTI when necessary.

TLI is implemented as a set of function calls in a library (`libnsl`) with which the applications link. XTI applications are compiled using the c89 frontend and must be linked with the `xnet` library (`libxnet`). For additional information on compiling with XTI, see **`standards`**(5).

Note - An application using the XTI interface uses the `xti.h` header file, whereas an application using the TLI interface includes the `tiuser.h` header file.

Intrinsic to XTI/TLI are the notions of *transport endpoints* and a *transport provider*. The transport endpoints are two entities that are communicating, and the transport provider is the set of routines on the host that provides the underlying communication support. XTI/TLI is the interface to the transport provider, not the provider itself. See Figure 3–1.

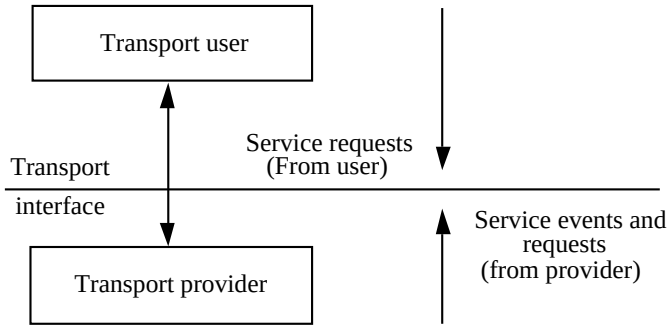

Figure 3–1 How XTI/TLI Works

XTI/TLI code can be written to be independent of current transport providers in conjunction with some additional interfaces and mechanisms described in Chapter 4. The SunOS 5 product includes some transport providers (TCP, for example) as part of the base operating system. A transport provider performs services, and the transport user requests the services. The transport user issues service requests to the transport provider. An example is a request to transfer data over a connection TCP and UDP.

XTI/TLI can also be used for transport-independent programming. XTI/TLI has two components to achieve this:

- Library routines that perform the transport services, in particular, transport selection and name-to-address translation. The network services library includes a set of functions that implement XTI/TLI for user processes. See Chapter 4.

  Programs using TLI should be linked with the network services library, `libnsl`, as follows:

```
cc prog.c -lnsl
```

- State transition rules that define the sequence in which the transport routines can be invoked. For more information on state transition rules, see section, “State Transitions” on page 93. The state tables define the legal sequence of library calls based on the state and the handling of events. These events include user-generated library calls, as well as provider-generated event indications. XTI/TLI programmers should understand all state transitions before using the interface.

XTI/TLI provides two modes of service: connection mode and connectionless mode. The next two sections give an overview of these modes.

Connectionless Mode

Connectionless mode is message oriented. Data are transferred in self-contained units with no relationship between the units. This service requires only an established association between the peer users that determines the characteristics of the data. All information required to deliver a message (such as the destination address) is presented to the transport provider, with the data to be transmitted, in one service request. Each message is entirely self-contained. Use connectionless mode service for applications that:

- Have short-term request/response interactions
- Are dynamically reconfigurable
- Do not require sequential delivery of data

Connectionless transports can be unreliable. They need not necessarily maintain message sequence, and messages are sometimes lost.

Connectionless Mode Routines

Connectionless-mode transport service has two phases: local management and data transfer. The local management phase defines the same local operations as for the connection mode service.

The data transfer phase lets a user transfer data units (usually called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. `t_sndudata( )` sends and `t_rcvudata( )` receives messages. Table 3–1 summarizes all routines for connectionless mode data transfer.

TABLE 3–1 Routines for Connectionless-Mode Data Transfer

Command	Description
`t_sndudata`	Sends a message to another user of the transport
`t_rcvudata`	Receives a message sent by another user of the transport
`t_rcvuderr`	Retrieves error information associated with a previously sent message

Connectionless Mode Service

Connectionless mode service is appropriate for short-term request/response interactions, such as transaction-processing applications. Data are transferred in self-contained units with no logical relationship required among multiple units.

Endpoint Initiation

Transport users must initialize XTI/TLI endpoints before transferring data. They must choose the appropriate connectionless service provider using `t_open()` and establish its identity using `t_bind()`.

Use `t_optmgmt()` to negotiate protocol options. Like connection mode service, each transport provider specifies the options, if any, it supports. Option negotiation is a protocol-specific activity. In Code Example 3–1, the server waits for incoming queries, and processes and responds to each query. The example also shows the definitions and initiation sequence of the server.

CODE EXAMPLE 3–1 CLTS Server

```
#include <stdio.h>
#include <fcntl.h>
#include <xti.h> /* TLI applications use <tiuser.h>  */
#define SRV_ADDR 2 /* server's well known address */

main()
{
   int fd;
   int flags;
   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;
   extern int t_errno;

 if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *) NULL))
        == –1) {
      t_error("unable to open /dev/exmp");
      exit(1);
   }
  if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR))
        == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->addr.len = sizeof(int);
   *(int *)bind->addr.buf = SRV_ADDR;
   bind->qlen = 0;
   if (t_bind(fd, bind, bind) == –1) {
```

(continued)

(Continuation)

```
    t_error("t_bind failed");
    exit(3);
}
/*
 * TLI interface applications need the following code which
 * is no longer needed for XTI interface applications.
 * ------------------------------------
 * Verify if the bound address correct?
 *
 * if (bind -> addr.len != sizeof(int) ||
 *      *(int *)bind->addr.buf != SRV_ADDR) {
 * fprintf(stderr, "t_bind bound wrong address\n");
 * exit(4);
 * }
 * --------------------------------------
 */
```

The server establishes a transport endpoint with the desired transport provider using `t_open( )`. Each provider has an associated service type, so the user can choose a particular service by opening the appropriate transport provider file. This connectionless mode server ignores the characteristics of the provider returned by `t_open( )` by setting the third argument to `NULL`. The transaction server assumes the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the `T_CLTS` service type (connectionless transport service, or datagram).
- The transport provider does not require any protocol-specific options.

The connectionless server binds a transport address to the endpoint so that potential clients can access the server. A `t_bind` structure is allocated using `t_alloc( )` and the `buf` and `len` fields of the address are set accordingly.

One difference between a connection mode server and a connectionless mode server is that the `qlen` field of the `t_bind` structure is `0` for connectionless mode service. There are no connection requests to queue.

XTI/TLI interfaces define an inherent client-server relationship between two users while establishing a transport connection in the connection mode service. No such relationship exists in connectionless mode service.

TLI requires that the server check the bound address returned by `t_bind( )` to ensure that it is the same as the one supplied. `t_bind( )` can also bind the endpoint to a separate, free address if the one requested is busy.

Data Transfer

After a user has bound an address to the transport endpoint, datagrams can be sent or received over the endpoint. Each outgoing message carries the address of the destination user. XTI/TLI also lets you specify protocol options to the transfer of the data unit (for example, transit delay). Each transport provider defines the set of options on a datagram. When the datagram is passed to the destination user, the associated protocol options can be passed, too.

Code Example 3–2 illustrates the data transfer phase of the connectionless mode server.

CODE EXAMPLE 3–2 Data Transfer Routine

```
 if ((ud = (struct t_unitdata *) t_alloc(fd, T_UNITDATA,T_ALL))
        == (struct t_unitdata *) NULL) {
     t_error("t_alloc of t_unitdata struct failed");
     exit(5);
  }
  if ((uderr = (struct t_uderr *) t_alloc(fd, T_UDERROR, T_ALL))
        == (struct t_uderr *) NULL) {
     t_error("t_alloc of t_uderr struct failed");
     exit(6);
  }
  while(1) {
     if (t_rcvudata(fd, ud, &flags) == -1) {
        if (t_errno == TLOOK) {
              /* Error on previously sent datagram */
              if(t_rcvuderr(fd, uderr) == -1) {
                 exit(7);
              }
           fprintf(stderr, "bad datagram, error=%d\n",
              uderr->error);
           continue;
        }
        t_error("t_rcvudata failed");
        exit(8);
     }
     /*
      * Query() processes the request and places the response in
      * ud->udata.buf, setting ud->udata.len
      */
     query(ud);
     if (t_sndudata(fd, ud) == -1) {
        t_error("t_sndudata failed");
        exit(9);
     }
  }
}

/* ARGS USED */
void
query(ud)
struct t_unitdate *ud;
```

(continued)

(Continuation)

```
{
   /* Merely a stub for simplicity */
}
```

To buffer datagrams, the server first allocates a `t_unitdata` structure, which has the following format:

```
struct t_unitdata {
  struct netbuf addr;
  struct netbuf opt;
  struct netbuf udata;
}
```

`addr` holds the source address of incoming datagrams and the destination address of outgoing datagrams. `opt` holds any protocol options on the datagram. `udata` holds the data. The `addr`, `opt`, and `udata` fields must all be allocated with buffers large enough to hold any possible incoming values. The `T_ALL` argument of `t_alloc( )` ensures this and sets the `maxlen` field of each `netbuf` structure accordingly. The provider does not support protocol options in this example, so `maxlen` is set to `0` in the `opt netbuf` structure. The server also allocates a `t_uderr` structure for datagram errors.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls `t_rcvudata( )` to receive the next query. `t_rcvudata( )` blocks until a datagram arrives, and returns it.

The second argument of `t_rcvudata( )` identifies the `t_unitdata` structure in which to buffer the datagram.

The third argument, `flags`, points to an integer variable and can be set to `T_MORE` on return from `t_rcvudata( )` to indicate that the user's `udata` buffer is too small to store the full datagram.

If this happens, the next call to `t_rcvudata( )` retrieves the rest of the datagram. Because `t_alloc( )` allocates a `udata` buffer large enough to store the maximum size datagram, this transaction server does not have to check `flags`. This is true only of `t_rcvudata( )` and not of any other receive primitives.

When a datagram is received, the transaction server calls its `query` routine to process the request. This routine stores a response in the structure pointed to by `ud`, and sets `ud-->udata.len` to the number of bytes in the response. The source address returned by `t_rcvudata( )` in `ud-->addr` is the destination address for `t_sndudata( )`. When the response is ready, `t_sndudata( )` is called to send the response to the client.

Datagram Errors

If the transport provider cannot process a datagram sent by `t_sndudata( )`, it returns a unit data error event, `T_UDERR`, to the user. This event includes the destination address and options of the datagram, and a protocol-specific error value that identifies the error. Datagram errors are protocol specific.

Note - A unit data error event does not always indicate success or failure in delivering the datagram to the specified destination. Remember, connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of an error when it tries to receive another datagram. In this case, `t_rcvudata( )` fails, setting `t_errno` to `TLOOK`. If `TLOOK` is set, the only possible event is `T_UDERR`, so the server calls `t_rcvuderr( )` to retrieve the event. The second argument of `t_rcvuderr( )` is the `t_uderr` structure that was allocated earlier. This structure is filled in by `t_rcvuderr( )` and has the following format:

```
struct t_uderr {
  struct netbuf addr;
  struct netbuf opt;
  t_scalar_t error;
}
```

where `addr` and `opt` identify the destination address and protocol options specified in the bad datagram, and `error` is a protocol-specific error code. The transaction server prints the error code, then continues.

Connection Mode

Connection mode is circuit oriented. Data are transmitted in sequence over an established connection. The mode also provides an identification procedure that avoids address resolution and transmission in the data transfer phase. Use this service for applications that require data stream-oriented interactions. Connection mode transport service has four phases:

- Local management
- Connection establishment
- Data transfer
- Connection release

The local management phase defines local operations between a transport user and a transport provider as shown in Figure 3–2. For example, a user must establish a

channel of communication with the transport provider. Each channel between a transport user and transport provider is a unique endpoint of communication, and is called the transport endpoint. `t_open( )` lets a user choose a particular transport provider to supply the connection mode services, and establishes the transport endpoint.

