Part 4

XTI: X/Open Transport Interface

28

XTI: TCP Clients

28.1 Introduction

Figure 1.15 showed that the sockets API was introduced in 1983 with 4.2BSD and initially worked with the TCP/IP protocol suite and the Unix domain protocols. During the mid-1980s, before Posix.l was complete, there was still a rift within the Unix community between "Berkeley Unix" and "AT&T Unix." in the networking world, the claim was that TCP/IP would be replaced "shortly" with the OSI protocols.

In 1986 AT&T introduced a different networking API called TLI (Transport Layer Interface) with Release 3.0 of System V (SVR3). Although there are numerous similarities between TLI and sockets, TLI was modeled after the OSI Transport Service Definition. SVR3 also provided the first commercial release of the streams subsystem, which we say more about in Chapter 33. Unfortunately SVR3 did not include any networking protocols such as TCP/IP: it included only the streams and TLI building blocks. This led to a few companies providing third-party networking protocols for System V, usually TCP/IP and some preliminary implementations of the OSI protocols. System V Release 4 (SVR4) in 1990 finally provided the TCP/IP protocols as part of the basic operating system.

We mentioned X/Open in Section 1.10. In 1988 they released a modification of TLI called XTI: the *X/Open Transport Interface. XTI is* basically a superset of TLI and has gone through several versions. We describe XTI instead of TLI in this text because the Posix.lg standard started with XTI, not TLI. What we describe in the following chapters is XTI as specified for Unix 98 [Open Group 1997], which is nearly identical to the Posix.lg XTI.

XTI uses the term *communications provider* to describe the protocol implementation. The commonly available communications providers are for the Internet protocols, that is, TCP and UDP. The term *communications endpoint* refers to an object that is created

and maintained by a communications provider and then used by an application. These endpoints are referred to by file descriptors. We will often shorten these two terms to just *provider* and *endpoint.*

TLI referred to these as the *transport provider* and the *transport endpoint.*

All the XTI functions begin with `t_`. The header that the application includes to obtain all the XTI definitions is `<xti.h>`. Some Internet-specific definitions are obtained by including `<xti_inet.h>`.

We discuss XTI in the following order:

- TCP clients,
- name and address functions,
- TCP servers,
- UDP clients and servers,
- options,
- streams, and
- additional functions.

Our discussion of XTI is shorter than our discussion of sockets because the network programming *techniques* are the same. What changes are the function names, function arguments, and some of the nitty-gritty details (e.g., accepting TCP connections), but there is no need to duplicate every one of the sockets examples using XTI.

28.2 t_open Function

The first step in establishing a communications endpoint is to open the Unix device that identifies the particular communications provider. This function returns a descriptor (a small integer) that is used by the other XTI functions.

```
#include <xti.h>
#include <fcntl.h>

int t_open(const char *pathname, int oflag, struct t_info *info);
```

Returns: 0 if OK, –1 on error

The actual *pathname* to use depends on the implementation, but typical values for TCP/IP endpoints are `/dev/tcp`, `/dev/udp`, or `/dev/icmp`. Typical values for loopback endpoints are `/dev/ticots`, `/dev/ticotsord`, and `/dev/ticots`.

The *oflag* argument specifies the open flags. Its value is `O_RDWR`. For a nonblocking endpoint the flag `O_NONBLOCK` is logically ORed with `O_RDWR`.

This XTI function is similar to the socket function. Both return a file descriptor that is associated with a user-specified protocol.

The `t_info` structure is a collection of integer values that describe the protocol-dependent features of the provider. This structure is returned through the *info* pointer,

if this argument is not a null pointer. This is our first encounter with one of the XTI structures that begins with t_. There are seven of these structures, which we say more about in Section 28.4.

```
struct t_info {
  t_scalar_t  addr;      /* max #bytes of communications protocol address */
  t_scalar_t  options;   /* max #bytes of protocol-specific options */
  t_scalar_t  tsdu;      /* max #bytes of transport service data unit (TSDU) */
  t_scalar_t  etsdu;     /* max #bytes of expedited TSDU (ETSDU) */
  t_scalar_t  connect;   /* max #bytes of data on conn. establishment */
  t_scalar_t  discon;    /* max #bytes of data on t_XXXdis() & t_XXXreldata */
  t_scalar_t  servtype;  /* service type supported */
  t_scalar_t  flags;     /* other information (new with XTI) */
```

This is our first encounter with the t_scalar_t datatype, which is new with Unix 98. Older implementations use a long integer for all these members, but this presents a problem on 64-bit architectures as we discussed in Section 1.11. t_scalar_t and t_uscalar_t are therefore defined to be int32_t anduint32_t, respectively.

Before describing each of the members of the t_info structure, we show some typical values for TCP and UDP, in Figures 28.1 and 28.2, which we explain shortly.

	AIX 4.2	DUnix 4.0B	HP-UX 10.30	Solaris 2.6	UnixWare 2.1.2
addr	16	16	16	16	16
options	512	4096	1024	504	360
tsdu	0	0	0	0	0
etsdu	-1	-1	-1	-1	-1
connect	-2	-2	-2	-2	-2
discon	-2	-2	-2	-2	-2
servtype	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD

Figure 28.1 t_info values for TCP.

	AIX 4.2	DUnix 4.0B	HP-UX 10.30	Solaris 2.6	UnixWare 2.1.2
addr	16	16	16	16	16
options	512	768	256	468	328
tsdu	8192	9216	65508	65508	65508
etsdu	-2	-2	-2	-2	-2
connect	-2	-2	-2	-2	-2
discon	-2	-2	-2	-2	-2
servtype	T_CLTS	T_CLTS	T_CLTS	T_CLTS	T_CLTS

Figure 28.2 t_info values for UDP.

We are interested in three cases for each of the first six variables in the t_info structure: >= 0, -1 (also called T_INFINITE), and -2 (also called T_INVALID).

addr This specifies the maximum size in bytes of a protocol-specific address. A value of -1 indicates there is no limit to the size. A value of -2 indicates there is no user access to the protocol addresses.

The value of 16 shown for TCP and UDP is the size of a `sockaddr_in` structure. For an IPv6 endpoint this value will probably be the size of a `sockaddr_in6` structure.

`options` This specifies the size in bytes of the protocol-specific options. A value of -1 indicates there is no limit to the size. A value of -2 indicates there is no user access to the options. We talk more about XTI options in Chapter 32.

As we can see in the examples, there is little commonality amongst the various implementations, with the size ranging from 256 to 1024 bytes.

`tsdu` TSDU stands for "transport service data unit." This variable specifies the maximum size in bytes of a record whose boundaries are preserved from one endpoint to the other. A value of zero indicates that the communications provider does not support the concept of a TSDU, although it sup-ports a byte stream of data (i.e., no record boundaries). A value of -1 indicates there is no limit to the size. A value of -2 indicates that the trans-port of normal data is not supported (a rare condition).
For TCP the value is always 0, since TCP provides a byte-stream service without any record boundaries. The predominant value for UDP is 65508, which is wrong. The maximum size of an IP datagram is 65535 bytes (the 16-bit total length field in Figure A.1), so the maximum size of a UDP datagram is 65535 minus 20 (for the IP header) minus 8 (for the UDP header), or 65507.

`etsdu` ETSDU stands for "expedited transport service data unit" and this variable specifies the maximum size in bytes of an ETSDU. This is what we called out-of-band data in Chapter 21. A value of zero indicates that the communications provider does not support the concept of ETSDU, although it supports a byte stream of out-of-band data (i.e., record boundaries are not preserved in the out-of-band data). A value of -1 indicates there is no limit to the size. A value of -2 indicates that the transport of expedited data is not supported.

As we expect, UDP does not support any form of out-of-band data. TCP supports the concept, but there is no limit to the amount of out-of-band data that the application can send. (Recall our discussion of TCP's urgent mode in Section 21.2.)

`connect` Some connection-oriented protocols support the transfer of user data along with a connection request. This variable specifies the maximum amount of this data. A value of -1 indicates there is no limit to the size. A value of -2 indicates that the communications provider does not support this feature.

TCP does not support this feature, so its value is always -2, and since UDP is not a connection-oriented protocol, its value is also -2. The connection-oriented OSI transport layer supports this feature.

> Note that TCP allows sending data with a SYN, as described on pp. 14—16 of TCPv3. Sockets and XTI, however, provide no *way* to cause TCP to send data with a SYN.

Nevertheless, what this member of the `t_info` structure is referring to is something different (e.g., the capability provided by the OSI transport layer).

`discon` Some connection-oriented protocols support the transfer of user data along with a disconnection request. We will see the possibility of this when we discuss the `t_snddis` and `t_rcvdis` functions later in this chapter. This variable specifies the maximum amount of this data. A value of -1 indicates there is no limit to the size. A value of -2 indicates that the communications provider does not support this feature. This variable also specifies the amount of user data that can be sent with an orderly release, using the `t_sndreldata` and `t_rcvreldata` functions, which we describe in Section 34.10.

TCP does not support this feature, but it is supported by the OSI transport layer.

`servtype` This specifies the type of service provided by the communications provider. There are three possibilities which we show in Figure 28.3.

servtype	Description
T_COTS	connection-oriented service, without orderly release
T_COTS_ORD	connection-oriented service, with orderly release
T_CLTS	connectionless service

Figure 28.3 Types of service provided by communications providers.

TCP is connection oriented with orderly release and UDP is connectionless.

`flags` This member, which is new with XTI, specifies additional flags for the communications provider. The two constants shown in Figure 28.4 are defined by including the `<xti.h>` header that can be returned in this member.

flag	Description
T_SENDZERO	provider supports 0-length writes
T_ORDRELDATA	provider supports orderly release data (`t_sndreldata` and `t_rcvreldata`)

Figure 28.4 Values for `flags` member of `t_info` structure.

TCP does not support 0-length writes, but UDP does (resulting in a 28-byte IP datagram, with just an IP header and a UDP header, but no data). TCP does not support the T_ORDRELDATA flag either.

28.3 `t_error` and `t_strerror` Functions

Recall that most of the socket functions (e.g., `socket, bind, connect,` and so on) return -1 when they encounter an error and set the variable `errno` to provide

additional information about the error. The XTI functions normally return -1 on an error and set the variable t_errno to provide additional information about the error. (Recall our discussion of errno in Section 23.1 and how it is a per-thread variable. In a threads environment t_errno must also be a per-thread variable.) t_errno is similar to errno in that it is set only when an error occurs and it is not cleared on successful calls.

All the XTI error codes are defined as a result of including <xti.h> and begin with T, as in TBADADDR (incorrect address format), TBADF (illegal transport descriptor), and so on.

One special error value is TSYSERR and when it is returned in t_errno, it tells the application to look at the value in errno for the system error indication.

The two functions t_error and t_strerror are provided to help format error messages resulting from XTI functions.

```
*include <xti.h>

int t_error(const char *msg);

                                                              Returns: 0

const char *t_strerror(int errnum);

                                                  returns: pointer to message
```

t_error produces a message on the standard error output. This message consists of the string pointed to by *msg* (assuming this pointer is nonnull) followed by a colon and a space, followed by a message string corresponding to the current value of t_errno. If t_errno equals TSYSERR, then a message string is also output corresponding to the current value of errno. Finally a newline is output.

t_strerror returns a string describing the value of *errnum*, which is assumed to be one of the possible t_errno values. Unlike t_error, t_strerror does nothing special if this value is TSYSERR.

The program in Figure 28.5 shows the use of these two XTI error functions, along with our err_xti function. (We describe the latter in Section D.4.)

xtiintro/strerror.c

```
 1 #include     "unpxti.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     printf("%s\n", t_strerror(TPROTO));
 6     errno = ETIMEDOUT;
 7     printf("%s\n", t_strerror(TSYSERR));
 8     t_errno = TSYSERR;
 9     errno = ETIMEDOUT;
10     t_error("t_error says");
```

```
11      t_errno = TSYSERR;
12      errno = ETIMEDOUT;
13      err_xti("err_xti says");
14      exit(0);
15  }
```
xtiintro/strerror.c

Figure 28.5 Example of `t_error` and `t_strerror` functions.