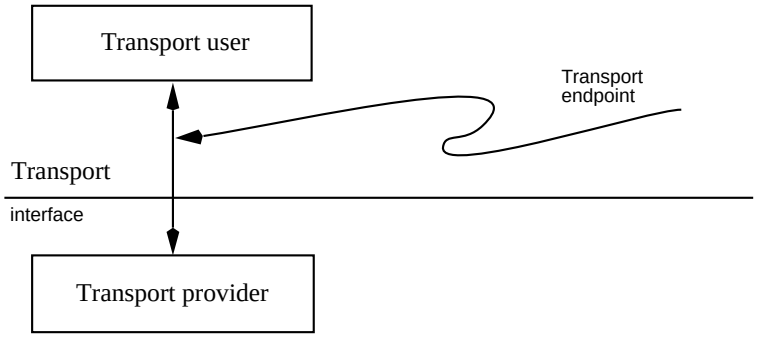

Figure 3–2 Transport Endpoint

Connection Mode Routines

Each user must establish an identity with the transport provider. A transport address is associated with each transport endpoint. One user process can manage several transport endpoints. In connection mode service, one user requests a connection to another user by specifying the other's address. The structure of a transport address is defined by the transport provider. An address can be as simple as an unstructured character string (for example, `file_server`), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses can be assigned to the endpoint of a transport by `t_bind( )`.

In addition to `t_open( )` and `t_bind( )`, several routines support local operations. Table 3–2 summarizes all local management routines of XTI/TLI.

TABLE 3–2 Routines of XTI/TLI for Operating on the Endpoint

Command	Description
`t_alloc`	Allocates XTI/TLI data structures
`t_bind`	Binds a transport address to a transport endpoint

TABLE 3–2 Routines of XTI/TLI for Operating on the Endpoint *(continued)*

Command	Description
`t_close`	Closes a transport endpoint
`t_error`	Prints an XTI/TLI error message
`t_free`	Frees structures allocated using `t_alloc`
`t_getinfo`	Returns a set of parameters associated with a particular transport provider
`t_getprotaddr`	Returns the local and/or remote address associated with endpoint (XTI only)
`t_getstate`	Returns the state of a transport endpoint
`t_look`	Returns the current event on a transport endpoint
`t_open`	Establishes a transport endpoint connected to a chosen transport provider
`t_optmgmt`	Negotiates protocol-specific options with the transport provider
`t_sync`	Synchronizes a transport endpoint with the transport provider
`t_unbind`	Unbinds a transport address from a transport endpoint

The connection phase lets two users create a connection, or virtual circuit, between them, as shown in Figure 3–3.

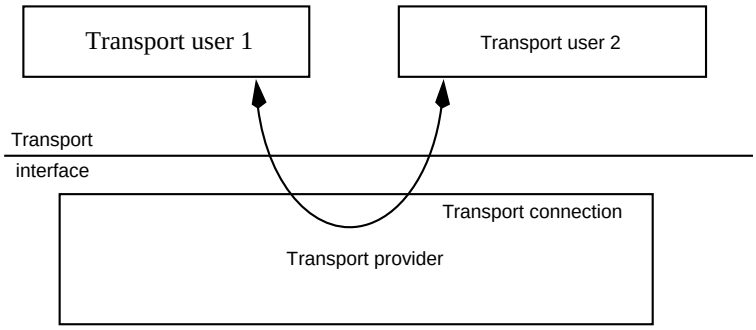

Figure 3–3 Transport Connection

For example, the connection phase occurs when a server advertises its service to a group of clients, then blocks on `t_listen()` to wait for a request. A client tries to connect to the server at the advertised address by a call to `t_connect()`. The connection request causes `t_listen()` to return to the server, which can call `t_accept()` to complete the connection.

Table 3–3 summarizes all routines available for establishing a transport connection. Refer to man pages for the specifications on these routines.

TABLE 3–3 Routines for Establishing a Transport Connection

Command	Description
`t_accept`	Accepts a request for a transport connection
`t_connect`	Establishes a connection with the transport user at a specified destination
`t_listen`	Listens for connect request from another transport user
`t_rcvconnect`	Completes connection establishment if `t_connect` was called in asynchronous mode (see "Advanced Topics" on page 86)

The data transfer phase lets users transfer data in both directions via the connection. `t_snd()` sends and `t_rcv()` receives data through the connection. It is assumed that all data sent by one user is guaranteed to be delivered to the other user in the order in which it was sent. Table 3–4 summarizes the connection mode data-transfer routines.

TABLE 3–4 Connection Mode Data Transfer Routines

Command	Description
`t_rcv`	Receives data that has arrived over a transport connection
`t_snd`	Sends data over an established transport connection

XTI/TLI has two types of connection release. The abortive release directs the transport provider to release the connection immediately. Any previously sent data that has not yet been transmitted to the other user can be discarded by the transport provider. `t_snddis()` initiates the abortive disconnect. `t_rcvdis()` receives the abortive disconnect. Transport providers usually support some form of abortive release procedure.

Some transport providers also support an orderly release that terminates communication without discarding data. `t_sndrel()` and `t_rcvrel()` perform this function. Table 3–5 summarizes the connection release routines. Refer to man pages for the specifications on these routines.

TABLE 3–5 Connection Release Routines

Command	Description
`t_rcvdis`	Returns a reason code for a disconnection and any remaining user data
`t_rcvrel`	Acknowledges receipt of an orderly release of a connection request
`t_snddis`	Aborts a connection or rejects a connect request
`t_sndrel`	Requests the orderly release of a connection

Connection Mode Service

The main concepts of connection mode service are illustrated through a client program and its server. The examples are presented in segments.

In the examples, the client establishes a connection to a server process. The server transfers a file to the client. The client receives the file contents and writes them to standard output.

Endpoint Initiation

Before a client and server can connect, each must first open a local connection to the transport provider (the transport endpoint) through `t_open()`, and establish its identity (or address) through `t_bind()`.

Many protocols perform a subset of the services defined in XTI/TLI. Each transport provider has characteristics that determine the services it provides and limit the services. Data defining the transport characteristics are returned by `t_open()` in a `t_info` structure. Table 3–6 shows the fields in a `t_info` structure.

TABLE 3–6 `t_info` Structure

Field	Content
`addr`	Maximum size of a transport address
`options`	Maximum bytes of protocol-specific options that can be passed between the transport user and transport provider
`tsdu`	Maximum message size that can be transmitted in either connection mode or connectionless mode
`etsdu`	Maximum expedited data message size that can be sent over a transport connection
`connect`	Maximum number of bytes of user data that can be passed between users during connection establishment
`discon`	Maximum bytes of user data that can be passed between users during the abortive release of a connection
`servtype`	The type of service supported by the transport provider

The three service types defined by XTI/TLI are:

1. `T_COTS` — The transport provider supports connection mode service but does not provide the orderly release facility. Connection termination is abortive, and any data not already delivered is lost.
2. `T_COTS_ORD` — The transport provider supports connection mode service with the orderly release facility.

3. T_CLTS — The transport provider supports connectionless mode service.

Only one such service can be associated with the transport provider identified by t_open( ).

t_open( ) returns the default provider characteristics of a transport endpoint. Some characteristics can change after an endpoint has been opened. This happens with negotiated options (option negotiation is described later in this section). t_getinfo( ) returns the current characteristics of a transport endpoint.

After a user establishes an endpoint with the chosen transport provider, the client and server must establish their identities. t_bind( ) does this by binding a transport address to the transport endpoint. For servers, this routine informs the transport provider that the endpoint is used to listen for incoming connect requests.

t_optmgmt( ) can be used during the local management phase. It lets a user negotiate the values of protocol options with the transport provider. Each transport protocol defines its own set of negotiable protocol options, such as quality-of-service parameters. Because the options are protocol-specific, only applications written for a specific protocol use this function.

Client

The local management requirements of the example client and server are used to discuss details of these facilities. Code Example 3–3 shows the definitions needed by the client program, followed by its necessary local management steps.

CODE EXAMPLE 3–3 Client Implementation of Open and Bind

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#define SRV_ADDR 1          /* server's address */

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   extern int t_errno;

   if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *),NULL))
         == –1) {
      t_error("t_open failed");
      exit(1);
   }
   if (t_bind(fd, (struct t_bind *) NULL, (struct t_bind *) NULL)
         == –1) {
```

(continued)

(Continuation)

```
    t_error("t_bind failed");
    exit(2);
}
```

The first argument of `t_open()` is the path of a file system object that identifies the transport protocol. `/dev/exmp` is the example name of a special file that identifies a generic, connection-based transport protocol. The second argument, `O_RDWR`, specifies to open for both reading and writing. The third argument points to a `t_info` structure in which to return the service characteristics of the transport.

This data is useful to write protocol-independent software (see "Guidelines to Protocol Independence" on page 100). In this example, a `NULL` pointer is passed. For Code Example 3–3, the transport provider must have the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the `T_COTS_ORD` service type, since the example uses orderly release.
- The transport provider does not require protocol-specific options.

If the user needs a service other than `T_COTS_ORD`, another transport provider can be opened. An example of the `T_CLTS` service invocation is shown in the section "Read / Write Interface" on page 84.

`t_open()` returns the transport endpoint file handle that is used by all subsequent XTI / TLI function calls. The identifier is a file descriptor from opening the transport protocol file. See **open**(2).

The client then calls `t_bind()` to assign an address to the endpoint. The first argument of `t_bind()` is the transport endpoint handle. The second argument points to a `t_bind` structure that describes the address to bind to the endpoint. The third argument points to a `t_bind` structure that describes the address that the provider has bound.

The address of a client is rarely important because no other process tries to access it. That is why the second and third arguments to `t_bind()` are `NULL`. The second `NULL` argument directs the transport provider to choose an address for the user.

If `t_open()` or `t_bind()` fails, the program calls `t_error()` to display an appropriate error message via `stderr`. The global integer `t_errno` is assigned an error value. A set of error values is defined in `tiuser.h`.

`t_error()` is analogous to `perror()`. If the transport function error is a system error, `t_errno()` is set to `TSYSERR`, and `errno` is set to the appropriate value.

Server

The server example must also establish a transport endpoint at which to listen for connection requests. Code Example 3–4 shows the definitions and local management steps.

CODE EXAMPLE 3–4 Server Implementation of Open and Bind

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT –1
#define SRV_ADDR 1        /* server's address */
int conn_fd;   /* connection established here */
extern int t_errno;

main()
{
   int listen_fd;       /* listening transport endpoint */
   struct t_bind *bind;
   struct t_call *call;

   if ((listen_fd = t_open("/dev/exmp", O_RDWR,
      (struct t_info *) NULL)) == –1) {
      t_error("t_open failed for listen_fd");
      exit(1);
   }
   if ((bind = (struct t_bind *)t_alloc( listen_fd, T_BIND, T_ALL))
         == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->qlen = 1;

   /*
    * Because it assumes the format of the provider's address,
    * this program is transport-dependent
    */
    bind->addr.len = sizeof(int);
   *(int *) bind->addr.buf = SRV_ADDR;
   if (t_bind (listen_fd, bind, bind) < 0 ) {
      t_error("t_bind failed for listen_fd");
      exit(3);
   }

   #if (!defined(_XOPEN_SOURCE) ||(_XOPEN_SOURCE_EXTENDED –0 != 1))
   /*
    * Was the correct address bound?
    *
    * When using XTI, this test is unnecessary
    */

   if (bind->addr.len != sizeof(int) ||
```

(continued)

(Continuation)

```
  *(int *)bind->addr.buf != SRV_ADDR) {
  fprintf(stderr, "t_bind bound wrong address\n");
  exit(4);
}
#endif
```

Like the client, the server first calls `t_open()` to establish a transport endpoint with the desired transport provider. The endpoint, `listen_fd`, is used to listen for connect requests.

Next, the server binds its address to the endpoint. This address is used by each client to access the server. The second argument points to a `t_bind` structure that specifies the address to bind to the endpoint. The `t_bind` structure has the following format:

```
struct t_bind {
  struct netbuf addr;
  unsigned qlen;
}
```

Where `addr` describes the address to be bound, and `qlen` specifies the maximum number of outstanding connect requests. All XTI structure and constant definitions made visible for use by applications programs through `xti.h`. All TLI structure and constant definitions are in `tiuser.h`.

The address is specified in the `netbuf` structure with the following format:

```
struct netbuf {
  unsigned int maxlen;
  unsigned int len;
  char *buf;
}
```

Where `maxlen` specifies the maximum length of the buffer in bytes, `len` specifies the bytes of data in the buffer, and `buf` points to the buffer that contains the data.