The output from this program is

```
aix % strerror
XTI protocol error
system error
t_error says: system error, Connection timed out
err_xti says: system error: Connection timed out
```

28.4 netbuf Structures and XTI Structures

XTI defines seven structures that are used to pass information between the application and the XTI functions. One of these, the `t_info` structure that we described in Section 28.2, is just a collection of integer values that describe protocol-dependent features of the provider. The remaining six structures each contain between one and three `netbuf` structures. The `netbuf` structure defines a buffer that is used to pass data from the application to the XTI function or vice versa.

```
struct netbuf
   unsigned int maxlen;    /* maximum size of buf */
   unsigned  int  len;     /* actual amount of data in buf */
   void          *buf      /* data  (char* before Posix.lg) */
};
```

Figure 28.6 shows the six XTI structures that contain one or more `netbuf` structures, and the various other members of the XTI structure.

Datatype	XTI structure					
	t_bind	t_call	t_discon	t_optmgmt	t_uderr	t_unitdata
struct netbuf struct netbuf struct netbuf	addr	addr opt udata	 udata	opt	addr opt	addr opt udata
t_scalar_t t_scalar_t unsigned int int int	glen	 sequence	 reason sequence	flags	error	

Figure 28.6 Six XTI structures and their members.

These six XTI structures that contain the `netbuf` structures are always passed by reference between the application and the XTI function. That is, we pass the address of

the XTI structure as an argument to an XTI function. Therefore the XTI function can always read and update any of the three members of the `netbuf` structure (although none of the functions change the `maxlen` member).

The use of the three members of the `ne tbu f` structure depends on which direction the structure is being passed: from the application to the XTI function, or vice versa, as shown in Figure 28.7. We also note whether the XTI function reads the value of the member or writes the value of the member.

Member	Data from application to XTI	Data from XTI to application
maxlen	Ignored.	Read-only. Size of buffer pointed to by buf. XTI function will not store more than this amount of data in buf. If 0, then nothing is returned and len and buf are ignored.
len	Read-only. Application sets this to amount of data pointed to by buf.	Write-only XTI function sets this member to the actual amount of data stored in buf, and this value will always be less than or equal to the value of maxlen.
buf	Pointer to data stored by application and then	Pointer to data stored by XTI function and then nrnrPCCOri by annliratinn

Figure 28.7 Processing of three members of netbuf structure.

If XTI has more data to return than `maxlen` allows, the XTI call fails with `t_errno` set to `TBUFOVFLW`.

Since the address of the `netbuf` structure is always passed to an XTI function, and since the structure contains both the size of the buffer `(maxlen)` and the amount of data actually stored in the buffer `(len),` there is no need in XTI for all the value-result arguments used with sockets.

28.5 t_bind Function

This function assigns the local address to an endpoint and activates the endpoint. In the case of TCP or UDP the local address is an IP address and a port.

```
*include <xti.h>

int t_bind(int fd, const struct t_bind *request, struct t_bind *return) ;
```

Returns: 0 if OK, -1 on error

The second and third arguments point to `t_bind` structures:

```
struct t_bind {
  struct netbuf addr;     /* protocol-specific address */
  unsigned int glen;      /* max# of outstanding connections (if server) */
}
```

The endpoint is specified by *fd*. There are three cases to consider for the *request* argument.

request == NULL
: The caller does not care what local address gets assigned to the endpoint. The provider selects an address. The value of the `glen` element is assumed to be zero (see below).

request != NULL, but *request->addr.len == 0*
: The caller does not care what local address gets assigned to the endpoint, and again the provider selects an address. Unlike the previous case, however, the caller can now specify a nonzero value for the `glen` member of the *request* structure.

request != NULL, and *request->addr.len > 0*
: The caller specifies a local address for the communications provider to assign to the communications endpoint.

Whether the application specifies the address or whether the provider selects an address, the provider returns the address that it assigns to the endpoint in the *return* structure. If the *return* argument is a null pointer, the provider does not return the actual address.

The value of `glen` has meaning only for a connection-oriented server: it specifies the maximum number of connections to queue for this endpoint. It is possible for this value to be changed by the provider, in which case the `glen` element of the *return* structure indicates the actual value supported by the provider. We say more about this value and measure the actual number of connections queued for various values of `glen` with Figure 30.14.

Notice that the `addr` member of the `t_bind` structure is an actual `netbuf` structure, and not a pointer to one of these structures. We will see that this is common to these XTI structures: most contain one or more `netbuf` structures within the `t_XXX` structure.

If XTI cannot bind the requested address, the error `TADDRBUSY` is returned. If TLI encountered this problem, it could bind another local address to the endpoint, requiring the caller to then compare the assigned address to the requested address.

The XTI method for the caller telling the provider to select an appropriate address is more generic than the method used by bind. For example, with TCP and UDP over IPv4 we must specify an Internet address of `INADDR_ANY` and a port of zero for the provider to select the local address. This is IPv4-specific and not generic to bind.

The glen-value corresponds to the backlog argument specified to `listen`. For a connection-oriented server, the `t_bind` function does the same work as the bind and `listen` functions.

A connection-oriented XTI client must call `t_bind` before calling `t_connect` (which we describe next). This differs from connect, which calls bind internally, if the socket has not been bound.

28.6 t_connect Function

A connection-oriented client initiates a connection with a server by calling t_connect. The client specifies the server's protocol address (e.g., IP address and port for a TCP server).

```
#include <xti.h>

int t_connect(int fd, const struct t_call *sendcall, struct t_call *recvcall);
```

Returns: 0 if OK, –1 on error

The second and third arguments point to a t_call structure:

```
struct t_call {
  struct netbuf  addr;         /* protocol-specific address */
  struct netbuf  opt;          /* protocol-specific options */
  struct netbuf  udata;        /* user data to accompany connection request */
  int            sequence;     /* for t_listen() & t_accept() functions */
};
```

The t_call structure pointed to by the *sendcall* argument specifies the information needed by the transport provider to establish the connection: the addr structure specifies the server's address, opt specifies any protocol-specific options desired by the caller, and udata contains any user data to be transferred to the server during connection establishment. (Recall from Figure 28.1 that TCP does not support any user data being sent with the connection request.) The sequence member has no significance for this function but is used when this structure is used with the t_accept function.

On return from this function, the t_call structure pointed to by the *recvcall* argument contains information associated with the connection that is returned by the communications provider to the caller: the addr structure contains the address of the peer process, opt contains any protocol-dependent optional data associated with the connection, and udata contains any user data returned by the peers transport provider during connection establishment. Again, the sequence member has no meaning.

The contents of the opt structure are protocol dependent. The caller can set the len field of this structure to 0, telling the communications provider to use default values for any connection options. We talk more about XTI options in Chapter 32.

The caller can specify a null pointer for the *recvcall* argument, if the return information about the connection is not desired.

By default, this function does not return until the connection is completed, or an error occurs. We discuss how to perform a nonblocking connect in Section 34.3.

We saw in Section 4.3 that common errors when establishing a TCP connection are receiving an RST, receiving an ICMP destination unreachable, and timing out. Unfortunately, when one of these common errors occurs, t_connect returns -1, but t_errno is set to TLOOK, requiring more code to determine the exact reason. We discuss this problem in Sections 28.9 and 28.10 and show an example in Figure 28.13.

The t_connect function is similar to the connect function.

28.7 t_rcv and t_snd Functions

By default, XTI applications cannot call the normal `read` and `write` functions (unless the `t i rdwr` module is pushed onto the stream, as we describe in Section 28.12). Instead XTI applications must call `t_rcv` and `t_snd`.

```
#include <xti.h>

int t_rcv(int fd, void *buff, unsigned int nbytes, int *flagsp) ;

int t_snd (int ft, const void *buff, unsigned int nbytes, int flags);
```

Both return: number of bytes read or written if OK, -1 on error

The first three arguments are similar to the first three arguments to `read` and `write`: descriptor, buffer pointer, and number of bytes to read or write.

> The input and output functions in the sockets API all use size_t for the buffer size, and ssize_t for the return value. The XTI functions use unsigned int and int.

The *flags* argument to `t_snd` is either zero, or some combination of the constants shown in Figure 28.8.

flag	Description
T_EXPEDITED	send or receive expedited (out-of-band) data
T_MORE	there is more data to send or receive

Figure 28.8 *flags* for t_rcv and t_snd.

T_EXPEDITED is used with `t_snd` to send out-of-band data (Section 34.12). This flag is set on return from `t_rcv` when out-of-band data is received.

`T_MORE` is provided so that multiple `t_rcv` or `t_snd` function calls can read or write what the protocol considers a logical record. This feature applies only to those protocols that support the concept of records. We show an example of this flag with the `t_rcvudata` function and the record-oriented UDP protocol in Figure 31.7. This flag is also used with TCP when reading out-of-band data, as we describe in Section 34.12, but is never used with normal TCP data.

> XTI defines a T_PUSH flag that tells the provider to send all the data that is has accumulated but not yet sent. This flag is used with XTI over SNA (IBM's Systems Network Architecture) but should not be used with TCP, and more specifically does *not* cause TCP's PUSH flag to be set.

Note that the *flags* argument to `t_snd` is an integer value, while the corresponding argument for `t_rcv` is a pointer to an integer. But the value pointed to by *flagsp* for `t_rcv` is not a true value–result argument, because its value is not examined by the function; it is set only on return.

Both of these functions return the actual number of bytes read or written. The return value from `t_snd` can be less than *nbytes* if the endpoint is nonblocking or if a signal is caught by the process.

These two functions correspond to the `send` and recv functions. The XTI T_EXPEDITED flag corresponds to *MSG_00B*, although with XTI we cannot specify this flag to t_rcv.

Recall with a TCP socket that the receipt of a FIN causes `read` to return 0 and the receipt of an RST causes `read` to return -1, with `errno` set to `ECONNRESET`. `t_rcv` behaves differently when either of these conditions occur on an XTI endpoint:

- When a TCP FIN is received for an XTI endpoint, `t_rcv` returns -1 with `t_errno` set to `TLOOK`. The XTI function `t_look` must then be called, and it returns `T_ORDREL`. This is called an *orderly release indication.*
- When a TCP RST is received for an XTI endpoint, `t_rcv` returns -1 with `t_errno` set to `TLOOK`. The XTI function `t_look` must then be called, and it returns `T_DISCONNECT`. This is called *a disconnect* or an *abortive release.*

We first discuss the `t_look` function, followed by the orderly release and abortive release functions.

28.8 `t_look` Function

Various *events* can occur for an XTI endpoint and these events can occur *asynchronously.* By that we mean that the application can be performing some task when an unrelated event occurs on the endpoint. Some events indicate an error condition (`T_UDERR`, an error in a previously sent datagram) while other events are not an error (`T_EXDATA`, the arrival of expedited data).

For example, assume the application calls `t_snd` to send data to the peer, but right before this something happens at the peer and the peer process sends an RST and terminates. This unexpected event (having received an RST when the application calls `t_snd`) is passed to the application by having `t_snd` return -1 with `t_errno` set to `TLOOK`. The application then calls `t_look` to determine what happened (e.g., which event occurred) on the endpoint. The event in this case will be `T_DISCONNECT`, the receipt of a disconnect (an RST).

```
#include <xti.h>

int t_look(int fd);
```

Returns: event (Figure 28.9) if OK, -1 on error

The integer value returned by this function corresponds to one of the nine events shown in Figure 28.9.

When an event occurs on an XTI endpoint, it is considered *outstanding* until it is *consumed.* Figure 28.10 shows which XTI functions consume the XTI events and also shows that two events are consumed by calling `t_look`.

Event	Description
T_CONNECT	connection confirmation received
T_DATA	normal data received
T_DISCONNECT	disconnect received
T_EXDATA	expedited data received
T_GODATA	flow control restrictions on normal data lifted
T_GOEXDATA	flow control restrictions on expedited data lifted
T_LISTEN	connect indication received
T_ORDREL	orderly release indication received
T_UDERR	error in previously sent datagram

Figure 28.9 Events for an XTI endpoint.

Event	Cleared by t._look?	Consuming function
T_CONNECT		t_connect,t_rcvconnect
T_DATA		t_rcv,t_rcvv,t_rcvudata,t_rcvvudata
T_DISCONNECT		t_rcvdis
T_EXDATA		t_rcv,t_rcvv
T_GODATA		t_snd,t_sndv,t_sndudata,t_sndvudata
T_GOEXDATA	yes	
T_LISTEN	yes	t_snd,t_sndv
T_ORDREL		t_listen
T_UDERR		t_rcvrel
		t_rcvuderr

Figure 28.10 XTI events and which functions consume the event.

For the `t_connect` function on a blocking endpoint (the default), the T_CONNECT event is handled by the function itself and not seen by the application. In the example we were considering (the receipt of a FIN when we call `t_snd`) this figure shows that we must call `t_rcvrel` to clear the event.

> At the beginning of this section we described how the receipt of an RST generates a `T_DISCONNECT` event for the endpoint. Until Unix 98 the receipt of a FIN would generate a `T_ORDREL` event for the endpoint. Unix 98 makes this optional.

28.9 t_sndrel and t_rcvrel Functions

XTI supports two ways of releasing a connection: an *orderly release* and an *abortive release*. The differences are that an abortive release does not guarantee the delivery of any outstanding data, while the orderly release guarantees this. All communications providers must support an abortive release, while the support of an orderly release is optional. Recall, however, from Figure 28.1 that TCP provides an orderly release.

We can send and receive an orderly release with the following functions.

```
#include <xti.h>

int t_sndrei(int fd);

int t_rcvrel(int fd);
```

Both return: 0 if OK, -1 on error

To understand the semantics of an orderly release, we must remember that a connection-oriented protocol is usually a full-duplex connection between the two processes. The data transfer in one direction is independent of the data being transferred in the other direction. Figure 28.11 shows one use of these functions with TCP, to take advantage of TCP's half-close.

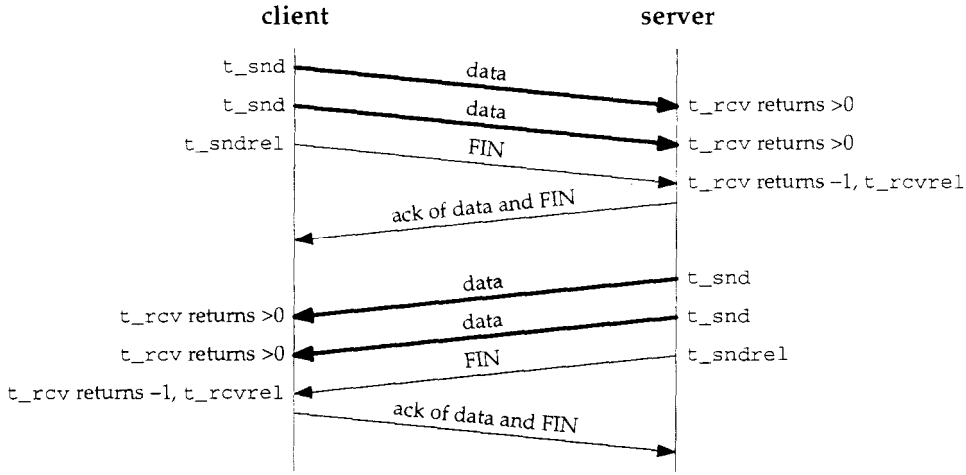

Figure 28.11 TCP's half-close using XTI.

A process issues an orderly release by calling `t_sndrel`. This tells the provider that the application has no more data to send on this endpoint. For a TCP endpoint, TCP sends a FIN to the peer (after any data that is already queued to be sent to the peer). The process that calls `t_sndrel` can continue to receive data, it can still read from the descriptor, but it can no longer write to the descriptor.

> This function performs the same action as `shutdown` with a second argument of `SHUT_WR (1)` on a TCP socket.

A process acknowledges the receipt of a connection release by calling the `t_rcvrel` function. This process can still write to the descriptor but it can no longer read from the descriptor.

> There is nothing in the sockets API comparable to `t_rcvrel`. The receipt of a FIN is delivered to the process as an end-of-file (e.g., `read` returns 0).
>
> This feature of XTI forces the application to deal with the full-duplex orderly release, even if the application is not interested in using this feature, as we will see in Figure 28.13.

28.10 t_snddis and t_rcvdis Functions

The following two functions handle an abortive release (a disconnect).

```
#include <xti.h>

int t_snddis ( int fd, const struct t_call *call );

int t_rcvdis(int fd, struct t_discon *discon) ;
```

Both return: 0 if OK, -1 on error

The `t_snddis` function is used for two different purposes:

- to perform an abortive release of an existing connection, which in terms of TCP causes an RST to be sent, and
- to reject a connection request.

For an abortive release of an existing connection, the *call* argument can be a null pointer, in which case no information is sent to the peer process. Otherwise, the interpretation of the fields in the `t_call` structure is shown in Figure 28.12.

Member	Disconnection of existing connection	Rejection of new connection
addr	ignored	ignored
opt	ignored	ignored
udata	optional	optional
sequence	ignored	required

Figure 28.12 `t_call` structure used with `t_snddis`.

The optional `udata` member specifies user data to accompany the disconnection, but we saw in Figure 28.1 (the `discon` member of the `t_info` structure) that this is not supported by TCP.

> An abortive release is generated by a sockets application by setting the `SO_LINGER` socket option, setting `l_onoff` to a nonzero value and `l_linger` to 0, and then closing the socket (Chapter 7).

When a T_DISCONNECT event occurs on an XTI endpoint (e.g., an RST is received by TCP), the application must receive the abortive release by calling `t_rcvdis`. If the *discon* argument is a nonnull pointer, a `t_discon` structure is filled in with the reason for the abortive release.

```
struct t_discon {
  struct netbuf udata;      /* user data */
  int           reason;     /* protocol-specific reason code */
  int           sequence;
};
```

he `udata` member contains the optional user data that accompanied the disconnect, `reason` is a protocol-dependent reason for the disconnect, and `sequence` is applicable only for servers that are receiving connections.

> There is nothing in the sockets API comparable to `t_rcvdis`. The receipt of an RST is delivered to the process as an input error (e.g., `read` returns -1) with `errno` .et to `ECONNRESET`. Writing to a socket that has received an RST generates `SIGPIPE`.

28.11 XTI TCP Daytime Client

We now recode our TCP daytime client from Figure 1.5 using XTI. Figure 28.13 shows the function.

xtiin tro/day timecli0l.c

```
 1 *include     "unpxti.h"

 2 int
 3 main(int argc, char *argv)
 4 {
 5      int    tfd, n,  flags;
 6      char       recvline[MAXLINE + 11];
 7       struct sockaddr_in servaddr;
 8       struct t_ca11 tcall;
 9       struct t_discon tdiscon;
10      if (argc != 2)
11         err_quit("usage: daytimecli0l <IPaddress>");
12      tfd = T_open(XTI_TCP, 0_RDWR, NULL);
13      T_bind)tfd, NULL, NULL);
14      bzero(&servaddr, sizeof(servaddr));
15      servaddr.sin_family = AF_INET;
16      servaddr.sin_port = htons(13); /* daytime server */
17      Inet_pton(AF_INET, argv[1), &servaddr.sin_addr);
18      tcall.addr.maxlen = sizeof(servaddr);
19      tcall.addr.len = sizeof(servaddr);
20      tcall.addr.buf = &servaddr;
21      tcall.opt.len = 0;           /* no options with connect */
22      tcall.udata.len = 0;         /* no user data with connect */
23      if (t_connect(tfd, &tcall, NULL) < 0) {
24          if (t_errno == TLOOK) {
25              if ( (n = T_look(tfd)) == T_DISCONNECT) {
26                  tdiscon.udata.maxlen = 0;
27                  T_rcvdis(tfd, &tdiscon);
28                  errno = tdiscon.reason;
29                  err_sys("t_connect error");.
30              } else
31                   err_quit("unexpected event after t_connect: %d", n);
32          } else
33              err_xti("t_connect error");
34      }
```

```
35      for ( ; ; ) {
36          if ( (n = t_rcv(tfd, recvline, MAXLINE, &flags)) < 0) {
37              if (t_errno == TLOOK)
38                  if ( (n = T_look(tfd)) == T_ORDREL) {
39                      T_rcvrel(tfd);
40                      break;
41                  } else if (n == T_DISCONNECT) {
42                      tdiscon.udata.maxlen = 0;
43                      T_rcvdis(tfd, &tdiscon);
44                      errno = tdiscon.reason;     /* probably ECONNRESET */
45                      err_sys("server terminated prematurely");
46                  } else
47                      err_quit("unexpected event after t_rcv: %d", n);
48              } else
49                  err_xti(" _rcv error");
50          }
51          recvline[n] = 0;          /* null terminate */
52          fputs(recvline, stdout);
53      }
54      exit(0);
55  }
```

Figure 28.13 Daytime client using XTI.

`unpxti.h` header

We define our own `unpxti.h` header that we `include` in all our XTI programs. We show this header in Section D.3.

Create endpoint, bind any local address

12-13 `t_open` creates the XTI endpoint and we let the system choose its local protocol address by calling `t_bind` with a null second argument.

Specify server's address and port

14-22 We fill in an Internet socket address structure with the server's IP address and port, identical to Figure 1.5. We then fill in a `t_call` structure to point to this socket address structure and we also set the `len` members of the `opt` and `udata` structure to 0, indicating no options and no user data.

> There is nothing in XTI that requires the `t_call` structure to point to a `sockaddr_in` structure for IPv4. Nevertheless, almost all Unix implementations implement XTI with the Internet protocols using the `sockaddr_in` structure to pass the protocol address between the application and the provider. In protocol-independent code the use of this structure should be hidden from the application. We show how to do this in the next chapter.

Establish connection

23-34 `t_connect` establishes the connection, which in this case performs TCP's three-way handshake. As we mentioned earlier, if the connection establishment fails with one of the common errors, `t_connect` returns `TLOOK` and we then call `t_look` to find the event, calling `t_rcvdis` if the event is `T_DISCONNECT`. In this case we also store the reason for the disconnect in `errno` and call our `err_sys` function to print the appropriate error message.

Read from server, copy to standard output

35-53 We read the data from the server with `t_rcv`, printing the data on the standard output, until we hit the end of the connection.

Handle orderly release and disconnect

37-47 When `t_rcv` returns an error, if `t_errno is TLOOK` we call `t_look` to obtain the current event for the endpoint. If that event is `T_ORDREL`, we call `t_rcvrel`; otherwise if it is `T_DISCONNECT`, we call `t_rcvdis`.

If we run this program to the same set of hosts as in Section 4.3, we see the following output. First we connect to a host that is running the daytime server.

```
unixware % daytimecliOl 206.62.226.35
Tue Feb 4 15:00:26 1997
```

Next we specify a nonexistent host on the local subnet.

```
unixware 4 daytimecliOl 206.62.226.55
t_connect. error: Connection timed out
```

Next we specify a router that is not running a daytime server, receiving an RST in response to our SYN.

```
unixware 4 daytimecliOl 140.252.1.4
t_connect error: Connection refused
```

Finally we specify an IP address that is not connected to the Internet, receiving an ICMP host unreachable in response to our SYNs.

```
unixware 3 daytimecliOl 192.3.4.5
t_connect error: No route to host
```

XTI and Sockets Interoperability

We note in the first example shown with our XTI daytime client, connecting to the server on the host bsdi (206.62.226.35), the server is written using sockets but our client is written using XTI. Nevertheless, the client communicates fine with the server. Similarly we could write a daytime server using XTI and it would communicate fine with our client from Figure 1.5 that uses sockets.

This interoperability is provided by the Internet protocol suite and has nothing to do with sockets or XTI. A client written using TCP or UDP interoperates with a server using the same transport protocol if the client and server speak the same *application protocol*, regardless of what API is used to write either the client or the server. It is the application protocol (e.g., HTTP, FTP, Telnet, and so on) and the transport layer (TCP or UDP) that determine the interoperability. The API we use to write either the client or server makes no difference.

28.12 xti_rdwr Function.

As shown in the example in the previous section, by default we cannot use `read` and `write` on a descriptor that references an XTI endpoint. To see what happens, if we modify Figure 28.13 to use `read` and `write` instead of `t_rcv` and `t_snd`, copying the `for` loop from Figure 1.5, we encounter the following error after the connection is established:

```
unixware % daytimecliO3 206.62.226.35
read error: Not a data message
```

We get the same results under AIX, but under HP-UX and Solaris the server's response is read and then an error results:

```
hpux % daytimecliO3 198.69.10.4
Wed Apr 2 18:59:401997 read
error: Bad message
```

We will explain the difference in these two scenarios when discussing Figure 33.11.