In the `t_bind` structure, the data identifies a transport address. `qlen` specifies the maximum number of connect requests that can be queued. If the value of `qlen` is positive, the endpoint can be used to listen for connect requests. `t_bind()` directs the transport provider to queue connect requests for the bound address immediately. The server must dequeue each connect request and accept or reject it. For a server that fully processes a single connect request and responds to it before receiving the next request, a value of `1` is appropriate for `qlen`. Servers that dequeue several connect requests before responding to any should specify a longer queue. The server in this example processes connect requests one at a time, so `qlen` is set to `1`.

`t_alloc( )` is called to allocate the `t_bind` structure. `t_alloc( )` has three arguments: a file descriptor of a transport endpoint; the identifier of the structure to allocate; and a flag that specifies which, if any, `netbuf` buffers to allocate. `T_ALL` specifies to allocate all `netbuf` buffers, and causes the `addr` buffer to be allocated in this example. Buffer size is determined automatically and stored in the `maxlen` field.

Each transport provider manages its address space differently. Some transport providers allow a single transport address to be bound to several transport endpoints, while others require a unique address per endpoint. XTI and TLI differ in some significant ways in providing the address binding.

In TLI, based on its rules, a provider determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint. The application program must check the bound address to ensure that it is the one previously advertised to clients. In XTI, if the provider determines it cannot bind to the requested address, it fails the `t_bind( )` request with an error.

If `t_bind( )` succeeds, the provider begins queueing connect requests, entering the next phase of communication.

Connection Establishment

XTI/TLI imposes different procedures in this phase for clients and servers. The client starts connection establishment by requesting a connection to a specified server using `t_connect( )`. The server receives a client's request by calling `t_listen( )`. The server must accept or reject the client's request. It calls `t_accept( )` to establish the connection, or `t_snddis( )` to reject the request. The client is notified of the result when `t_connect( )` returns.

TLI supports two facilities during connection establishment that might not be supported by all transport providers:

- Data transfer between the client and server when establishing the connection. The client can send data to the server when it requests a connection. This data is passed to the server by `t_listen( )`. The server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by `t_open( )` determines how much data, if any, two users can transfer during connect establishment.
- The negotiation of protocol options. The client can specify preferred protocol options to the transport provider and/or the remote user. XTI/TLI supports both local and remote option negotiation. Option negotiation is a protocol-specific capability.

These facilities produce protocol-dependent software (see "Guidelines to Protocol Independence" on page 100).

Client

The steps for the client to establish a connection are shown in Code Example 3–5.

CODE EXAMPLE 3–5 Client-to-Server Connection

```
if ((sndcall = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR))
      == (struct t_call *) NULL) {
   t_error("t_alloc failed");
   exit(3);
}

/*
 * Because it assumes it knows the format of the provider's
 * address, this program is transport-dependent
 */
sndcall->addr.len = sizeof(int);
*(int *) sndcall->addr.buf = SRV_ADDR;
if (t_connect( fd, sndcall, (struct t_call *) NULL) == -1 ) {
   t_error("t_connect failed for fd");
   exit(4);
}
```

The `t_connect( )` call connects to the server. The first argument of `t_connect( )` identifies the client's endpoint, and the second argument points to a `t_call` structure that identifies the destination server. This structure has the following format:

```
struct t_call {
  struct netbuf addr;
  struct netbuf opt;
  struct netbuf udata;
  int sequence;
}
```

`addr` identifies the address of the server, `opt` specifies protocol-specific options to the connection, and `udata` identifies user data that can be sent with the connect request to the server. The `sequence` field has no meaning for `t_connect( )`. In this example, only the server's address is passed.

`t_alloc( )` allocates the `t_call` structure dynamically. The third argument of `t_alloc( )` is `T_ADDR`, which specifies that the system needs to allocate a `netbuf` buffer. The server's address is then copied to `buf`, and `len` is set accordingly.

The third argument of `t_connect( )` can be used to return information about the newly established connection, and can return any user data sent by the server in its response to the connect request. The third argument here is set to `NULL` by the client. The connection is established on successful return of `t_connect( )`. If the server rejects the connect request, `t_connect( )` sets `t_errno` to `TLOOK`.

Event Handling

The TLOOK error has special significance. TLOOK is set if an XTI/TLI routine is interrupted by an unexpected asynchronous transport event on the endpoint. TLOOK does not report an error with an XTI/TLI routine, but the normal processing of the routine is not done because of the pending event. The events defined by XTI/TLI are listed in Table 3–7.

TABLE 3–7 Asynchronous Endpoint Events

Name	Description
T_LISTEN	Connection request arrived at the transport endpoint
T_CONNECT	Confirmation of a previous connect request arrived (generated when a server accepts a connect request)
T_DATA	User data has arrived
T_EXDATA	Expedited user data arrived
T_DISCONNECT	Notice that an aborted connection or a rejected connect request arrived
T_ORDREL	A request for orderly release of a connection arrived
T_UDERR	Notice of an error in a previous datagram arrived. (See “Read/Write Interface” on page 84.)

The state table in “State Transitions” on page 93 shows which events can happen in each state. t_look( ) lets a user determine what event has occurred if a TLOOK error is returned. In the example, if a connect request is rejected, the client exits.

Server

When the client calls t_connect( ), a connect request is sent at the server’s transport endpoint. For each client, the server accepts the connect request and spawns a process to service the connection.

```
if ((call = (struct t_call *) t_alloc(listen_fd, T_CALL, T_ALL))
      == (struct t_call *) NULL) {
   t_error("t_alloc of t_call structure failed");
   exit(5);
```

(continued)

(Continuation)

```
}
while(1) {
   if (t_listen( listen_fd, call) == –1) {
      t_error("t_listen failed for listen_fd");
      exit(6);
   }
   if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
      run_server(listen_fd);
}
```

The server allocates a `t_call` structure, then does a closed loop. The loop blocks on `t_listen( )` for a connect request. When a request arrives, the server calls `accept_call( )` to accept the connect request. `accept_call( )` accepts the connection on an alternate transport endpoint (as discussed below) and returns the handle of that endpoint. (*conn_fd* is a global variable.) Because the connection is accepted on an alternate endpoint, the server can continue to listen on the original endpoint. If the call is accepted without error, `run_server( )` spawns a process to service the connection.

Note - XTI/TLI supports an asynchronous mode for these routines that prevents a process from blocking. See "Advanced Topics" on page 86.

When a connect request arrives, the server calls `accept_call( )` to accept the client's request, as Code Example 3–6 shows.

Note - It is implicity assumed that this server only needs to handle a single connection request at a time. This is not normally true of a server. The code required to handle multiple simultaneous connection requests is complicated because of XTI/TLI event mechanisms. See "Advanced Programming Example" on page 87 for such a server.

CODE EXAMPLE 3–6 `accept_call( )` Function

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
   int resfd;

   if ((resfd = t_open("/dev/exmp", O_RDWR, (struct t_info *) NULL))
         == –1) {
      t_error("t_open for responding fd failed");
      exit(7);
```

(continued)

(Continuation)

```
  }
   if (t_bind(resfd,(struct t_bind *) NULL, (struct t_bind *NULL))
         == –1) {
      t_error("t_bind for responding fd failed");
      exit(8);
   }
   if (t_accept(listen_fd, resfd, call) == –1) {
      if (t_errno == TLOOK) {        /* must be a disconnect */
         if (t_rcvdis(listen_fd,(struct t_discon *) NULL) == –1) {
            t_error("t_rcvdis failed for listen_fd");
            exit(9);
         }
         if (t_close(resfd) == –1) {
            t_error("t_close failed for responding fd");
            exit(10);
         }
         /* go back up and listen for other calls */
         return(DISCONNECT);
      }
      t_error("t_accept failed");
      exit(11);
   }
   return(resfd);
}
```

`accept_call()` has two arguments:

listen_fd	The file handle of the transport endpoint where the connect request arrived.
call	Points to a `t_call` structure that contains all information associated with the connect request

The server first opens another transport endpoint by opening the clone device special file of the transport provider and binding an address. A `NULL` specifies not to return the address bound by the provider. The new transport endpoint, *resfd*, accepts the client's connect request.

The first two arguments of `t_accept()` specify the listening transport endpoint and the endpoint where the connection is accepted, respectively. Accepting a connection on the listening endpoint prevents other clients from accessing the server for the duration of the connection.

The third argument of `t_accept()` points to the `t_call` structure containing the connect request. This structure should contain the address of the calling user and the sequence number returned by `t_listen()`. The sequence number is significant if the server queues multiple connect requests. The"Advanced Topics" on page 86 shows an example of this. The `t_call` structure also identifies protocol options and user data to pass to the client. Because this transport provider does not support

protocol options or the transfer of user data during connection, the `t_call` structure returned by `t_listen( )` is passed without change to `t_accept( )`.

The example is simplified. The server exits if either the `t_open( )` or `t_bind( )` call fails. `exit( )` closes the transport endpoint of `listen_fd`, causing a disconnect request to be sent to the client. The client's `t_connect( )` call fails, setting `t_errno` to `TLOOK`.

`t_accept( )` can fail if an asynchronous event occurs on the listening endpoint before the connection is accepted, and `t_errno` is set to `TLOOK`. Table 3–8 shows that only a disconnect request can be sent in this state with only one queued connect request. This event can happen if the client undoes a previous connect request. If a disconnect request arrives, the server must respond by calling `t_rcvdis( )`. This routine argument is a pointer to a `t_discon` structure, which is used to retrieve the data of the disconnect request. In this example, the server passes a `NULL`.

After receiving a disconnect request, `accept_call( )` closes the responding transport endpoint and returns `DISCONNECT`, which informs the server that the connection was disconnected by the client. The server then listens for further connect requests.

Figure 3–4 illustrates how the server establishes connections:

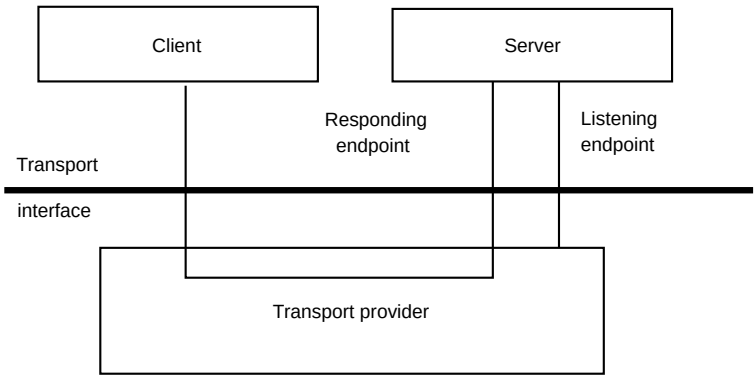

Figure 3–4 Listening and Responding Transport Endpoints

The transport connection is established on the new responding endpoint, and the listening endpoint is freed to retrieve further connect requests.

Data Transfer

After the connection is established, both the client and the server can transfer data through the connection using `t_snd( )` and `t_rcv( )`. XTI/TLI does not differentiate the client from the server from this point on. Either user can send data, receive data, or release the connection.

There are two classes of data on a transport connection:

1. Normal data
2. Expedited data

Expedited data is for urgent data. The exact semantics of expedited data vary between transport providers. Not all transport protocols support expedited data (see **t_open**(3N)).

Most connection-oriented mode protocols transfer data in byte streams. “Byte stream” implies no message boundaries in data sent over a connection. Some transport protocols preserve message boundaries over a transport connection. This service is supported by XTI/TLI, but protocol-independent software must not rely on it.

The message boundaries are invoked by the `T_MORE` flag of `t_snd( )` and `t_rcv( )`. The messages, called transport service data units (TSDU), can be transferred between two transport users as distinct units. The maximum message size is defined by the underlying transport protocol. Get the message size through `t_open( )` or `t_getinfo( )`.

You can send a message in multiple units. Set the `T_MORE` flag on every `t_snd( )` call, except the last to send a message in multiple units. The flag specifies that the data in the current and the next `t_snd( )` calls are a logical unit. Send the last message unit with `T_MORE` turned off to specify the end of the logical unit.

Similarly, a logical unit can be sent in multiple units. If `t_rcv( )` returns with the `T_MORE` flag set, the user must call `t_rcv( )` again to receive the rest of the message. The last unit in the message is identified by a call to `t_rcv( )` that does not set `T_MORE`.

The `T_MORE` flag implies nothing about how the data is packaged below XTI/TLI or how the data is delivered to the remote user. Each transport protocol, and each implementation of a protocol, can package and deliver the data differently.

For example, if a user sends a complete message in a single call to `t_snd( )`, there is no guarantee that the transport provider delivers the data in a single unit to the receiving user. Similarly, a message transmitted in two units can be delivered in a single unit to the remote transport user.

If supported by the transport, the message boundaries are preserved only by setting the value of `T_MORE` for `t_snd( )` and testing it after `t_rcv( )`. This guarantees that the receiver sees a message with the same contents and message boundaries as was sent.

Client

The example server transfers a log file to the client over the transport connection. The client receives the data and writes it to its standard output file. A byte stream interface is used by the client and server, with no message boundaries. The client receives data by the following:

```
while ((nbytes = t_rcv(fd, buf, nbytes, &flags))!= –1){
   if (fwrite(buf, 1, nbytes, stdout) == –1) {
      fprintf(stderr, "fwrite failed\n");
      exit(5);
   }
}
```

The client repeatedly calls `t_rcv( )` to receive incoming data. `t_rcv( )` blocks until data arrives. `t_rcv( )` writes up to *nbytes* of the data available into *buf* and returns the number of bytes buffered. The client writes the data to standard output and continues. The data transfer loop ends when `t_rcv( )` fails. `t_rcv( )` fails when an orderly release or disconnect request arrives. If `fwrite( )` fails for any reason, the client exits, which closes the transport endpoint. If the transport endpoint is closed (either by `exit( )` or `t_close( )`) during data transfer, the connection is aborted and the remote user receives a disconnect request.

Server

The server manages its data transfer by spawning a child process to send the data to the client. The parent process continues the loop to listen for more connect requests. `run_server( )` is called by the server to spawn this child process, as shown in Code Example 3–7.

CODE EXAMPLE 3–7 Spawning Child Process to Loopback and Listen

```
connrelease()
{
   /* conn_fd is global because needed here */
   if (t_look(conn_fd) == T_DISCONNECT) {
      fprintf(stderr, ‘‘connection aborted\n’’);
      exit(12);
   }
   /* else orderly release request - normal exit */
   exit(0);
}
run_server(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp;                    /* file pointer to log file */
   char buf[1024];

   switch(fork()) {
   case –1:
      perror("fork failed");
      exit(20);
   default:         /* parent */
      /* close conn_fd and then go up and listen again*/
      if (t_close(conn_fd) == –1) {
```

(continued)

(Continuation)

```
        t_error("t_close failed for conn_fd");
        exit(21);
    }
    return;
case 0:                         /* child */
    /* close listen_fd and do service */
    if (t_close(listen_fd) == -1) {
        t_error("t_close failed for listen_fd");
        exit(22);
    }
    if ((logfp = fopen("logfile", "r")) == (FILE *) NULL) {
        perror("cannot open logfile");
        exit(23);
    }
    signal(SIGPOLL, connrelease);
    if (ioctl(conn_fd, I_SETSIG, S_INPUT) == -1) {
        perror("ioctl I_SETSIG failed");
        exit(24);
    }
    if (t_look(conn_fd) != 0){      /*disconnect there?*/
        fprintf(stderr, "t_look: unexpected event\n");
        exit(25);
    }
    while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
        if (t_snd(conn_fd, buf, nbytes, 0) == -1) {
            t_error("t_snd failed");
            exit(26);
        }
```

After the fork, the parent process returns to the main listening loop. The child process manages the newly established transport connection. If the fork fails, `exit()` closes both transport endpoints, sending a disconnect request to the client, and the client's `t_connect()` call fails.

The server process reads 1024 bytes of the log file at a time and sends the data to the client using `t_snd()`. *buf* points to the start of the data buffer, and *nbytes* specifies the number of bytes to transmit. The fourth argument can be zero or one of the two optional flags below:

- `T_EXPEDITED` specifies that the data is expedited.
- `T_MORE` specifies that the next block continues the message in this block.

Neither flag is set by the server in this example.

If the user floods the transport provider with data, `t_snd()` blocks until enough data is removed from the transport.

`t_snd()` does not look for a disconnect request (showing that the connection was broken). If the connection is aborted, the server should be notified, since data can be lost. One solution is to call `t_look()` to check for incoming events before each

`t_snd( )` call or after a `t_snd( )` failure. The example has a cleaner solution. The `I_SETSIG ioctl( )` lets a user request a signal when a specified event occurs. See the **`streamio`**(7I) manpage. `S_INPUT` causes a signal to be sent to the user process when any input arrives at the endpoint `conn_fd`. If a disconnect request arrives, the signal-catching routine (`connrelease( )`) prints an error message and exits.

If the server alternates `t_snd( )` and `t_rcv( )` calls, it can use `t_rcv( )` to recognize an incoming disconnect request.

Connection Release

At any time during data transfer, either user can release the transport connection and end the conversation. There are two forms of connection release.

- The first way, abortive release, breaks the connection immediately and discards any data that has not been delivered to the destination user.

  Either user can call `t_snddis( )` to perform an abortive release. The transport provider can abort a connection if a problem occurs below XTI/TLI. `t_snddis( )` lets a user send data to the remote user when aborting a connection. The abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

  When the remote user is notified of the aborted connection, call `t_rcvdis( )` to receive the disconnect request. The call returns a code that identifies why the connection was aborted, and returns any data that can have accompanied the disconnect request (if the abort was initiated by the remote user). The reason code is specific to the underlying transport protocol, and should not be interpreted by protocol-independent software.

- The second way, orderly release, ends a connection so that no data is lost. All transport providers must support the abortive release procedure, but orderly release is an option not supported by all connection-oriented protocols.

See “Transport Selection” on page 108for information on how to select a transport that supports orderly release.

Server

This example assumes that the transport provider supports orderly release. When all the data has been sent by the server, the connection is released as follows:

```
if (t_sndrel(conn_fd) == –1) {
   t_error(‘‘t_sndrel failed’’);
   exit(27);
}
```

(continued)

(Continuation)

```
pause(); /* until orderly release request arrives */
```

Orderly release requires two steps by each user. The server can call `t_sndrel( )`. This routine sends a disconnect request. When the client receives the request, it can continue sending data back to the server. When all data have been sent, the client calls `t_sndrel( )` to send a disconnect request back. The connection is released only after both users have received a disconnect request.

In this example, data is transferred only from the server to the client. So there is no provision to receive data from the client after the server initiates release. The server calls `pause( )` after initiating the release.

The client responds with its orderly release request, which generates a signal caught by `connrelease( )`. (In Code Example 3–7, the server issued an `I_SETSIG ioctl( )` call to generate a signal on any incoming event.) The only XTI/TLI event possible in this state is a disconnect request or an orderly release request, so `connrelease( )` exits normally when the orderly release request arrives. `exit( )` from `connrelease( )` closes the transport endpoint and frees the bound address. To close a transport endpoint without exiting, call `t_close( )`.

Client

The client releases the connection similar to the way the server releases it. The client processes incoming data until `t_rcv( )` fails. When the server releases the connection (using either `t_snddis( )` or `t_sndrel( )`), `t_rcv( )` fails and sets `t_errno` to `TLOOK`. The client then processes the connection release as follows:

```
if ((t_errno == TLOOK) && (t_look(fd) == T_ORDREL)) {
   if (t_rcvrel(fd) == –1) {
      t_error(‘‘t_rcvrel failed’’);
      exit(6);
  }
  if (t_sndrel(fd) == –1) {
      t_error(‘‘t_sndrel failed’’);
      exit(7);
  }
  exit(0);
 }
```

Each event on the client’s transport endpoint is checked for an orderly release request. When one is received, the client calls `t_rcvrel( )` to process the request and `t_sndrel( )` to send the response release request. The client then exits, closing its transport endpoint.

If a transport provider does not support the orderly release, use abortive release with `t_snddis( )` and `t_rcvdis( )`. Each user must take steps to prevent data loss. For example, use a special byte pattern in the data stream to indicate the end of a conversation.

Read / Write Interface

A user might want to establish a transport connection using `exec( )` on an existing program (such as `/usr/bin/cat`) to process the data as it arrives over the connection. Existing programs use `read( )` and `write( )`. XTI / TLI does not directly support a read / write interface to a transport provider, but one is available. The interface lets you issue `read( )` and `write( )` calls over a transport connection in the data transfer phase. This section describes the read / write interface to the connection mode service of XTI / TLI. This interface is not available with the connectionless mode service.

The read / write interface is presented using the client example of “Connection Mode Service” on page 67 with modifications. The clients are identical until the data transfer phase. Then the client uses the read / write interface and `cat` to process incoming data. `cat` is run without change over the transport connection. Only the differences between this client and that of the client in Code Example 3–3 are shown in Code Example 3–8.

CODE EXAMPLE 3–8 Read / Write Interface