Fortunately there is often a way around this problem. If we have a streams-based implementation of XTI (which is common), we can push the streams module `tirdwr` onto the stream and then we can use `read` and `write`. Figure 33.3 shows the streams module involved. But since this capability is implementation dependent, instead of putting the actual `ioctl` command in our program, we define a simple function of our own. This way, if other implementations are encountered, only this library function needs to be changed.

```
#include "unpxti.h"

int xti_rdwr(int fd);
```

Returns: 0 if OK, -1 on error

In Figure 28.14 we show the trivial implementation of this function that just pushes the streams module `tirdwr` onto the stream.

libxti/xti_rdwr.c

```
1 #include    "unpxti.h"
2 int
3 xti_rdwr(int fd)
4 {
5       return (ioctl(fd, I_PUSH, "tirdwr"));
6 }
```

libxti/xti_rdwr.c

Figure 28.14 `xti_rdwr`: push the streams module `tirdwr` onto the stream.

There are a few caveats when using this feature.

- This streams module can be used only when the endpoint is in the data transfer phase. In our example in Figure 28.13, we place our call to `xti_rdwr` after `t_connect` returns.

- Once this module has been pushed onto the stream, none of the XTI functions (those beginning with `t_`) can be called.
- This module cannot be used by applications that use out-9f-band data, as the arrival of out-of-band data cannot be handled by `read`.
- If an orderly release is received, it causes `read` to return 0 (i.e., end-of-file).
- Unfortunately, if a disconnect is received, it also causes `read` to return 0. This makes it impossible to distinguish between the (normal) receipt of a FIN and the (exceptional) receipt of an RST. The receipt of an RST also causes any future calls to `write` to fail.

Recall that with sockets the receipt of an RST causes read to return -1 of ECONNRESET.

28.13 Summary

XTI clients are similar to sockets clients, calling `t_open` instead of `socket`, and `t_connect` instead of `connect`. The same Internet socket address structure is used by both to specify the protocol address of the server, although with XTI this structure is described by another `netbuf` structure. By default `t_rcv` and `t_snd` are used by XTI instead of `read` and `write`, although the latter two can be used, depending on the environment, and we saw that in a streams environment a different streams module must be pushed onto the stream to use these two functions.

XTI defines nine events that can occur on an endpoint. When one of these occurs the XTI function returns an error of `TLOOK` and we must call the `t_look` function to determine what has happened and then handle it as desired. Handling these events often increases the amount of code that we must write, compared to the same scenario with sockets.

Exercises

28.1 In the second scenario for `t_bind` we mentioned that the *request* argument is nonnull but the `addr . ten` member of that structure is 0, allowing the caller to specify 4 nonzero value for the `glen` member. Is this scenario useful?

28.2 When we ran the program in Figure 28.13 to the unreachable host `192.3.4.5` we said that we received an ICMP host unreachable in response to our SYNs. Why is SYNs plural?

28.3 At the beginning of Section `28.8` we described the scenario where an application calls `t_snd` but an RST has been received on the connection. Compare this to how the sockets API handles a `write` when an RST has been received on the connection.

28.4 Write a function named `xti_read` that has the same arguments as `read` but calls `t_rcv` and handles the two scenarios from Figure `28.13:` (a) if an orderly release is received, return 0, and (b) if a disconnect is received, return -1 with `errno` set to the reason.

29

XTI: Name and Address Functions

29.1 Introduction

XTI itself says nothing about name and address translation. The only functions required by Unix 98 in this area are the ones we covered in Chapter 9: `gethostbyname`, `gethostbyaddr`, `getservbyname`, and the like. Nevertheless, since many implementations of XTI are on SVR4-derived systems, most of these provide the name and address functions derived from SVR4: what we call the `netconfig` and `netdir` functions. These functions are called the "Network selection and name-toaddress mapping facility" in SVR4.

In our client in Section 28.11 we filled in an Internet socket address structure with an IP address and a port number. This is protocol dependent and our goal in this chap-ter is to avoid knowing the contents of the `netbuf` structure, handling it instead as an opaque structure. We should start with a hostname and a service name, call some functions, and the end result should be a `netbuf` structure, ready for a call to `t_connect`, for example, in a TCP client. This is similar to our use of the `getaddrinfo` function in Section 11.2.

> One problem with the omission of the functions that we will describe from any standard is the lack of a definitive description as to how they operate. For example, most implementations of `netdir_getbyname` accept either a name or a decimal port number for a TCP or UDP service name, but other implementations accept only a name and not a port number.

29.2 /etc/netconfig File and netconfig Functions

The starting point for XTI name and address mapping is the /etc/netconfig file. This is a text file with one line for each supported protocol. Some typical values for the fields for each protocol are shown in Figure 29.1.

Network ID	Semantics	Flags	Protocol family	Protocol name	Device
tcp	tpi_cots_ord	v	inet	tcp	/dev/tcp
udp	tpi_clts	v	inet	udp	/dev/udp
iomp	tpi_raw	-	inet	icmp	/dev/icmp
rawip	tpi_raw	-	inet	-	/dev/rawip
ticlts	tpi_clts	v	loopback	-	/dev/ticlts
ticots	tpi_cots	v	loopback		/dev/ticots
ticotsord	tpi_cots_ord	v	loopback)		/dev/ticotsordl
spx	tpi_cots_ord	v	netware	spx	/dev/nspx2
ipx	tpi_clts	v	netware	ipx	/dev/ipx I

Figure 29.1 Typical entries in the /etc/netconfig file.

There are actually seven fields for each line of the file but we do not show the final field, which specifies one or more libraries for directory lookups for that network. Typical values for this final field for the Internet protocols are /usr/ lib/ t cp ip . so or /usr/lib/ resoly. so. These are normally dynamically loadable libraries that provide the network-specific portion of the name-to-address translations.

The network IDs for the four Internet protocols are as we expect. The next three rows are loopback entries (ti stands for "transport independent"), and the final two rows are for the Novell Netware protocols (which we do not discuss in this text).

The values shown for the network semantics for the Internet protocols correspond to the service types shown in Figure 28.1, with the exception of tpi_raw, which is used for ICMP and raw IP. Note that TCP provides a connection-oriented service with an orderly release, as we saw in Figure 28.1.

The only flag currently defined is v, which means the entry is visible to the NETPATH library routines (described shortly).

The device name is used as the argument to t_open.

The network services library provides numerous functions to read the netconfig file. The function setnetconf ig opens the file and the function getnetconf ig then reads the next entry in the file. endnetconfig closes the file and releases any memory that was allocated.

> The term *network services library is* from System V and usually refers to the library that is specified to the linker as -1nsi. This library, say /usr/1ib/libnsl.so, contains all the XTI library functions as well as the functions we are about to describe.

```
#include <netconfig.h>

void *setnetconfig(void);
                                        Returns: nonnull pointer if OK, NULL on error
struct netconfig *getnetconfig(void *handle);
                                        Returns: nonnull pointer if OK, NULL on end-of-file
int endnetconfig(void *handle);
                                        Returns: 0 if OK, -1 on error
```

The pointer returned by `setnetconfig` (called *a handle) is* then used as the argument to the remaining two functions. Each entry in the file is returned as a `netconfig` structure:

```
struct netconfig {
  char            *nc_netid;         "tcp", "udp", etc. */
  unsigned long   nc_semantics; /* NC_TPI_CLTS, etc. */
  unsigned long   nc_flag;        !' NC_VISIBLE, etc. */
  char            *nc_protofmly; /* "inet", loopback", etc. */
  char            *nc_proto;      /* "tcp", "udp", etc. */
  char            *nc_device;     /* device name for network id */
  unsigned long   nc_nlookups; /* # of entries in nc_lookups */
  char            nc_lookups;     /* list of lookup libraries */
  unsigned long   nc_unused[8];
};
```

The first six members in this structure correspond to the six columns in Figure 29.1. If we wrote a program that looked like the following outline

```
void *handle;
struct netconfig *nc;

handle = setnetconfig();
while ( (nc = getnetconfig(handle)) != NULL) { /*
     print netconfig structure */
} endnetconfig(handle);
```

and assuming the `/etc/netconfig` file were as shown in Figure 29.1, nine `netconfig` structures would be printed, one per line of the figure, and in that order.

29.3 NETPATH Variable and netpath Functions

The `getnetconfig` function returns the next entry in the file, letting us go through the entire file, line by line. But for interactive programs (typically clients) we want the searching of the file limited only to the protocols that the user is interested in. This is done by allowing the user to set an environment variable named `NETPATH` and then using the following functions instead of the `netconfig` functions described in the previous section.

```
include <netconfig.h>

void *setnetpath(void);
                                        Returns: nonnull pointer if OK, NULL on error

struct netconfig *getnetpath(void *handle)
                                        Returns: nonnull pointer if OK, NULL on end-of-file
int endnetpath(void 'handle);
                                        Returns: 0 if OK, -1 on error
```

For example, we could set the environment variable with the KornShell as

```
export NETPATH=udp:tcp
```

Using this setting, if we coded a program as shown in the following outline

```
void *handle;
struct netconfig *nc;

handle ; setnetpath O ;
while ( (nc = getnetpath(handle)) != NULL) {
     i* print netconfig structure */
}
endnetpath(handle);
```

only two entries would be printed, one for UDP followed by one for TCP. The order of the two structures returned now corresponds to the order of the protocols in the environment variable, and not to the order in the `netconfig` file.

If the `NETPATH` environment variable is not set, all visible entries are returned, in their order in the `netconfig` file.

29.4 `netdir` Functions

The `netconfig` and `netpath` functions let us find a desired protocol. We also need to look up a hostname and a service name, based on the protocol that we choose with the `netconfig` or `netpath` functions. This is provided by the `netdir_getb ä ame` function.

```
#include <netdir.h>

int netdir_getbyname(const struct netconfig *ncp,
                     const struct nd_hostserv *hsp,
                     struct nd_addrlist **alpp);

                                        Returns: 0 if OK, nonzero on error

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

The first function converts a hostname and service name into an address. *ncp* points to a `netconfig` structure that was returned by `getnetconfig` or `getnetpath`. We must also fill in an `nd_hostsery` structure with the hostname and service name and pass a pointer to this structure as the second argument.

```
struct nd_hostsery {
   char *h_host;                /* hostname */
   char *h_serv;                /* service name */
};
```

The third argument points to a pointer to an `nd_addrlist` structure, and on success **alpp* contains a pointer to one of these structures:

```
struct nd_addrlist {
   int                  n_cnt:     /* number of netbufs */
   struct netbuf *n_addrs;         /* array of netbufs containing the addrs */
};
```

Notice that this `nd_addrlist` structure points to an array of one or more `netbuf` structures, each of which contains one of the host's addresses. (Recall that a host can be multihomed.)

For example, using an example similar to Figure 11.1, where the hostname is `bsdi` (which has two IP addresses) and the service name is `domain` (TCP and UDP ports 53), then Figure 29.2 shows the information returned by `netdir_getbyname`, assuming that the `netconf ig` structure used as the first argument to this function was for TCP.

We again assume that the format used by the provider to represent an Internet address is the sockaddr_in structure. While this is common, it is not required.

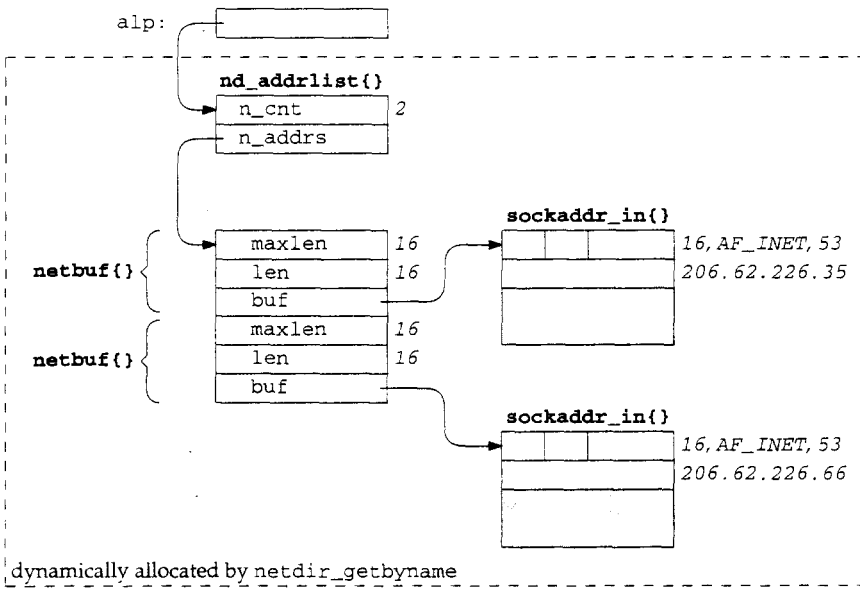

Figure 29.2 Data structures returned by netdir_getbyname.

The final argument to `netdir_getbyname` for this example would be a pointer to our `alp` variable.

When we have finished with these dynamically allocated structures, we call `netdir_free` with *ptr* pointing to the `nd_addrlist` structure, and *type* set to `ND_ADDRLIST`.

The reverse conversion—given a `netbuf` structure containing an address, return the hostname and service name—is provided by `netdir_getbyaddr`.

```
#include <netdir.h>

int netdir_getbyaddr(const struct netconfig *ncp,
                     struct nd_hostservlist **hslpp,
                     const struct netbuf *addr);
```

Returns: 0 if OK, nonzero on error

The first and third arguments provide the input: a pointer to a `netconfig` structure and a pointer to a `netbuf` structure. The result is a pointer to an `nd_hostservlist` structure, and this pointer is stored in **hslpp*.

```
struct nd_hostservlist {
   int                 h_cnt;       /* number of nd_hostservs
   struct nd_hostsery  *h_hostservs; /* the hostname/service-name pairs };
```

This structure in turn points to an array of one or more `nd_hostsery` structures. The memory for the `nd_hostservlist` structure, the array of `nd_hostsery` structures that it points to, and the hostname and service name strings that this last structure points to are all dynamically allocated. This space is freed by calling `netdir_free` with a *type of* `ND_HOSTSERVLIST`.

29.5 `t_alloc` and `t_free` Functions

One of the requirements for protocol independence with an API is knowing the size of the protocol's addresses without having to know the exact format of the address. With sockets, this size is provided by the `ai_addrlen` member of the `addrinfo` structure that is returned by the `getaddrinfo` function (Section 11.2). With XTI, this size is provided by the `addr` member of the `t_info` structure that is returned by the `t_open` function.

Given this size, the next step is to dynamically allocate the required structures. With sockets we *only* had to worry about socket address structures and we just called `malloc` when necessary (e.g., Figure 27.5). But with XTI there are six structures (Figure 28.6), each of which contains one or more `netbuf` structures. These `netbuf` structures point to a buffer whose size depends on the size of the protocol address (such as the `addr` member of the `t_call` structure in Section 28.6). To simplify the dynamic allocation of these XTI structures and the `netbuf` structures that they contain, the `t_alloc` and `t_free` functions are provided.

```
*include <xti.h>

void *t_alloc(int fd, int structtype, int fields);

                                        Returns: nonnull pointer if OK, NULL on error

int t_free (void *ptr, int structtype);

                                        Returns: 0 if OK, -1 on error
```

The *structtype* argument specifies which of the seven XTI structures is to be allocated or freed and must be one of the constants shown in Figure 29.3.

The *fields* argument lets us specify that space for one or more `netbuf` structures should also be allocated and initialized appropriately *fields* is the bitwise-OR of the constants shown in Figure 29.4. Recall from Figure 28.6 that the `netbuf` structure is always named `addr, opt,` or `udata.`

structtype	Type of structure
T_BIND	struct t_bind
T_CALL	struct t_call
T_DIS	struct t_discon
T_INFO	struct t _info
T_OPTMGMT	struct t_optmgmt
T_UDERROR	struct t_uderr
T_UNITDATA	struct t_unitdata

Figure 29.3 *structtype* argument for t_alloc and t_free.

fields	*Allocate and initialize*
T_ALL	all relevant fields of the given structure
T_ADDR	addr field of t_bind, t_call, t_uderr, or t_unitdata
T_OPT	opt field of t_optmgmt, t_call, t_uderr, or t_unitdate
T_UDATA	udata field of t_call, t_discon, or t_unitdata

The reason for these different values of the *fields* argument is that some of the XTI structures contain more than one `netbuf` structure, and we might not want to allocate space for all the buffers. For example, the `t_call` structure that we showed in Section 28.6, contains three `netbuf` structures.

```
struct t_call   {
   struct  netbuf   addr;        /* protocol-specific address */
   struct  netbuf   opt;         /* protocol-specific options */
   struct  netbuf   udata;       /* user data */
   int              sequence;    /* applies only to t_listen() func */
   };
```

Specifying some combination of `T_ADDR, T_OPT,` and `T_UDATA` gives us complete control of the allocation. We will normally use `TALL` for our examples, because this is the simplest. (See also Exercise 29.2.)

Given the values in Figure 28.1 for AIX 4.2, if we call `t_alloc` with an *fd* that refers to a TCP endpoint, *structtype* of `T_CALL`, and *fields* of `TALL`, we get the picture shown in Figure 29.5 on return.

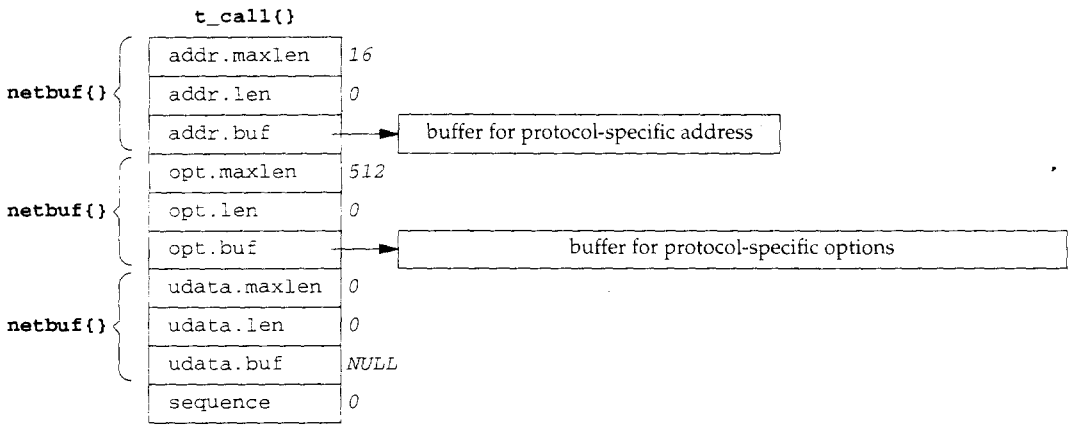

Figure 29.5 Allocation of structures and buffers by `t_alloc`.

This call to `t_alloc` allocates space for the `t_call` structure, which contains three `netbuf` structures. One buffer is allocated for the protocol-specific address (the `addr` member) and another for the protocol-specific options (the `opt` member). The two `buf` pointers are initialized along with the two `maxlen` members, and the two `len` members are set to O. The third `netbuf` structure is not used for TCP (the user data to accompany the connection request), so the two lengths in the `udata` member are set to 0 and the buffer pointer is set to a null pointer.

The `t_free` function frees a structure that was previously allocated by `t_alloc`. The *structtype* argument specifies the type of the structure, and the constants shown in Figure 29.3 are used for this. `t_free` not only frees the memory that was allocated for the structure specified by *structtype*, but it also first checks any `netbuf` structures contained therein and releases the memory used by those buffers. In our example in Figure 29.5, `t_free` would first release the two buffers, and then the `t_call` structure.

29.6 t_getprotaddr Functions

The `t_getprotaddr` function returns both the local and foreign protocol addresses associated with an endpoint.

```
#include <xti.h>
int t_getprotaddr (int fd, struct t_bind *localaddr, struct t_bind *peeraddr);

                                                    Returns: 0 if OK, -1 on error
```

The addr member (a netbuf structure) of the two t_bind structures is used by this function. When the function is called, the maxlen and buf members of the netbuf structures specify where the result is to be stored. A maxlen of 0 indicates that the corresponding address should not be returned. On return the ten members of the netbuf structures contain the size of the address that was stored in buf. This value will be 0 for the local address, if it has not yet been bound, and will be 0 for the peer address, if the endpoint has not yet been connected.

> TZI had an undocumented function named t_getname that could return both the local protocol address and the foreign protocol address.
>
> The t_getprotaddr function is a combination of both get sockname and getpeername.
>
> If we are interested in only one of the two addresses, we must still allocate a t_bind structure for the unwanted address and set its maxlen to O. A simpler design would allow us to specify a null pointer for either of the two arguments when we do not want that address returned.

29.7 xti_ntop Function

We want a simple way to print an XTI protocol address (simpler than netdir_getbyaddr) so we write our own function xti_ntop to do this, similar to our sock_ntop function in Section 3.8. Most XTI implementations provide two functions named taddr2uaddr and uaddr2taddr. The term taddr refers to a *transport address,* contained in a netbuf structure, and the term uaddr refers to a *universal address, a* human-readable text string, stored as a null-terminated C string. These implementations print IPv4 universal addresses as six decimal numbers separated by five decimal points, with the first four numbers being the dotted-decimal IPv4 address and the final two numbers being the 2 bytes of the TCP or UDP port number.

The problem with these two functions, however, is that they require an argument to a netconfig structure, which gives information about the protocol whose address is being converted. But XTI addresses for IPv4 and IPv6 are self-defining. For example, when an IPv4 addresses.is stored within a netbuf structure, the address is really a socket address structure and the first member is the address family, AF_INET. The length of this netbuf structure is 16 bytes (e.g., the addr row in Figures 28.1 and 28.2). We expect IPv6 addresses to be stored as sockaddr_in6 structures, with an address family of AF_INET6 and a length of 24 bytes. We are not guaranteed that all XTI addresses stored in netbuf structures are self-defining like this, but IPv4 and IPv6 addresses will be.

> Self-defining may be too strong a term. It is possible for some other protocol suite to use a 16-byte address whose first 2 bytes just happen to equal the constant AF_INET. But practically speaking, this should not be a problem.

We must then choose how to pass the protocol address to our function. Our first choice would be one of the XTI t_XXX structures. But for clients the protocol address of the server is in the addr member of a t_call structure (Section 28.6), for servers the

protocol address of the client is in the addr member of a t_bind structure (Section 30.2), and for any endpoint the local and foreign addresses returned by t_getprotaddr are in a t_bind structure. Since there is no consistency here (if we passed a pointer to one of these structures, we would also have to pass a flag indicating the type of structure), we will skip the XTI structures and pass a pointer to a netbuf structure to our function instead.

```
#include "unpxti.h"
char -xti_ntop(const struct netbuf *np);
```

Returns: nonnull pointer if OK, NULL on error

The argument is a pointer to a netbuf structure containing the address. The result is stored in static storage within the function. On success the return value is a pointer to the string containing the presentation format of the address.

We will use another function named xt i_ntop_ho s t, with the same calling sequence, that formats only the IP address, ignoring the port number.

> These two functions are similar to the code shown in Figure 3.13. We do not show the source code, but it is freely available (see the Preface).

29.8 tcp_connect Function

We can now combine the getnetpath function, which returns information about one or more protocols with the netdir_getbyname function, which looks up a hostname and service and redo our tcp_connect function from Section 11.8 to use XTI instead of sockets and getaddrinfo. We show this function in Figure 29.6.

Initialize

13-15 setnetpath opens the netconfig file. The nd_hostsery structure is initialized with pointers to the hostname and service name.

Get next entry from netconfig file

16-18 getnetpath searches the netconfig file for the next protocol in the NETPATH variable. If the protocol is not TCP, we ignore the entry. Since we are looking jor only the entry for TCP, we could call

```
ncp    getnetconfigent("tcp");
```

to locate just this entry. The call

```
freenetconfigent(ncp);
```

would then free the memory allocated by getnetconfigent. But since we would also like this code to work with IPv6, we go through the loop looking at each netconfig structure. Currently it is not known what the entry in the netconfig file will look like for TCP over IPv6, and how the XTI name functions will handle IPv6.

libxti/tcp_connect.c

```
 1 #include    "unpxti.h"
 2 int
 3 tcp_connect(const char *host, const char *serv)
 4 {
 5      int    tfd; i;
 6      void   *handle;
 7      struct t_call tcall;
 8      struct t_discon tdiscon;
 9      struct netconfig *ncp;
10      struct nd_hostserv hs;
11      struct nd_addriist *alp;
12      struct netbuf *np;