```
#include <stropts.h>
   .
   ./*
     Same local management and connection establishment steps.
     */
   .
   if (ioctl(fd, I_PUSH, "tirdwr") == –1) {
      perror(‘‘I_PUSH of tirdwr failed’’);
      exit(5);
  }
   close(0);
   dup(fd);
   execl(‘‘/usr/bin/cat’’, ‘‘/usr/bin/cat’’, (char *) 0);
   perror(‘‘exec of /usr/bin/cat failed’’);
   exit(6);
}
```

The client invokes the read / write interface by pushing `tirdwr` onto the stream associated with the transport endpoint. See `I_PUSH` in **`streamio`**(7I). `tirdwr`

converts XTI/TLI above the transport provider into a pure read/write interface. With the module in place, the client calls `close()` and `dup()` to establish the transport endpoint as its standard input file, and uses `/usr/bin/cat` to process the input.

By pushing `tirdwr` onto the transport provider, XTI/TLI is changed. The semantics of `read()` and `write()` must be used, and message boundaries are not preserved. `tirdwr` can be popped from the transport provider to restore XTI/TLI semantics (see `I_POP` in **`streamio`**(7I).

Caution - The `tirdwr` module can only be pushed onto a stream when the transport endpoint is in the data transfer phase. After the module is pushed, the user cannot call any XTI/TLI routines. If an XTI/TLI routine is invoked, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream, rendering it unusable. If you then pop the `tirdwr` module off the stream, the transport connection is aborted. See `I_POP` in **`streamio`**(7I).

Write

Send data over the transport connection with `write()`. `tirdwr` passes data through to the transport provider. If you send a zero-length data packet, which the mechanism allows, `tirdwr` discards the message. If the transport connection is aborted—for example, because the remote user aborts the connection using `t_snddis()`—a hang-up condition is generated on the stream, further `write()` calls fail, and `errno` is set to `ENXIO`. You can still retrieve any available data after a hang-up.

Read

Receive data that arrives at the transport connection with `read()`. `tirdwr`, which passes data from the transport provider. Any other event or request passed to the user from the provider is processed by `tirdwr` as follows:

- `read()` cannot identify expedited data to the user. If an expedited data request is received, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream. The error causes further system calls to fail. Do not use `read()` to receive expedited data.
- `tirdwr` discards an abortive disconnect request and generates a hang-up condition on the stream. Subsequent `read()` calls retrieve any remaining data, then return zero for all further calls (indicating end of file).
- `tirdwr` discards an orderly release request and delivers a zero-length message to the user. As described in `read()`, this notifies the user of end of file by returning 0.
- If any other XTI/TLI request is received, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream. This causes further system calls to fail. If a user pushes

`tirdwr` onto a stream after the connection has been established, no request is generated.

Close

With `tirdwr` on a stream, you can send and receive data over a transport connection for the duration of the connection. Either user can terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the `tirdwr` module off the stream. In either case, `tirdwr` does the following:

- If an orderly release request was previously received by `tirdwr`, it is passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected request when data transfer completes.
- If a disconnect request was previously received by `tirdwr`, no special action is taken.
- If neither an orderly release nor a disconnect request was previously received by `tirdwr`, a disconnect request is passed to the transport provider to abort the connection.
- If an error previously occurred on the stream and a disconnect request has not been received by `tirdwr`, a disconnect request is passed to the transport provider.

A process cannot initiate an orderly release after `tirdwr` is pushed onto a stream. `tirdwr` handles an orderly release if it is initiated by the user on the other side of a transport connection. If the client in this section is communicating with the server program in “Connection Mode Service” on page 67, the server terminates the transfer of data with an orderly release request. The server then waits for the corresponding request from the client. At that point, the client exits and the transport endpoint is closed. When the file descriptor is closed, `tirdwr` initiates the orderly release request from the client’s side of the connection. This generates the request that the server is blocked on.

Some protocols, like TCP, require this orderly release to ensure that the data is delivered intact.

Advanced Topics

This section presents additional XTI/TLI concepts:

- An optional nonblocking (asynchronous) mode for some library calls
- How to set and get TCP and UDP options under XTI/TLI

- A program example of a server supporting multiple outstanding connect requests and operating in an event-driven manner

Asynchronous Execution Mode

Many XTI/TLI library routines block to wait for an incoming event. However, some time-critical applications should not block for any reason. An application can do local processing while waiting for some asynchronous XTI/TLI event.

Asynchronous processing of XTI/TLI events is available to applications through the combination of asynchronous features and the non-blocking mode of XTI/TLI library routines. Use of the `poll( )` system call and the `I_SETSIG ioctl` command to process events asynchronously is described in *ONC+ Developer's Guide*.

Each XTI/TLI routine that blocks for an event can be run in a special non-blocking mode. For example, `t_listen( )` normally blocks for a connect request. A server can periodically poll a transport endpoint for queued connect requests by calling `t_listen( )` in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting `O_NDELAY` or `O_NONBLOCK` in the file descriptor. These modes can be set as a flag through `t_open( )`, or by calling `fcntl( )` before calling the XTI/TLI routine. `fcntl( )` enables or disables this mode at any time. All program examples in this chapter use the default synchronous processing mode.

`O_NDELAY` or `O_NONBLOCK` affect each XTI/TLI routine differently. You will need to determine the exact semantics of `O_NDELAY` or `O_NONBLOCK` for a particular routine.

Advanced Programming Example

The following example demonstrates two important concepts. The first is a server's ability to manage multiple outstanding connect requests. The second is event-driven use of XTI/TLI and the system call interface.

The server example in Code Example 3–4 supports only one outstanding connect request, but XTI/TLI lets a server manage multiple outstanding connect requests. One reason to receive several simultaneous connect requests is to prioritize the clients. A server can receive several connect requests, and accept them in an order based on the priority of each client.

The second reason for handling several outstanding connect requests is the limits of single-threaded processing. Depending on the transport provider, while a server processes one connect request, other clients find it busy. If multiple connect requests are processed simultaneously, the server will be found busy only if more than the maximum number of clients try to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming XTI/TLI events, and takes the appropriate actions for the event received.

The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and endpoint establishment functions of Code Example 3–9 are similar to those of the server example in Code Example 3–4.

CODE EXAMPLE 3–9 Endpoint Establishment (Convertible to Multiple Connections)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 1                /* server's well known address */

int conn_fd;                      /* server connection here */
extern int t_errno;
/* holds connect requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
    * Only opening and binding one transport endpoint, but more can
    * be supported
    */
   if ((pollfds[0].fd = t_open(''/dev/tivc'', O_RDWR,
         (struct t_info *) NULL)) == -1) {
      t_error(''t_open failed'');
      exit(1);
   }
   if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
         T_ALL)) == (struct t_bind *) NULL) {
      t_error(''t_alloc of t_bind structure failed'');
      exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len = sizeof(int);
   *(int *) bind->addr.buf = SRV_ADDR;
   if (t_bind(pollfds[0].fd, bind, bind) == -1) {
      t_error(''t_bind failed'');
      exit(3);
   }
   /* Was the correct address bound? */
   if (bind->addr.len != sizeof(int) ||
      *(int *)bind->addr.buf != SRV_ADDR) {
      fprintf(stderr, ''t_bind bound wrong address\n'');
```

(continued)

(Continuation)

```
      exit(4);
   }
}
```

The file descriptor returned by `t_open()` is stored in a `pollfd` structure that controls polling the transport endpoints for incoming data. See **`poll`**(2). Only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to Code Example 3–9.

This server sets `qlen` to a value greater than `1` for `t_bind()`. This specifies that the server queues multiple outstanding connect requests. The server accepts the current connect request before accepting additional connect requests. This example can queue up to `MAX_CONN_IND` connect requests. The transport provider can negotiate the value of `qlen` smaller if it cannot support `MAX_CONN_IND` outstanding connect requests.

After the server has bound its address and is ready to process connect requests, it behaves as shown in Code Example 3–10.

CODE EXAMPLE 3–10 Processing Connection Requests

```
pollfds[0].events = POLLIN;

while (TRUE) {
 if (poll(pollfds, NUM_FDS, –1) == –1) {
   perror(‘‘poll failed’’);
   exit(5);
 }
 for (i = 0; i < NUM_FDS; i++) {
   switch (pollfds[i].revents) {
      default:
         perror(‘‘poll returned error event’’);
      exit(6);
      case 0:
         continue;
      case POLLIN:
         do_event(i, pollfds[i].fd);
         service_conn_ind(i, pollfds[i].fd);
   }
   }
}
```

The `events` field of the `pollfd` structure is set to `POLLIN`, which notifies the server of any incoming XTI/TLI events. The server then enters an infinite loop in which it polls the transport endpoint(s) for events, and processes events as they occur.

The `poll()` call blocks indefinitely for an incoming event. On return, each entry (one per transport endpoint) is checked for a new event. If `revents` is `0`, no event has occurred on the endpoint and the server continues to the next endpoint. If `revents` is `POLLIN`, there is an event on the endpoint. `do_event()` is called to process the event. Any other value in `revents` indicates an error on the endpoint, and the server exits. With multiple endpoints, it is better for the server to close this descriptor and continue.

For each iteration of the loop, `service_conn_ind()` is called to process any outstanding connect requests. If another connect request is pending, `service_conn_ind()` saves the new connect request and responds to it later.

The `do_event()` in Code Example 3–11 is called to process an incoming event.

CODE EXAMPLE 3–11 Event Processing Routine

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {
   default:
      fprintf(stderr, "t_look: unexpected event\n");
      exit(7);
   case T_ERROR:
      fprintf(stderr, "t_look returned T_ERROR event\n");
      exit(8);
   case –1:
      t_error("t_look failed");
      exit(9);
   case 0:
      /* since POLLIN returned, this should not happen */
      fprintf(stderr,"t_look returned no event\n");
      exit(10);
   case T_LISTEN:
      /* find free element in calls array */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (calls[slot][i] == (struct t_call *) NULL)
            break;
      }
      if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
               T_ALL)) == (struct t_call *) NULL) {
         t_error("t_alloc of t_call structure failed");
         exit(11);
      }
      if (t_listen(fd, calls[slot][i] ) == –1) {
         t_error("t_listen failed");
```

(continued)

(Continuation)

```
            exit(12);
         }
         break;
      case T_DISCONNECT:
         discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
         if (discon == (struct t_discon *) NULL) {
            t_error("t_alloc of t_discon structure failed");
            exit(13)
         }
         if(t_rcvdis( fd, discon) == -1) {
            t_error("t_rcvdis failed");
            exit(14);
         }
         /* find call ind in array and delete it */
         for (i = 0; i < MAX_CONN_IND; i++) {
            if (discon->sequence == calls[slot][i]->sequence) {
               t_free(calls[slot][i], T_CALL);
               calls[slot][i] = (struct t_call *) NULL;
            }
         }
         t_free(discon, T_DIS);
         break;
      }
}
```

The arguments are a number (*slot*) and a file descriptor (*fd*). *slot* is the index into the global array `calls` which has an entry for each transport endpoint. Each entry is an array of `t_call` structures that hold incoming connect requests for the endpoint.

`do_event()` calls `t_look()` to identify the XTI/TLI event on the endpoint specified by *fd*. If the event is a connect request (`T_LISTEN` event) or disconnect request (`T_DISCONNECT` event), the event is processed. Otherwise, the server prints an error message and exits.

For connect requests, `do_event()` scans the array of outstanding connect requests for the first free entry. A `t_call` structure is allocated for the entry, and the connect request is received by `t_listen()`. The array is large enough to hold the maximum number of outstanding connect requests. The processing of the connect request is deferred.

A disconnect request must correspond to an earlier connect request. `do_event()` allocates a `t_discon` structure to receive the request. This structure has the following fields:

```
struct t_discon {
  struct netbuf udata;
  int reason;
  int sequence;
}
```

udata contains any user data sent with the disconnect request. reason contains a protocol-specific disconnect reason code. sequence identifies the connect request that matches the disconnect request.

t_rcvdis( ) is called to receive the disconnect request. The array of connect requests is scanned for one that contains the sequence number that matches the sequence number in the disconnect request. When the connect request is found, its structure is freed and the entry is set to NULL.

When an event is found on a transport endpoint, service_conn_ind( ) is called to process all queued connect requests on the endpoint as Code Example 3–12 shows.

CODE EXAMPLE 3–12 Process All Connect Requests

```
service_conn_ind(slot, fd)
{
   int i;