13      handle = Setnetpath();

14      hs.h_host = (char *) host;
15      hs.h_serv = (char *) serv;

16      while ( (ncp = getnetpath(handle)) != NULL) {
17          if (strcmp(ncp->nc_proto, "tcp") != 0)
18              continue;

19          if (netdir_getbyname(ncp, &hs, &alp) != 0)
20              continue;

21              /* try each server address */
22          for (i = 0, np = alp->n_addrs; i < alp->n_cnt; i++, np++) {
23              tfd = T_open(ncp->nc_device, O_RDWR, NULL);

24              T_bind(tfd, NULL, NULL);

25              tcall.addr.len = np->len;
26              tcall.addr.buf = np->buf;   /* pointer copy */
27              tcali.opt.len = 0; /* no options */
28              tcall.udata.len = 0;    /* no user data with connect */

29              if (t_connect(tfd, &tcall, NULL) == 0) {
30                  endnetpath(handle);     /* success, connected to server */
31                  netdir_free(alp, ND_ADDRLIST);
32                  return (tfd);
33              }
34              if (t_errno == TLOOK && t_look(tfd) == T_DISCONNECT) {
35                  t_rcvdis(tfd, &tdiscon);
36                  errno = tdiscon.reason;
37              }
38              t_close(tfd);
39          }
40          netdir_free(alp, ND_ADDRLIST);
41      }
42      endnetpath(handle);
43      return (-1);
44 }
```

Figure 29.6 tcp_connect function for XTI.

Search for hostname and service name

19-20 `netdir_getbyname` looks up the hostname and service name, using the `netconfig` structure returned by `getnetpath`.

Go through all server addresses

21-28 This loop tries each returned address for the server, calling `t_open`, `t_bind`, and `t_connect` for each address, until a connection is established, or until all the addresses have been tried. The `t_call` structure is initialized from the `netbuf` structure returned by `netdir_getbyname`.

Connection succeeds

29-33 If the connection succeeds, we clean up and return the connected descriptor. `endnetpath` frees the memory allocated for the `netconfig` structure and closes the `netconfig` file, and `netdir_free` frees all the memory starting with the `nd_addr list` structure (Figure 29.2).

Handle `t_connect` errors

34-38 If `t_connect` fails, we check for TLOOK and call `t_rcvdis` if the connection was refused. We set `errno` to the protocol-dependent error code for the caller to examine. The endpoint is closed.

Finished with all addresses

40-41 After all the addresses have been tried, the `nd_addrlist` structure and the array of `netbuf` structures pointed to by it are freed by `netdir_free`. The `while loop will` keep going through the `netconfig` file, possibly returning additional protocols to try.

`getaddrinfo` combines the call to `getnetpath` with the testing for the correct protocol or semantics, with the call to `netdir_getbyname`.

An XTI endpoint that fails connection establishment can still be used in another call to `t_connect`. That is, we could move the calls to `t_open` and `t_bind` outside the `for loop`, calling these two functions once for each time through the `while loop`. Naturally, we would also move the call to `t_close` outside the `for loop`. With sockets, however, when a call to `connect` fails, the socket is no longer usable and must be closed (see Figure 11.6, for example).

But there is a subtle problem with this approach with XTI. The problem appears when the host to which we are trying to connect has multiple addresses, and the connection establishment fails. In this scenario the local port never changes and each call to `t_connect` for the next address is delayed by an exponential backoff from the previous call to `t_aonnect`, because all these connection establishments are from the same local endpoint. That is, if the first call to `t_connect` fails, the next call to `t_connect` might be delayed by 1 second, and if this fails the next call to `t_connect` might be delayed by 2 seconds, and so on. To avoid this problem we `t_close` the endpoint when `t_connect` fails and then create a new endpoint for the next call to `t_connect`.

Example

We now use our `tcp_connect` function and redo our protocol independent daytime client from Figure 11.7 using XTI instead of sockets. Our XTI version is in Figure 29.7.

xtiintro/daytimec1i02.c

```
 1 #include    "unpxti.h"

 2 int
 3 main(int argc, char **argv)
 4 (
 5     int     tfd, n, flags;
 6     char    recvline[MAXLINE + 1];
 7     struct t_bind *bound, *peer;
 8     struct t_discon tdiscon;

 9     if (argc != 3)
10         err_quit("usage: daytimecli02 <hostname/IPaddress> <service/port#>");
11     tfd = Tcp_connect(argv[1], argv[2]);

12     bound = T_alloc(tfd, T_BIND, T_ALL);
13     peer = T_alloc(tfd, T_BIND, T_ALL);
14     T_getprotaddr(tfd, bound, peer);
15     printf("connected to %s\n", Xti_ntop(&peer->addr));
16     for ( ; ; ) {
17     if ( (n = t_rcv(tfd, recvline, MAXLINE, &flags)) < 0) {
18              if (t_errno == TLOOK) {
19                  if ( (n = T_look(tfd)) == T_ORDREL) {
20                      T_rcvrel(tfd);
21                      break;
22                  } else if (n == T_DISCONNECT) {
23                      T_rcvdis(tfd, &tdiscon);
24                      errno = tdiscon.reason;    /* probably ECONNRESET */
25                      err_sys("server terminated prematurely");
26                  } else
27                      err_quit("unexpected event after t_rcv: %d", n);
28              } else
29                  err_xti("t_rcv error");
30          }
31          recvline[n] = 0;        /* null terminate */
32          fputs(recvline, stdout);
33      }
34      exit(0);
35 }
```

xti in tro/daytimecl i02.c

Figure 29.7 Protocol independent daytime client.

Establish connection

11 **We** call our `tcp_connect` function from Figure 29.6 to look up the hostname and service name and establish the connection.

Print peer's protocol address

12-15 We allocate two `t_bind` structures and call `t_getprotaddr` to obtain the local protocol address and the peer's protocol address. We print the peer's address by calling our `xti_ntop` function.

Read data from server until EOF

16-33 The reading of the data from the server is identical to the code in Section 28.11.

We can run the program as follows:

```
unixware % daytimecli02 aix daytime
connected to 206.62.226.43.13
Fri Feb 7 13:28:24 1997
```

29.9 Summary

In SVR4 implementations of XTI network selection is normally done using the `/etc/netconfig` file and the function `netdir_getbyname` then looks up the host-name and service name, returning an array of `netbuf` structures, one per address and service. This is similar to the `getaddrinfo` function in Chapter 11. The reverse map-ping, from the protocol address to the presentation form, is done by `netdir_getbyaddr`, which is similar to `getnameinfo`.

Since so many structures are used by XTI, the seven `t_XXX` structures, and the `netbuf` structures contained therein, two functions are provided to dynamically allocate and free these structures: `t_alloc` and `t_free`.

Exercises

29.1 `getnetconfig` returns a pointer to a structure that it fills in, similar to `gethostbyname`. But we said the latter function was not thread-safe. Is `getnetconfig` thread-safe, and if so, how does it do this?

29.2 Write a program that calls `t_alloc` twice for a `t_call` structure for a TCP endpoint. The first time specify the third argument as `T_ALL` and the second time specify the third argument as `T_ADDR/T_OPT/T_UDATA`. What happens?

29.3 Why does `t_free` require a structtype argument?

29.4 In Figure 29.6 why don't we initialize the `nd_hostserv` structure as

```
struct nd_hostserv hs = { host, serv };
```

30

XTI: TCP Servers

30.1 Introduction

Undoubtedly the most confusing aspect of XTI is the handling of incoming connections by a connection-oriented server. With sockets we just call `accept` and all the details are handled by the kernel or by the sockets library. In the case of TCP, arriving SYNs from clients are placed onto an incomplete connection queue for that endpoint (Figure 4.6). When the three-way handshake completes, `accept` returns (Figure 2.5). If multiple connections are on the completed connection queue, they are returned in a FIFO order by `accept`. In the real world (Figure 4.9) the number of completed connections is normally 0 while the number of incomplete connections is often nonzero.

The *intent* of the XTI model (which is based on the design of the OSI transport service) is to allow the transport layer to tell the server process when a SYN arrives from a client (called *a connect indication),* passing the client's protocol address (IP address and port) to the server. The server process is then allowed to either accept or reject the connection request. The server's TCP, in this model, would not send its SYN/ACK or its RST until the server process tells it what to do. This model is shown in Figure 30.1.

Notice the server's function calls: the first call to `t_bind` (with a nonzero `glen)` indicates that the endpoint will be accepting incoming connections, `t_listen` returns when the connection is "available" (we say more about this shortly), and the server must then call `t_open, t_bind,` and `t_accept` to accept the connection. LISTEN indicates the endpoint's TCP state (Figure 2.4).

When the server process receives notification of the connection request it can also choose not to accept the connection, calling `t_snddis` to reject the request. We show this in Figure 30.2. The *intent* here is that the server is notified when the SYN arrives (the connect indication) and chooses not to accept the connection (perhaps based on the client's IP address or port number, or on the user data sent with the connection request,

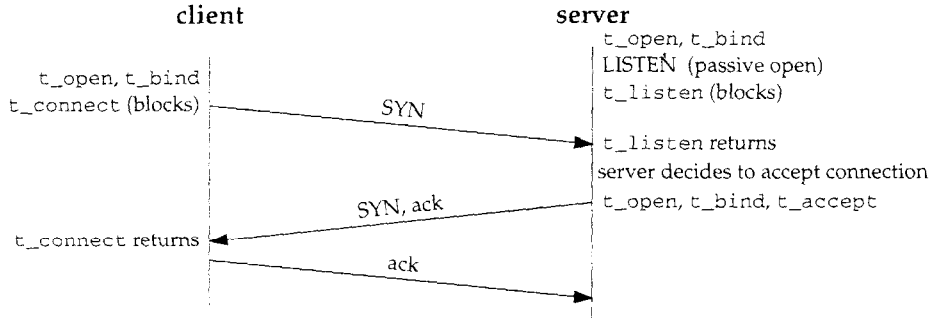

Figure 30.1 Intended model when XTI server accepts connection request.

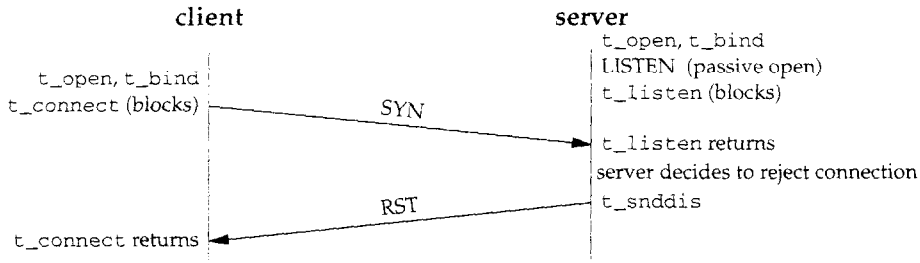

Figure 30.2 Intended model when XTI server rejects connection request.

if supported by the protocol). The application then calls t_snddi s, causing an RST to be sent instead of completing the three-way handshake. This would make the client's call to t_connect return an error.

> Recall from our sockets discussion and the time line in Figure 2.5 that a sockets server can never cause a client's `connect` to fail, because `accept` returns when the three-way handshake completes, one-half of an RTT after `connect` returns. If a sockets server doesn't like a client (perhaps based on the client's IP address or port returned by `accept),` all the server can do is terminate the connection, either with a normal `close` (sending a FIN) or by first setting the `So_LINGER` socket option and then calling `close` (sending an RST).

We have italicized the word *intent* when describing the XTI scenario because this is not what really happens. This scenario was the intent of the OSI protocols, but most existing TCP implementations automatically accept incoming connection requests (as long as the complete and incomplete connection queues are not full) and do not notify the server process until the three-way handshake is complete.

> The technique of notifying the application when the SYN arrives and then not completing the three-way handshake until the application indicates whether it wants to accept or reject the connection request is sometimes called *a lazy accept.* At least two historical implementations of TLI, from The Wollongong Group and Sequent Computer Systems, performed lazy accepts.

Both have changed to the de facto "standard" method of returning from t_listen when the three-way handshake completes. One reason for the change is that the lazy accept breaks most implementations of FTP.

Posix.lg also requires that when t_listen returns successfully for a TCP endpoint, this indicates a completed connection, and not a connect indication.

[Jacobson 19941 notes that 4.4BSD was supposed to provide a per-socket option to allow a lazy accept with the sockets API for TCP, but this was never implemented. 4.4BSD supports the lazy accept for the-OSI protocols.

30.2 t_listen Function

The normal scenario for a connection-oriented XTI server is to call the following functions.

```
listenfd = t_open( ... );           /* create listening endpoint */
t_bind(listenfd, ... );             /* t_bind.glen > 0 */

for ( ;  ; ) {
     t_listen(listenfd, ... );      /* blocks awaiting connection */
    connfd = t_open( ... );         /* create new fd for connected endpoint */
     t_bind(connfd, NULL, NULL);    /* any local addr */
          t_accept(listenfd, connfd, ... ); /* accept on new fd */
        ...                      /* process connected endpoint */t_close(connfd);
}
```

t_listen is the function that normally blocks, waiting for a connection from a client.

```
#include <xti.h>

int t_listen(int fd, struct t_call *call);

                                                        Returns: 0 if OK, -1 on error
```

We described the t_call structure when we described the t_connect function but show it again here:

```
struct t_call {
   struct netbuf addr;          /* protocol-specific address */
   struct netbuf opt;           /* protocol-specific options */
   struct netbuf udata;         /* user data to accompany connection request */
   int              sequence;   /* for t_listen() & t_accept() functions */
};
```

The structure returned through the *call* pointer contains relevant parameters of the connection: addr contains the protocol address of the client, opt contains any protocol-specific options, and udata contains any user data that was sent along with the connection request (which is not supported by TCP). The sequence variable contains a unique value that identifies this connection request. This value will be used when we call t_accept (or t_snddis) to identify which connection to accept (or reject).

Although this function appears similar to the accept function, it is different, as t_listen waits only for a connection to arrive; it does not accept the connection. To do that, the XTI user has to call the t_accept function.

Although *sequence is* an integer, some implementations store an address in this member. Do not assume it is a small integer like a descriptor.

30.3 tcp_listen Function

We now write our own function that creates a listening endpoint on which incoming connections can be accepted. The calling sequence is identical to the function of the same name shown in Figure 11.8.

Initialize

16-18 setnetconfig opens the /etc/netconfig file. If the host argument is a null pointer, we pass the special string HOST_SELF to netdir_getbyname. This causes the listening socket to be bound to the wildcard address (0.0.0.0 for IPv4).

Find matching protocol

19-22 We process each line of the /etc/netconfig file looking for the TCP protocol. Notice that we use getnetconf ig for a server, while in Figure 29.6 we called getnetpath for a client. This is because a server should not assume that the NETPATH environment variable is set to anything meaningful, since the server might be started by an initialization script or from the command line. Clients, on the other hand, are normally started from an interactive shell on behalf of a user and can assume the variable might be set by the user.

Look up hostname and service name

23-27 netdir_getbyname looks up the hostname and service name, using the pointer to the netconfig structure for the desired protocol.

Open *device*

28-29 t_open opens the appropriate device (such as /dev/tcp) and we save a copy of this device name in the external xti_sery dev. We do this because the caller of tcp_listen will need to call t_open again, once per connection, and needs this device name to maintain protocol independence. With the sockets version of tcp_listen we didn't need to save anything like this, because accept (not the process) automatically creates the new socket for each connection.

This technique is not thread-safe. This is an unfortunate side effect of requiring the application to maintain state information (the name of the device) between the call to t_open for the listening descriptor, and the later calls to t_open for each connected descriptor. One way to make this operation thread-safe is to call strdup to copy the device name into dynamically allocated storage and then return the pointer through another argument to tcp_listen, for the caller to free.

libxti/tcp_listen.c

```
 1 #include     "unpxti.h"
 2 #include     <limits.h>          /* PATH MAX */

 3 char     xti_serv_dev[PATH_MAX + 1];

 4 int
  5 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
  6{
  7     int     listenfd;
  8     void   *handle;
  9     char    *ptr;
 10     struct t_bind tbind;
 11     struct t_info tinfo;
 12     struct netconfig *ncp;
 13     struct nd_hostserv hs;
 14     struct nd_addrlist *alp;
 15     struct netbuf'*np;

 16     handle = Setnetconfig () ;

 17     hs.h_host = (host == NULL) ? HOST_SELF : (char *) host;
 18     hs.h_sery = (char *) serv;

 19     while ( (ncp = getnetconfig(handle)) != NULL &&
 20            strcmp(ncp->nc_proto, "tcp") != 0) ;

 21     if (ncp == NULL)
 22         return (-1);

 23     if (netdir_getbyname(ncp, &hs, &alp) != 0) {
 24         endnetconfig(handle);
 25         return (-2);
 26     }
 27     np = alp->n_addrs;          /* use first address */

 28     listenfd = T_open(ncp->nc_device, O_RDWR, &tinfo);
 29     strncpy(xti_serv_dev, ncp->nc_device, sizeof(xti_serv_dev));

 30     tbind.addr = *np;           /* copy entire netbuf{} */
 31         /* can override LISTENQ constant with environment variable */
 32     if ( (ptr = getenv("LISTENQ")) != NULL)
 33         tbind.glen = atoi(ptr);
 34     else
 35         tbind.glen = LISTENQ;
 36     T_bind(listenfd, &tbind, NULL);

 37     netdir_free(alp, ND_ADDRLIST);
 38     endnetconfig(handle);

 39     if (addrlenp)
 40         *addrlenp = tinfo.addr; /* size of protocol addresses */
 41     return (listenfd);
 42 }
```

Figure 30.3 XTI tcp_listen function: create listening endpoint.

libxti/tcp_listen.c

Enter TCP's LISTEN state

30-36 We call `t_bind`, binding the address returned by `netdir_getbyname` to the endpoint. By setting the `glen` member of the `t_bind` structure nonzero, this indicates a listening endpoint, and in the case of TCP the endpoint enters the LISTEN state. (We are talking about TCP's LISTEN state here. The XTI state is called `T_IDLE.`) Incoming connections will now be accepted by the transport provider. We let the environment variable `LISTENQ` override the default value of this constant from our `unp .h` header. We have similar code in our `Listen` wrapper function for the sockets `listen` function (Figure 4.8).

Free memory, return values

37-41 We call `netdir_free` and `endnetconfig` to free the allocated memory. We return the size of the protocol addresses (if requested) and the return value of the function is the listening endpoint.

Notice that we do not call `t_listen` as that is where the server blocks awaiting the incoming connection.

30.4 t_accept Function

Once the `t_listen` function indicates that a connection has arrived, we choose whether to accept the request or not. To accept the request the `t_accept` function is called.

```
#include <xti.h>
int t_accept ( int listenfd, int connfd, struct t_ca11 * call) ;
```

Returns: 0 if OK, –I on error

listenfd specifies the endpoint where the connection arrived; that is, this is the endpoint that was the argument to `t_listen.` The *connfd* argument specifies the endpoint where the connection is to be established. Normally the server creates a new endpoint, *connfd,* to receive the connection.

The *call* argument identifies which connection is being accepted (in case multiple connections are pending, which we talk about shortly), and its value is whatever was returned by `t_listen.`

Notice that it is the server's responsibility to create the new endpoint for a server. This is usually done by calling `t_open` between the calls to `t_listen` and `t_accept.`

We also have the option of specifying the same descriptor for *listenfd* and *connfd;* that is, we accept the new connection on the listening endpoint. But if we do this, no further connections are accepted by the provider until we have finished with this connection (e.g., this is an iterative server). It only makes sense in this scenario to set the q- to one. Given the limitations of this scenario, and the need of most real-world servers to handle multiple connections at the same time, we won't show any examples of this.

30.5 xti_accept Function

We now write a simple function named xti_accept to perform the steps required to accept a connection using XTI. In the common case we should write something like the following for our XTI server applications:

```
listenfd = Tcp_listen( ... ); /* create listening endpoint */
for ( ; ; ) {
    connfd = Xti_accept(listenfd, ... );   /* block, then accept */
     ...     /* process connfd */
    t_close(connfd);
}
```

This is similar to the sockets code, just replacing accept with xti_accept.

```
#include "unpxti.h"

int xti_accept (int listenfd, struct netbuf *cliaddr, int rdwr) ;
```

Returns: nonnegative descriptor if OK, -1 on error

On success, the return value is the new connected descriptor, the client's address is returned in the netbuf structure pointed to by *cliaddr*, and if the *rdwr* argument is nonzero, our xt i_rdwr function is called for the connected endpoint.

We show a simple version of our xti_accept function in Figure 30.4. We say "simple" because we will see shortly that it can fail when multiple connections are ready at the same time. We will fix this in Section 30.8.

Wait for connection

8-9 A t_call structure is allocated to hold the information about the client's connection. t_listen blocks, waiting for a connection.

Create new endpoint and bind any local address

10-12 A new endpoint is created by t_open, using the pathname that was saved in the external variable xti_serv_dev by tcp_listen. Any local address is bound to the endpoint. This call to t_bind is optional. If we do not bind something to the endpoint, it will be unbound when t_accept is called, and the communications provider will bind it automatically to some address that is appropriate.

Accept the connection

13 t_accept accepts the connection. t_accept knows which connection to accept by looking at the sequence member of the t_call structure, which was filled in by t_1isten to identify this particular connection.

Allow read and write, if desired

14-15 If the caller specifies a nonzero *rdwr* argument, our xti_rdwr function pushes the t i rdwr module onto the stream, allowing read and write to be used instead of t_rcv and t_snd.

libxti/xti_accept_simple.c

```
 1 #include    "unpxti.h"

 2 int
 3 xti_accept(int listenfd, struct netbuf *cliaddr, int rdwr)
 4 {
 5     int     connfd;
 6     u_int   n;
 7     struct t_call *tcallp;

 8     tcallp = T_alloc(listenfd, T_CALL, T_ALL);

 9     T_listen(listenfd, tcallp); /* blocks */

10          /* following assumes caller called tcp_listen () */
11     connfd = T_open(xti_serv_dev, O_RDWR, NULL);
12     T_bind(connfd, NULL, NULL);
13     T_accept(listenfd, connfd, tcallp);

14     if (rdwr)
15         Xti_rdwr(connfd);

16     if (cliaddr) {        /* return client's protocol address */
17         n = min (cliaddr->maxlan, tcallp->addr.len);
18         memcpy(cliaddr->buf, tcallp->addr.buf, n);
19         cliaddr->len = n;
20     }
21      T-free(tcallp, T_CALL);
22      return (connfd);
23      }
```

libxti/xti_accept_simple.c

Figure 30.4 Simple version of `xti_accept` function.

Return client's protocol address

16-20 The caller can specify a nonnull *cliaddr* argument that points to a `netbuf` structure. That structure must be initialized by the caller to point to a buffer in which the client's protocol address is returned. We ensure we do riot overflow the caller's buffer and then set the `len` member to the size of the address that we return.

Clean up and return

21-22 The `t_call` structure is freed and the connected descriptor is returned.

30.6 Simple Daytime Server

We now rewrite our simple daytime server from Figure 11.10 using XTI, calling our `tcp_listen` and `xti_accept` functions.

Create endpoint

l0-1? `tcp_listen` creates the listening endpoint. We allocate memory for the client's protocol address and initialize our `netbuf` structure for this purpose.

xtiin tro/daytimesrv0l .c

```
 1 #include    "unpxti.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     char    buff[MAXLINE];
 7     time_t ticks;
 8     socklen_t addrlen;
 9     struct netbuf cliaddr;

10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12     else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14     else
15        err_quit("usage: daytimetcpsrv0l [ <host> ] <service or port>");
16     cliaddr.buf = Malloc(addrlen);
17     cliaddr.maxlen = addrlen;

18     for ( ; ; ) {
19         connfd = Xti_accept(listenfd, &cliaddr, 0);
20         printf("connection from %s\n", Xti_ntop(&cliaddr));

21         ticks = time(NULL);
22         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
23         T_snd(connfd, buff, strlen(buff), 0);