 for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == (struct t_call *) NULL)
         continue;
      if((conn_fd = t_open( ‘‘/dev/tivc’’, O_RDWR,
            (struct t_info *) NULL)) == –1) {
         t_error("open failed");
         exit(15);
      }
      if (t_bind(conn_fd, (struct t_bind *) NULL,
            (struct t_bind *) NULL) == –1) {
         t_error("t_bind failed");
         exit(16);
      }
      if (t_accept(fd, conn_fd, calls[slot][i]) == –1) {
         if (t_errno == TLOOK) {
            t_close(conn_fd);
            return;
         }
         t_error("t_accept failed");
         exit(167);
      }
      t_free(calls[slot][i], T_CALL);
      calls[slot][i] = (struct t_call *) NULL;
      run_server(fd);
   }
}
```

For each transport endpoint, the array of outstanding connect requests is scanned. For each request, the server opens a responding transport endpoint, binds an address to the endpoint, and accepts the connection on the endpoint. If another event (connect request or disconnect request) arrives before the current request is accepted, t_accept( ) fails and sets t_errno to TLOOK. (You cannot accept an outstanding connect request if any pending connect request events or disconnect request events exist on the transport endpoint.)

If this error occurs, the responding transport endpoint is closed and `service_conn_ind( )` returns immediately (saving the current connect request for later processing). This causes the server's main processing loop to be entered, and the new event is discovered by the next call to `poll( )`. In this way, multiple connect requests can be queued by the user.

Eventually, all events are processed, and `service_conn_ind( )` is able to accept each connect request in turn. After the connection has been established, the `run_server( )` routine used by the server in the Code Example 3–5 is called to manage the data transfer.

State Transitions

These tables describe all state transitions associated with XTI/TLI. First, however, the states and events are described.

XTI/TLI States

Table 3–8 defines the states used in XTI/TLI state transitions, along with the service types.

TABLE 3–8 XTI/TLI State Transitions and Service Types

State	Description	Service Type
`T_UNINIT`	Uninitialized – initial and final state of interface	`T_COTS`, `T_COTS_ORD`, `T_CLTS`
`T_UNBND`	Initialized but not bound	`T_COTS`, `T_COTS_ORD`, `T_CLTS`
`T_IDLE`	No connection established	`T_COTS`, `T_COTS_ORD`, `T_CLTS`
`T_OUTCON`	Outgoing connection pending for client	`T_COTS`, `T_COTS_ORD`
`T_INCON`	Incoming connection pending for server	`T_COTS`, `T_COTS_ORD`
`T_DATAXFER`	Data transfer	`T_COTS`, `T_COTS_ORD`

TABLE 3–8 XTI/TLI State Transitions and Service Types *(continued)*

State	Description	Service Type
T_OUTREL	Outgoing orderly release (waiting for orderly release request	T_COTS_ORD
T_INREL	Incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Outgoing Events

The outgoing events described in Table 3–9 correspond to the status returned from the specified transport routines, where these routines send a request or response to the transport provider. In the table, some events, such as `accept()`, are distinguished by the context in which they occur. The context is based on the values of the following variables:

- *ocnt* – Count of outstanding connect requests
- *fd* – File descriptor of the current transport endpoint
- *resfd* – File descriptor of the transport endpoint where a connection is accepted

TABLE 3–9 Outgoing Events

Event	Description	Service Type
opened	Successful return of `t_open()`	T_COTS, T_COTS_ORD, T_CLTS
bind	Successful return of `t_bind()`	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	Successful return of `t_optmgmt()`	T_COTS, T_COTS_ORD, T_CLTS
unbind	Successful return of `t_unbind()`	T_COTS, T_COTS_ORD, T_CLTS
closed	Successful return of `t_close()`	T_COTS, T_COTS_ORD, T_CLT
connect1	Successful return of `t_connect()` in synchronous mode	T_COTS, T_COTS_ORD

TABLE 3–9 Outgoing Events *(continued)*

Event	Description	Service Type
connect2	TNODATA error on t_connect( ) in asynchronous mode, or TLOOK error due to a disconnect request arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	Successful return of t_accept( ) with ocnt == 1, fd == resfd	T_COTS, T_COTS_ORD
accept2	Successful return of t_accept( ) with ocnt== 1, fd!= resfd	T_COTS, T_COTS_ORD
accept3	Successful return of t_accept( ) with ocnt > 1	T_COTS, T_COTS_ORD
snd	Successful return of t_snd( )	T_COTS, T_COTS_ORD
snddis1	Successful return of t_snddis( ) with ocnt <= 1	T_COTS, T_COTS_ORD
snddis2	Successful return of t_snddis( ) with ocnt > 1	T_COTS, T_COTS_ORD
sndrel	Successful return of t_sndrel( )	T_COTS_ORD
sndudata	Successful return of t_sndudata( )	T_CLTS

Incoming Events

The incoming events correspond to the successful return of the specified routines. These routines return data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is pass_conn, which occurs when a connection is transferred to another endpoint. The event occurs on the endpoint that is being passed the connection, although no XTI/TLI routine is called on the endpoint.

In Table 3–10, the rcvdis events are distinguished by the value of ocnt, the count of outstanding connect requests on the endpoint.

TABLE 3–10 Incoming Events

Event	Description	Service Type
`listen`	Successful return of `t_listen( )`	`T_COTS`, `T_COTS_ORD`
`rcvconnect`	Successful return of `t_rcvconnect( )`	`T_COTS`, `T_COTS_ORD`
`rcv`	Successful return of `t_rcv( )`	`T_COTS`, `T_COTS_ORD`
`rcvdis1`	Successful return of `rcvdis1t_rcvdis( )`, `onct <= 0`	`T_COTS`, `T_COTS_ORD`
`rcvdis2`	Successful return of `t_rcvdis( )`, `ocnt == 1`	`T_COTS`, `T_COTS_ORD`
`rcvdis3`	Successful return of `t_rcvdis( )` with `ocnt > 1`	`T_COTS`, `T_COTS_ORD`
`rcvrel`	Successful return of `t_rcvrel( )`	`T_COTS_ORD`
`rcvudata`	Successful return of `t_rcvudata( )`	`T_CLTS`
`rcvuderr`	Successful return of `t_rcvuderr( )`	`T_CLTS`
`pass_conn`	Receive a passed connection	`T_COTS`, `T_COTS_ORD`

Transport User Actions

Some state transitions (below) have a list of actions the transport user must take. Each action is represented by a digit from the list below:

- Set the count of outstanding connect requests to zero.
- Increment the count of outstanding connect requests.
- Decrement the count of outstanding connect requests.
- Pass a connection to another transport endpoint, as indicated in `t_accept( )`.

State Tables

The tables describe the XTI/TLI state transitions. Each box contains the next state, given the current state (column) and the current event (row). An empty box is an invalid state/event combination. Each box can also have an action list. Actions must be done in the order specified in the box.

The following should be understood when studying the state tables:

- `t_close()` causes an established connection to be terminated for a connection-oriented transport provider. The connection termination will be orderly or abortive, depending on the service type supported by the transport provider. See **`t_getinfo`**(3N).
- If a transport user issues a function out of sequence, the function fails and `t_errno` is set to `TOUTSTATE`. The state does not change.
- The error codes `TLOOK` or `TNODATA` after `t_connect()` can result in state changes described in "Event Handling" on page 75. The state tables assume correct use of XTI/TLI.
- Any other transport error does not change the state, unless the manual page for the function says otherwise.
- The support functions `t_getinfo()`, `t_getstate()`, `t_alloc()`, `t_free()`, `t_sync()`, `t_look()`, and `t_error()` are excluded from the state tables because they do not affect the state.

Table 3–11, Table 3–12, Table 3–13, and Table 3–14 show endpoint establishment, data transfer in connectionless mode, and connection establishment/connection release/data transfer in connection mode.

TABLE 3–11 Connection Establishment State

Event/State	T_UNINIT	T_UNBND	T_IDLE
`opened`	T_UNBND		
`bind`		T_IDLE[1]	
`optmgmt` (TLI only)			T_IDLE
`unbind`			T_UNBND
`closed`		T_UNINIT	

TABLE 3–12 Connection Mode State—Part 1

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
connect1	T_DATAXFER			
connect2	T_OUTCON			
rcvconnect		T_DATAXFER		
listen	T_INCON [2]		T_INCON [2]	
accept1			T_DATAXFER [3]	
accept2			T_IDLE [3] [4]	
accept3			T_INCON [3] [4]	
snd				T_DATAXFER
rcv				T_DATAXFER
snddis1		T_IDLE	T_IDLE [3]	T_IDLE
snddis2			T_INCON [3]	
rcvdis1		T_IDLE		T_IDLE
rcvdis2			T_IDLE [3]	
rcvdis3			T_INCON [3]	
sndrel				T_OUTREL
rcvrel				T_INREL
pass_conn	T_DATAXFER			

TABLE 3–12 Connection Mode State—Part 1 *(continued)*

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

TABLE 3–13 Connection Mode State—Part 2

Event/State	T_OUTREL	T_INREL	T_UNBND
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd		T_INREL	
rcv	T_OUTREL		
snddis1	T_IDLE	T_IDLE	
snddis2			
rcvdis1	T_IDLE	T_IDLE	
rcvdis2			
rcvdis3			

TABLE 3–13 Connection Mode State—Part 2 *(continued)*

Event/State	T_OUTREL	T_INREL	T_UNBND
`sndrel`		T_IDLE	
`rcvrel`	T_IDLE		
`pass_conn`			T_DATAXFER
`optmgmt`	T_OUTREL	T_INREL	T_UNBND
`closed`	T_UNINIT	T_UNINIT	

TABLE 3–14 Connectionless Mode State

Event/State	T_IDLE
`snudata`	T_IDLE
`rcvdata`	T_IDLE
`rcvuderr`	T_IDLE

Guidelines to Protocol Independence

XTI/TLI's set of services, common to many transport protocols, offers protocol independence to applications. Not all transport protocols support all XTI/TLI services. If software must run in a variety of protocol environments, use only the common services. The following is a list of services that might not be common to all transport protocols.

1. In connection mode service, a transport service data unit (TSDU) might not be supported by all transport providers. Make no assumptions about preserving logical data boundaries across a connection.

2. Protocol and implementation specific service limits are returned by the `t_open()` and `t_getinfo()` routines. Use these limits to allocate buffers to store protocol-specific transport addresses and options.
3. Do not send user data with connect requests or disconnect requests, such as `t_connect()` and `t_snddis()`. Not all transport protocols work this way.
4. The buffers in the `t_call` structure used for `t_listen()` must be large enough to hold any data sent by the client during connection establishment. Use the `T_ALL` argument to `t_alloc()` to set maximum buffer sizes to store the address, options, and user data for the current transport provider.
5. Do not specify a protocol address on `t_bind()` on a client side endpoint. Let the transport provider assign an appropriate address to the transport endpoint. A server should retrieve its address for `t_bind()` in such a way that it does not require knowledge of the transport provider's name space.
6. Do not make assumptions about formats of transport addresses. Transport addresses should not be constants in a program. Chapter 4 contains detailed information.
7. The reason codes associated with `t_rcvdis()` are protocol-dependent. Do not interpret this information if protocolindependence is important.
8. The `t_rcvuderr()` error codes are protocol dependent. Do not interpret this information if protocol independence is a concern.
9. Do not code the names of devices into programs. The device node identifies a particular transport provider and is not protocol independent. See Chapter 4 for details.
10. Do not use the optional orderly release facility of the connection mode service—provided by `t_sndrel()` and `t_rcvrel()`—in programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. Its use can prevent programs from successfully communicating with open systems.

XTI/TLI Versus Socket Interfaces

XTI/TLI and sockets are different methods of handling the same tasks. Mostly, they provide mechanisms and services that are functionally similar. They do not provide one-to-one compatibility of routines or low-level services. Observe the similarities and differences between the XTI/TLI and socket-based interfaces before you decide to port an application.

The following issues are related to transport independence, and can have some bearing on RPC applications:

- *Privileged ports* – Privileged ports are an artifact of the Berkeley Software Distribution (BSD) implementation of the TCP/IP Internet Protocols. They are not

portable. The notion of privileged ports is not supported in the transport-independent environment.

- *Opaque addresses* – There is no transport-independent way of separating the portion of an address that names a host from the portion of an address that names the service at that host. Be sure to change any code that assumes it can discern the host address of a network service.
- *Broadcast* – There is no transport-independent form of broadcast address.

Socket-to-XTI/TLI Equivalents

Table 3–15 shows approximate equivalents between XTI/TLI functions and socket functions. The comment field describes the differences. If there is no comment, either the functions are similar or there is no equivalent function in either interface.

TABLE 3–15 TLI and Socket Equivalent Functions

TLI function	Socket function	Comments
`t_open()`	`socket()`	
`--`	`socketpair()`	
`t_bind()`	`bind()`	`t_bind()` sets the queue depth for passive sockets, but `bind()` doesn't. For sockets, the queue length is specified in the call to `listen()`.
`t_optmgmt()`	`getsockopt()` `setsockopt()`	`t_optmgmt()` manages only transport options. `getsockopt()` and `setsockopt()` can manage options at the transport layer, but also at the socket layer and at the arbitrary protocol layer.
`t_unbind()`		`--`
`t_close()`	`close()`	
`t_getinfo()`	`getsockopt()`	`t_getinfo()` returns information about the transport. `getsockopt()` can return information about the transport and the socket.
`t_getstate()`	`-`	

TABLE 3–15 TLI and Socket Equivalent Functions *(continued)*

TLI function	Socket function	Comments
t_sync( )	-	
t_alloc( )	-	
t_free( )	-	
t_look( )	-	getsockopt( ) with the SO_ERROR option returns the same kind of error information as t_look( ).
t_error( )	perror( )	
t_connect( )	connect( )	A connect( ) can be done without first binding the local endpoint. The endpoint must be bound before calling t_connect( ). A connect( ) can be done on a connectionless endpoint to set the default destination address for datagrams. Data can be sent on a connect( ).
t_rcvconnect( )	-	
t_listen( )	listen( )	t_listen( ) waits for connection indications. listen( ) merely sets the queue depth.
t_accept( )	accept( )	
t_snd( )	send( )	
	sendto( )	
	sendmsg( )	sendto( ) and sendmsg( ) operate in connection mode as well as datagram mode.
t_rcv( )	recv( )	
	recvfrom( )	
	recvmsg( )	recvfrom( ) and recvmsg( ) operate in connection mode as well as datagram mode.

TABLE 3–15 TLI and Socket Equivalent Functions *(continued)*

TLI function	Socket function	Comments
`t_snddis( )`	-	
`t_rcvdis( )`	-	
`t_sndrel( )`	`shutdown( )`	
`t_rcvrel( )`	-	
`t_sndudata( )`	`sendto( )`	
	`recvmsg( )`	
`t_rcvuderr( )`	-	
`read( )`, `write( )`	`read( )`, `write( )`	In XTI/TLI you must push the `tirdwr( )` module before calling `read( )` or `write( )`; in sockets, just call `read( )` or `write( )`.

Additions to XTI Interface

The XNS 5 (Unix98) standard introduces some new XTI interfaces. These are briefly described below. The details may be found in the relevant manual pages. These interfaces are not available for TLI users.

Scatter/Gather Data Transfer Interfaces

`t_sndvudata(3N)`	Send a data unit from one or more non-contiguous buffers
`t_rcvvudata(3N)`	Receive a data unit into one or more non-contiguous buffers

t_sndv(3N)	Send data or expedited data from one or more non-contiguous buffers on a connection
t_rcvv(3N)	Receive data or expedited data sent over a connection and put the data into one or more non-contiguous buffers

XTI Utility Functions

t_sysconf(3N)	Get configurable XTI variables

Additional Connection Release Interfaces

t_sndreldata(3N)	Initiate / respond to an orderly release with user data
t_rcvreldata(3N)	Receive an orderly release indication or confirmation containing user data

---

Note - The additional interfaces **t_sndreldata**(3N) and **t_rcvreldata**(3N) are only for use with a specific transport called “minimal OSI” which is not available on the Solaris platform. These interfaces are not available for use in conjunction with Internet Transports (TCP or UDP).

CHAPTER 4

Transport Selection and Name-to-Address Mapping

This chapter describes selecting transports and resolving network addresses. It further describes interfaces that enable you to specify the available communication protocols for an application. The chapter also explains additional functions that provide direct mapping of names to network addresses.

Note - In this chapter the terms *network* and *transport* are used interchangeably to refer to the programmatic interface that conforms to the transport layer of the OSI Reference Mode. The term network is also used to refer to the physical collection of computers connected through some electronic medium.

Transport Selection Is Multithread Safe

The interface described in this chapter is multithread safe. This means that applications that contain transport selection function calls can be used freely in a multithreaded application. Note, however, that the degree of concurrency available to applications is not specified.

Transport Selection

A distributed application must use a standard interface to the transport services to be portable to different protocols. Transport selection services provide an interface that allows an application to select which protocols to use. This makes an application "protocol" and "medium" independent.

Transport selection makes it easy for a client application to try each available transport until it establishes communication with a server. Transport selection lets server applications accept requests on multiple transports, and in doing so, communicate over a number of protocols. Transports can be tried in either the order specified by the local default sequence or in an order specified by the user.

Choosing from the available transports is the responsibility of the application. The transport selection mechanism makes that selection uniform and simple.

How Transport Selection Works

The transport selection component is built around:

- A network configuration database (the `/etc/netconfig` file), which contains an entry for each network on the system
- Optional use of the `NETPATH` environment variable

The `NETPATH` variable is set by the user; it contains an ordered list of transport identifiers. The transport identifiers match the `netconfig` network ID field and are links to records in the `netconfig` file. The `netconfig` file is described in "`/etc/netconfig` File" on page 109. The network selection interface is a set of access routines for the network-configuration database.

One set of library routines accesses only the `/etc/netconfig` entries identified by the `NETPATH` environment variable:

`setnetpath( )`	Initializes the search of `NETPATH`
`getnetpath( )`	Returns a pointer to the `netconfig` entry that corresponds to the next component of the `NETPATH` variable
`endnetpath( )`	Releases the database pointer to elements in the `NETPATH` variable when processing is complete

These routines are described in "`NETPATH` Access to `netconfig` Data" on page 112 and in **`getnetpath`**(3N). They let the user influence the selection of transports used by the application.

To avoid user influence on transport selection, use the routines that access the `netconfig` database directly. These routines are described in "Accessing `netconfig`" on page 113 and in **`getnetconfig`**(3N):

`setnetconfig( )`	Initializes the record pointer to the first index in the database
`getnetconfig( )`	Returns a pointer to the current record in the `netconfig` database and increments the pointer to the next record
`endnetconfig( )`	Releases the database pointer when processing is complete

The following two routines manipulate `netconfig` entries and the data structures they represent. These routines are described in "Accessing `netconfig`" on page 113:

`getnetconfigent( )`	Returns a pointer to the `struct netconfig` structure corresponding to `netid`
`freenetconfigent( )`	Frees the structure returned by `getnetconfigent( )`

`/etc/netconfig` File

The `netconfig` file describes all transport protocols on a host. The entries in the `netconfig` file are explained briefly in Table 4–1 and in more detail in the **`netconfig`**(4) man page.

TABLE 4–1 The `netconfig` File

Entries	Description
`network ID`	A local representation of a transport name (such as `tcp`). Do not assume that this field contains a well-known name (such as `tcp` or `udp`) or that two systems use the same name for the same transport.
`semantics`	The semantics of the particular transport protocol. Valid semantics are: ■ `tpi_clts` – connectionless ■ `tpi_cots` – connection oriented ■ `tpi_cots_ord` – connection oriented with orderly release
`flags`	Can take only the values, `v`, or hyphen (`-`). Only the visible flag (`-v`) is defined.
`protocol family`	The protocol family name of the transport provider (for example, `inet` or `loopback`).

TABLE 4–1 The `netconfig` File *(continued)*

Entries	Description
`protocol name`	The protocol name of the transport provider. For example, if *protocol family* is `inet`, then *protocol name* is `tcp`, `udp`, or `icmp`. Otherwise, the value of *protocol name* is a hyphen (`-`).
`network device`	The full path name of the device file to open when accessing the transport provider
`name-to-address translation libraries`	Names of the shared objects. This field contains the comma-separated file names of the shared objects that contain name-to-address mapping routines. Shared objects are located through the path in the `LD_LIBRARY_PATH` variable. A "`-`" in this field indicates redirection to the name service switch policies for hosts and services.

Code Example 4–1 shows a sample `netconfig` file. Use of the `netconfig` file has been changed for the `inet` transports, as described in the commented section in the sample file. This change is also described in "Name-to-Address Mapping" on page 116.

CODE EXAMPLE 4–1 Sample `netconfig` File

```
# The ‘‘Network Configuration’’ File.
#
# Each entry is of the form:
#
#<net <semantics>  <flags> <proto    <proto   <device>     <nametoaddr_libs>
# id>                       family>   name>
#
# The "-" in <nametoaddr_libs> for inet family transports indicates redirection
# to the name service switch policies for "hosts" and "services. The "-" may be
# replaced by nametoaddr libraries that comply with the SVR4 specs, in which
# case the name service switch will be used for netdir_getbyname, netdir_
# getbyaddr, gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
# There are no nametoaddr_libs for the inet family in Solaris anymore.
#
udp       tpi_clts     v   inet       udp      /dev/udp        -
#
tcp       tpi_cots_ord v   inet       tcp      /dev/tcp        -
#
icmp      tpi_raw      -   inet       icmp     /dev/icmp       -
#
rawip     tpi_raw      -   inet       -        /dev/rawip      -
#
ticlts    tpi_clts     v   loopback   -        /dev/ticlts     straddr.so
#
ticots    tpi_cots     v   loopback   -        /dev/ticots     straddr.so
#
```

(continued)

(Continuation)

```
ticotsord tpi_cots_ord v   loopback   -        /dev/ticotsord  straddr.so
#
```

Network selection library routines return pointers to `netconfig` entries. The `netconfig` structure is shown in Code Example 4–2.

CODE EXAMPLE 4–2 The `netconfig` Structure

```
struct netconfig {
   char  *nc_netid;               /* network identifier */
   unsigned  int   nc_semantics;  /* semantics of protocol */
   unsigned  int   nc_flag;       /* flags for the protocol */
   char *nc_protofmly;            /* family name */
   char *nc_proto;                /* proto specific */
   char *nc_device;               /* device name for network id */
   unsigned  int   nc_nlookups;   /* # entries in nc_lookups */
   char **nc_lookups;             /* list of lookup libraries */
   unsigned  int   nc_unused[8];
};
```

Valid network IDs are defined by the system administrator, who must ensure that network IDs are locally unique. If they are not, some network selection routines can fail. For example, it is not possible to know which network `getnetconfigent("udp")` will use if there are two `netconfig` entries with the network ID `udp`.

The system administrator also sets the order of the entries in the `netconfig` database. The routines that find entries in `/etc/netconfig` return them in order, from the beginning of the file. The order of transports in the `netconfig` file is the default transport search sequence of the routines. Loopback entries should be at the end of the file.

The `netconfig` file and the `netconfig` structure are described in greater detail in the **`netconfig`**(4) man page.

NETPATH Environment Variable

An application usually uses the default transport search path set by the system administrator to locate an available transport. However, when a user wants to influence the choices made by an application, the application can modify the interface by using the environment variable `NETPATH` and the routines described in

the section, "NETPATH Access to netconfig Data" on page 112. These routines access only the transports specified in the NETPATH variable.

NETPATH is similar to the PATH variable. It is a colon-separated list of transport IDs. Each transport ID in the NETPATH variable corresponds to the network ID field of a record in the netconfig file. NETPATH is described in the **environ**(4) man page.

The default transport set is different for the routines that access netconfig through the NETPATH environment variable (described in the next section) and the routines that access netconfig directly. The default transport set for routines that access netconfig via NETPATH consists of the visible transports in the netconfig file. For routines that access netconfig directly, the default transport set is the entire netconfig file. A transport is visible if the system administrator has included a v flag in the flags field of that transport's netconfig entry.

NETPATH Access to netconfig Data

Three routines access the network configuration database indirectly through the NETPATH environment variable. The variable specifies the transport or transports an application is to use and the order to try them. NETPATH components are read from left to right. The functions have the following interfaces:

```
#include <netconfig.h>

void *setnetpath(void);
struct netconfig *getnetpath(void *);
int endnetpath(void *);
```

A call to setnetpath( ) initializes the search of NETPATH. It returns a pointer to a database that contains the entries specified in a NETPATH variable. The pointer, called a handle, is used to traverse this database with getnetpath( ). The setnetpath( ) function must be called before the first call to getnetpath( ).

When first called, getnetpath( ) returns a pointer to the netconfig file entry that corresponds to the first component of the NETPATH variable. On each subsequent call, getnetpath( ) returns a pointer to the netconfig entry that corresponds to the next component of the NETPATH variable; getnetpath( ) returns NULL if there are no more components in NETPATH. A call to getnetpath( ) without an initial call to setnetpath( ) causes an error; getnetpath( ) requires the pointer returned by setnetpath( ) as an argument.

getnetpath( ) silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the netconfig database.

If the NETPATH variable is unset, getnetpath( ) behaves as if NETPATH were set to the sequence of default or visible transports in the netconfig database, in the order in which they are listed.

endnetpath( ) is called to release the database pointer to elements in the NETPATH variable when processing is complete. endnetpath( ) fails if setnetpath( ) was not called previously. Code Example 4–3 shows the setnetpath( ), getnetpath( ), and endnetpath( ) routines.

CODE EXAMPLE 4–3 setnetpath( ), getnetpath( ), and endnetpath( ) Functions

```
#include <netconfig.h>

void *handlep;
struct netconfig *nconf;

if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}

while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * nconf now describes a transport provider.
    */
}
endnetpath(handlep);
```

The netconfig structures obtained through getnetpath( ) become invalid after the execution of endnetpath( ). To preserve the data in the structure, use getnetconfigent(nconf->nc_netid) to copy them into a new data structure.

Accessing netconfig

Three functions access /etc/netconfig and locate netconfig entries. The routines setnetconfig( ), getnetconfig( ), and endnetconfig( ) have the following interfaces:

```
#include <netconfig.h>

void *setnetconfig(void);
struct netconfig *getnetconfig(void *);
int endnetconfig(void *);
```

A call to `setnetconfig( )` initializes the record pointer to the first index in the database; `setnetconfig( )` must be used before the first use of `getnetconfig( )`. `setnetconfig( )` returns a unique handle (a pointer into the database) to be used by the `getnetconfig( )` routine. Each call to `getnetconfig( )` returns the pointer to the current record in the `netconfig` database and increments its pointer to the next record. It can be used to search the entire `netconfig` database. `getnetconfig( )` returns a `NULL` at the end of file.

You must use `endnetconfig( )` to release the database pointer when processing is complete. `endnetconfig( )` must not be called before `setnetconfig( )`.

CODE EXAMPLE 4–4 `setnetconfig( )`, `getnetconfig( )`, and `endnetconfig( )` Functions

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig()) == (void *)NULL){
   nc_perror(argv[0]);
   exit(1);
}
/*
 * transport provider information is described in nconf.
 * process_transport is a user-supplied routine that
 * tries to connect to a server over transport nconf.
 */
while ((nconf = getnetconfig(handlep)) != (struct netconfig *)NULL){
    if (process_transport(nconf) == SUCCESS)
      break;
}
endnetconfig(handlep);
```

The last two functions have the following interface:

```
#include <netconfig.h>
struct netconfig *getnetconfigent(char *);
int freenetconfigent(struct netconfig *);
```

`getnetconfigent( )` returns a pointer to the `struct netconfig` structure corresponding to `netid`. It returns `NULL` if `netid` is invalid. `setnetconfig( )` need not be called before `getnetconfigent( )`.

`freenetconfigent( )` frees the structure returned by `getnetconfigent( )`. Code Example 4–5 shows the `getnetconfigent( )` and `freenetconfigent( )` routines.

CODE EXAMPLE 4–5 `getnetconfigent()` and `freenetconfigent()` Functions

```
/* assume udp is a netid on this host */
struct netconfig *nconf;

if ((nconf = getnetconfigent(''udp'')) == (struct netconfig *)NULL){
   nc_perror(''no information about udp'');
   exit(1);
}
process_transport(nconf);
freenetconfigent(nconf);
```

Loop Through all Visible `netconfig` Entries

The `setnetconfig()` call is used to step through all the transports marked *visible* (by a `v` flag in the `flags` field) in the `netconfig` database. The transport selection routine returns a `netconfig` pointer.

Looping Through User-Defined `netconfig` Entries

Users can control the loop by setting the `NETPATH` environment variable to a colon-separated list of transport names. If `NETPATH` is set as follows:

```
NETPATH=tcp:udp
```

The loop first returns the `tcp` entry, then the `udp` entry. If `NETPATH` is not defined, the loop returns all visible entries in the `netconfig` file in the order in which they are stored. The `NETPATH` environment variable lets users define the order in which client-side applications try to connect to a service. It also lets the server administrator limit transports on which a service can listen.

Use `getnetpath()` and `setnetpath()` to obtain or modify the network path variable. Code Example 4–6 shows the form and use, which are similar to the `getnetconfig()` and `setnetconfig()` routines.

CODE EXAMPLE 4–6 Looping Through Visible Transports

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig() == (void *) NULL) {
    nc_perror(''setnetconfig'');
   exit(1);
}
```

(continued)

(Continuation)

```
while (nconf = getnetconfig(handlep))
    if (nconf->nc_flag & NC_VISIBLE)
      doit(nconf);
(void) endnetconfig(handlep);
```

Name-to-Address Mapping

Name-to-address mapping lets an application obtain the address of a service on a specified host, independent of the transport used. Name-to-address mapping consists of the following functions:

`netdir_getbyname( )`	Maps the host and service name to a set of addresses
`netdir_getbyaddr( )`	Maps addresses into host and service names
`netdir_free( )`	Frees structures allocated by the name-to-address translation routines
`taddr2uaddr( )`	Translates an address and returns a transport-independent character representation of the address
`uaddr2taddr( )`	The universal address is translated into a `netbuf` structure
`netdir_options( )`	Interfaces to transport-specific capabilities (such as the broadcast address and reserved port facilities of TCP and UDP)

The first argument of each routine points to a `netconfig` structure that describes a transport. The routine uses the array of directory-lookup library paths in the `netconfig` structure to call each path until the translation succeeds.

The libraries are described in Table 4–2. The routines described in the section, “Using the Name-to-Address Mapping Routines” on page 118, are defined in the **`netdir`**(3N) man page.

Note - The following libraries no longer exist in the Solaris 2 environment: `tcpip.so`, `switch.so`, and `nis.so`. For more information on this change, see the **`nsswitch.conf`**(4) man page and the NOTES section of the **`gethostbyname`**(3N) man page.

TABLE 4–2 Name-to-Address Libraries

Library	Transport Family	Description
-	inet	For networks of the protocol family inet, its name-to-address mapping is provided by the name service switch based on the entries for *hosts* and *services* in the file nsswitch.conf. For networks of other families, the "-" indicates a non-functional name-to-address mapping.
straddr.so	loopback	Contains the name-to-address mapping routines of any protocol that accepts strings as addresses, such as the loopback transports.

straddr.so Library

Name–to–address translation files for the library are created and maintained by the system administrator. The straddr.so files are /etc/net/*transport-name*/hosts and /etc/net/*transport-name*/services. *transport-name* is the local name of the transport that accepts string addresses (specified in the *network ID* field of the /etc/netconfig file). For example, the host file for ticlts would be /etc/net/ticlts/hosts, and the service file for ticlts would be /etc/net/ticlts/services.

Even though most string addresses do not distinguish between *host* and *service*, separating the string into a host part and a service part is consistent with other transports. The /etc/net/*transport-name*/hosts file contains a text string that is assumed to be the host address, followed by the host name. For example:

```
joyluckaddr joyluck
carpediemaddr carpediem
thehopaddr thehop
pongoaddr pongo
```

For loopback transports, it makes no sense to list other hosts because the service cannot go outside the containing host.

The /etc/net/transport-name/services file contains service names followed by strings identifying the service address. For example:

```
rpcbind rpc
listen serve
```

The routines create the full-string address by concatenating the host address, a period (.), and the service address. For example, the address of the `listen` service on `pongo` is `pongoaddr.serve`.

When an application requests the address of a service on a particular host on a transport that uses this library, the host name must be in `/etc/net/`*transport*`/hosts`, and the service name must be in `/etc/net/`*transport*`/services`. If either is missing, the name-to-address translation fails.

Using the Name-to-Address Mapping Routines

This section provides an overview of what routines are available to use. The routines return or convert the network names to their respective network addresses. Note that `netdir_getbyname( )`, `netdir_getbyaddr( )`, and `taddr2uaddr( )` return pointers to data that must be freed by calls to `netdir_free( )`.

```
int netdir_getbyname(struct netconfig *nconf,
  struct nd_hostserv *service,
  struct nd_addrlist **addrs);
```

`netdir_getbyname( )` maps the host and service name specified in *service* to a set of addresses consistent with the transport identified in *nconf*. The `nd_hostserv` and `nd_addrlist` structures are defined in the **`netdir`**(3N) man page. A pointer to the addresses is returned in *addrs*.

To find all addresses of a host and service (on all available transports), call `netdir_getbyname( )` with each `netconfig` structure returned by either `getnetpath( )` or `getnetconfig( )`.

```
int netdir_getbyaddr(struct netconfig *nconf,
    struct nd_hostservlist **service,
    struct netbuf *netaddr);
```

`netdir_getbyaddr( )` maps addresses into host and service names. The function is called with an address in *netaddr* and returns a list of host-name and service-name pairs in *service*. The `nd_hostservlist` structure is defined in **`netdir`**(3N).

```
void netdir_free(void *ptr, int struct_type);
```

The `netdir_free( )` routine frees structures allocated by the name-to-address translation routines. The parameters can take the values shown in Table 4–3.

TABLE 4–3 `netdir_free( )` Routines

struct_type	ptr
`ND_HOSTSERV`	Pointer to an `nd_hostserv` structure
`ND_HOSTSERVLIST`	Pointer to an `nd_hostservlist` structure
`ND_ADDR`	Pointer to a `netbuf` structure
`ND_ADDRLIST`	Pointer to an `nd_addrlist` structure

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

`taddr2uaddr( )` translates the address pointed to by *addr* and returns a transport-independent character representation of the address (“universal address”). *nconf* specifies the transport for which the address is valid. The universal address can be freed by `free( )`.

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

The “universal address” pointed to by *uaddr* is translated into a `netbuf` structure; *nconf* specifies the transport for which the address is valid.

```
int netdir_options(struct netconfig *nconf, int option, int fd,
 char *point_to_args);
```

`netdir_options( )` interfaces to transport-specific capabilities (such as the broadcast address and reserved port facilities of TCP and UDP). *nconf* specifies a transport. *option* specifies the transport-specific action to take. *fd* might or might not be used depending upon the value of *option*. The fourth argument points to operation-specific data.

Table 4–4 shows the values used for `option`:

TABLE 4–4 Values for `netdir_options`

Option	Description
`ND_SET_BROADCAST`	Sets the transport for broadcast (if the transport supports broadcast)
`ND_SET_RESERVEDPORT`	Lets the application bind to a reserved port (if allowed by the transport)
`ND_CHECK_RESERVEDPORT`	Verifies that an address corresponds to a reserved port (if the transport supports reserved ports)
`ND_MERGEADDR`	Transforms a locally meaningful address into an address to which client hosts can connect

`netdir_perror()` displays the message stating why one of the name-to-address mapping routines failed on `stderr`.

```
void netdir_perror(char *s);
```

`netdir_sperror()` returns a string containing the error message stating why one of the name-to-address mapping routines failed.

```
char *netdir_sperror(void);
```

Code Example 4–7 shows network selection and name-to-address mapping.

CODE EXAMPLE 4–7 Network Selection and Name-to-Address Mapping

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                    /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
```

(continued)

(Continuation)

```
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * Print out the information associated with the
    * transport provider described in the "netconfig"
    * structure.
    */
   printf("Transport provider name: %s\n", nconf->nc_netid);
   printf("Transport protocol family: %s\n", nconf->nc_protofmly);
   printf("The transport device file: %s\n", nconf->nc_device);
   printf("Transport provider semantics: ");
    switch (nconf->nc_semantics) {
   case NC_TPI_COTS:
      printf("virtual circuit\n");
      break;
   case NC_TPI_COTS_ORD:
      printf("virtual circuit with orderly release\n");
      break;

   case NC_TPI_CLTS:
      printf("datagram\n");
      break;
   }
   /*
    * Get the address for service "date" on the host
    * named "gandalf" over the transport provider
    * specified in the netconfig structure.
    */
   if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp) != ND_OK) {
      printf("Cannot determine address for service\n");
      netdir_perror(argv[0]);
      continue;
   }
   printf("<%d> addresses of date service on gandalf:\n",
      nd_addrlistp->n_cnt);
   /*
    * Print out all addresses for service "date" on
    * host "gandalf" on current transport provider.
    */
   netbufp = nd_addrlistp->n_addrs;
   for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
      uaddr = taddr2uaddr(nconf,netbufp);
      printf("%s\n",uaddr);
```

(continued)

(Continuation)

```
      free(uaddr);
   }
    netdir_free( nd_addrlistp, ND_ADDRLIST );

}
endnetconfig(handlep);
```

Glossary

CLTS Connectionless transport service. Data can be exchanged without a prior link between processes. Also known as a datagram protocol because the operation is like sending a letter.

client A process that makes use of a service or services provided by other processes. A client process initiates requests for services.

concurrent server A multithreaded server that creates a new process to handle each request, leaving the main server process to listen for more requests. With a multithreaded OS, such as SunOS 5, it is possible to implement concurrent servers without creating a complete process to handle requests; each request can be dealt with in a single thread.

COTS Connection-oriented transport service. Requires a logical connection to be established between two processes before data can be exchanged. Conceptually analogous to a telephone call.

ICMP Internet Control Message Protocol. A network layer protocol dealing with routing, reliability, flow control and sequencing of data.

internetwork The connection of different physical networks into a large, virtual network. The Internet refers to the TCP/IP-based Internet that connects many commercial sites, government agencies, and universities.

IP Internet protocol. Core protocol of TCP/IP at the network layer. A connectionless service, it handles packet delivery for TCP, UDP, and ICMP protocols.

ISO/OSI The International Organization for Standards (ISO) model for Open Systems Interconnection (OSI) is a seven layer model for describing networked systems.

iterative server	A single-threaded server that can handle only one request at a time. Requests are received and processed within a single process. It is possible for client processes to be blocked for some time while waiting for requests to be finished.
protocol	A set of rules and conventions that describes how information is to be exchanged between two entities.
protocol stack	A set of layered protocols where each layer has a well-defined interface to the layer immediately above and immediately below.
protocol peers	A pair of protocols that reside in the same layer. They communicate with each other.
RFC	Request for Comments. Formal specifications of the Internet protocols.
server	A process that provides some facility that can be used by other processes. A server process waits for requests.
TCP	Transmission Control Protocol. Built on top of IP at the transport layer, TCP provides a reliable connection-oriented byte stream service between two hosts on an internetwork.
UDP	User Datagram Protocol. Built on top of IP at the transport layer, UDP provides an unreliable datagram-based service between two hosts on an internetwork.
well-known port numbers	TCP and UDP port numbers that identify individual processes on a host. Well-known services are provided at well-known port numbers.

Index

L

M

N

O

P

R

S

T

U

X

Z