24        T_close(connfd);
25     }

26 }
```

xtiin tro/daytimes rv01. c

Figure 30.5 Daytime server using XTI.

Wait for connection and accept it

19-20 Our `xti_accept` function waits for the connection, creates a new endpoint, returns the connected descriptor, and returns the client's IP address and port number. We print the client's protocol address using our `xti_ntop` function.

Generate daytime output

21-24 Calling `time` and then `ctime` generates the current time and date in a human-readable format, and `t_snd` sends this back to the client across the connection. The endpoint is closed with `t_close.`

Notice that we simply call `t_close` when have finished sending data. Since TCP provides an orderly release, this sends a FIN and goes through the normal four-packet connection termination sequence (Figure 2:5), but `t_close` returns immediately.

This is identical to calling `close` on a TCP socket.

If we want to wait until the peer TCP receives our data and sends its FIN, we must call `t_sndrel` to send our FIN and then wait for the FIN from the peer with `t_rcvre 1`. We would replace the call to `T_close` at the end of Figure 30.5 with the following:

```
T_sndrel(connfd);
while ( (n = t_rcv(connfd, buff, MAXLINE, &flags)) >= 0)

if (t_errno == TLOOK) {
     if ( (n = T_look(connfd)) == T_ORDREL) {
         T_rcvrel(connfd);
     }  else if (n == T_DISCONNECT) {
            T_rcvdis(connfd, NULL);
     }  else
         err_quit("unexpected event after t_rcv: %d", n);
} else
    err_xti("t_rcv error");
T_close(connfd);
```

`t_sndrel` sends the FIN and we must then wait for an orderly release indication at which time we call `t_rcvrel`. To do this we call `t_rcv`, ignoring any data that may arrive.

This scenario is similar to calling `shutdown` for a socket and then waiting until `read` returns an end-of-file (Figure 7.8).

With XTI we can also cause a `t_close` or `close` to linger, if there is still data queued to send to the peer, instead of returning immediately. This is done by setting the `XTI_LINGER` option, which we describe in Section 32.3. This is similar to the `SO_LINGER` socket option.

30.7 Multiple Pending Connections

We have alluded to the complexity involved when multiple connections arrive at nearly the same time for a listening endpoint. To demonstrate this problem we return to our TCP server in Figure 27.5. We used this server to measure the process control time required for various types of servers. We can run the client that we wrote for this server (Figure 27.4) and specify the number of children to fork, establishing multiple connections with the server.

Figure 30.6 shows the server, which is just Figure 27.5 coded to use XTI instead of sockets.

10-22 We call our `tcp_listen` and `xti_accept` functions that we developed earlier in this chapter.

`SIGINT` handler

31-37 Our signal handler calls our internal `xti_accept_dump` function, and we use this to collect the counters shown in Figure 30.13. This function prints the `count` member of each cli structure (Figure 30.7).

xtiserver/servOl.c

```
 1 #include     "unpxti.h"

 2 int
 3 main(int argc, char **argv)
 4 (
 5      int     listenfd, connfd;
 6      pid_t   childpid;
 7      void    sig_chld(int), sig_int(int), web_child(int);
 8      socklen_t addrlen;
 9      struct netbuf cliaddr;

10      if (argc == 2)
11          listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12      else if (argc == 3)
13          listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14      else
15          err_quit("usage: serv0l [ <host> ] <port#>");
16      cliaddr.buf = Malloc(addrlen);
17      cliaddr.maxlen = addrlen;

18      Signal (SIGCHLD, sig_chld);
19      Signal (SIGINT, sig_int);

20      for ( ; ; ) {
21        connfd = Xti_accept (listenfd, &cliaddr, 1);
22        printf("connection from %s\n", Xti_ntop(&cliaddr));

23      if ( (childpid = Fork() ) == 0) { /* child process */
24         Close(listenfd);       /* Close listening endpoint */
25         Web_child(connfd);     /* process the request */
26         Exit (0);
27      }
28      Close (connfd);           /* parent closes connected endpoint */
29      }
30      }

31 void
32 sig_int(int signo)
33 {
34     void   xti_accept_dump(void);

35     xti_accept_dump();
36     exit(0);
37 }
```

xtiserver/serv01.c

Figure 30.6 TCP concurrent server to demonstrate multiple connection problem.

If we start this server to listen on TCP port 9999

```
unixware % serv0l 9999
```

and run the client from another host as

```
solaris % client unixware 9999 1 600 4000
```

everything works fine. (1 is the number of children to `fork`, 600 is the number of connections per child, and 4000 is the number of bytes per connection.) Our server works because we tell the client to spawn only one child, so the connections arrive serially at the server from this one client.

If we change the third command-line argument from 1 to 2, however, we get the following error almost immediately from our server:

```
t_accept error: event requires attention
```

The problem is that two connections arrive at the server at about the same time—one connection from each of the two children. TCP's three-way handshake takes place for both connections because the server's TCP establishes the connections.

While the server TCP is establishing the connections, our server process is blocked in a call to `t_listen` from our server's call to `xti_accept`. When the first connection completes the three-way handshake, `t_listen` returns, and then `topen` and `t_accept` are called. But when `t_accept` is called for this first connection, the second connection has completed the three-way handshake and is also ready to be accepted. The rules of XTI now dictate that instead of `t_accept` completing the first connection, it returns an error with `t_errno` set to `TLOOK` ("event requires attention"). The event pending is `T_LISTEN` (a connect indication is pending) because there is another connection pending (the second connection).

What is happening here is that `t_accept` always returns art error if there is another connection ready. What we have to do is call `t_listen` to receive all the connect indications, saving the `t_call` structure for each connection, and then call `t_accept` for each connection.

> Why does XTI accept connections in this bizarre manner? If t_listen really returned when the client's SYN arrived (Figure 30.1), then forcing the server to call t_listen on all SYNs that have arrived, before calling t_accept for any single connection, gives the server process the opportunity to choose the order in which it accepts the pending connections. The server might prioritize them, for example, based on the IP address or port number returned by t_listen or based on the user data that might accompany the connection request (which is not supported by TCP). But given that t_1 is ten does not return for TCP until the three-way handshake completes, all this feature does is add (needless) complexity to the server.
>
> What we just described corresponds to XTI in Unix 95. Posix.ig and Unix 98 change the description of t_accept to say that it may fail with t_errno set to TLOOK. Nevertheless we must be prepared for t_accept to fail in this fashion.

30.8 `xti_accept` Function (Revisited)

We now redo our simple `xti_accept` function from Figure 30.4 into one that is more robust. There are two scenarios that we must handle.

- `t_accept` failing because there is another connection pending (`t_look` will return `T_LISTEN`), and
- `t_accept` failing because a pending connection has received an RST (`t_look` will return `T_DISCCNNECT`).

To handle these semantics we must maintain a queue of pending connections. The potential size of this queue is the value of qlim when we called `t_bind` for the listening endpoint. There are lots of possible data structures that we can use to keep track of the pending connections. For simplicity we will use a simple stack (array) of cli structures. Each structure contains the connected descriptor, a diagnostic counter of how often the structure is used, and a pointer to a `t_call` structure.

Let's assume that three clients establish connections with our server at about the same time. The server TCP will complete all three of the three-way handshakes and when the first one is returned by `t_listen` we use the `cli[0]` structure in our array, as shown in Figure 30.7.

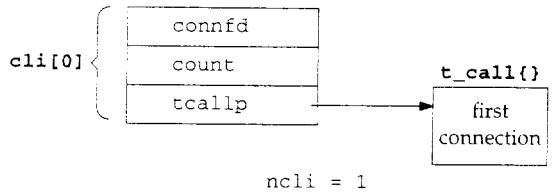

Figure 30.7 Data structures after first client connection returned by t_listen.

`connfd` is the new descriptor that we create by calling `t_open`, on which the connection will be accepted. `count` is a diagnostic counter that we examine shortly with our test program. `tcallp` is a pointer to a `t_call` structure that is filled in by `t_listen` and then passed to `t_accept`. It contains the client's protocol address (the `addr` member) and the connection identifier (the `sequence` member). We also keep a counter (`ncli`) of the number of entries in our array of `cli` structures, and its value would be 1.

Assume that we call `t_accept` to accept this first connection, but `t_accept` returns an error of `TLOOK` and `t_look` returns `T_LISTEN`. We must then call `t_listen` again to fetch the next connection. We just add another entry to our cli array and increment `ncli`. We show this in Figure 30.8.

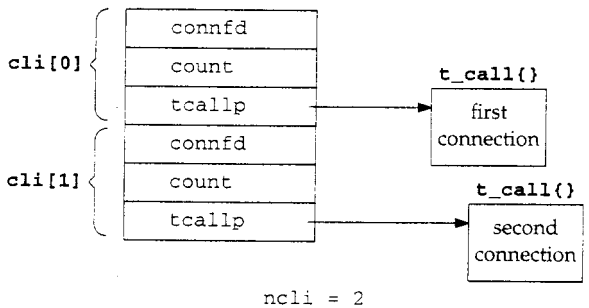

Figure 30.8 Data structures after second client connection returned by t_listen.

We always call `t_accept` for the "last" entry in the array, the one whose array index is `ncli-1`. This causes us to process the connections on a last-in, first-out basis (LIFO), instead of the first-in, first-out (FIFO) that one might expect. It is not hard to change this, if desired, but adds complexity.

Figure 4.9 shows that even on a moderately busy Web server, it is rare for more than one completed connection to be ready for the application to accept at any given time. Therefore the simplicity of our design is practical.

At this point we call `t_accept` for `cli[1]` but assume it also returns an error of `TLOOK` and t_look then returns `T_LISTEN`. We must add another entry to our array, `cli[2]`, as shown in Figure 30.9.

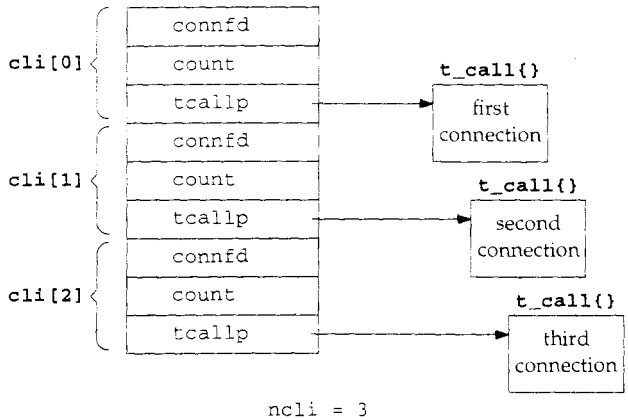

Figure 30.9 Data structures after third client connection returned by t_listen.

At this point we have three connections pending and we now call `t_accept` for `cli[2]`. But now assume that the first client (`cli[0]`) has just aborted its connection by sending an RST. Once again the call to `t_accept` fails with an error of `TLOOK`, but this time `t_look` returns `T_DISCONNECT`. We must now call `t_rcvdis` to receive the disconnect. Recall that one entry in the `t_discon` structure that this function fills in is the `sequence` identifier of the connection that was aborted (Section 28.10). We must use this to search our array of `cli` structures, looking for the entry whose `t_call` structure has a matching `sequence`. We then move the entries "up" by one, overwriting `cli[0]` with `cli[1]`, and then overwriting `cli[1]` with `cli[2]`. We also decrement `ncli` and have the data structures shown in Figure 30.10.

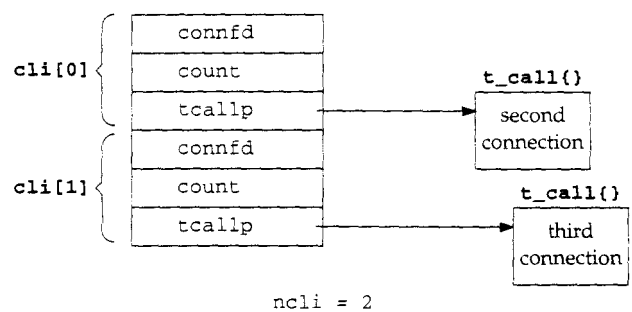

Figure 30.10 Data structures after first connection is aborted.

We call t_accept (again) for cli [1] but assume this time it succeeds. We then remove the cli [1] entry from the array, decrement ncli to become 1, and pass the third connection back to the caller of xti_accept. Remember that everything described so far in this example, starting with the first call to t_listen, has taken place in our xti_accept function. This is now the first time this function has returned a connected descriptor to the caller.

The next time xti_accept is called, ncli will be 1, and t_accept is called for cli [0]. Assuming it succeeds, the second connection is returned to the caller.

We now show the source code for our xti_accept function, the first half of which is in Figure 30.11.

libxti/xti_accept.c

```
 1 #include      "unpxti.h"
 2 static int ncli = -1, ndisconn;
 3 static struct cli {
 4      int      connfd;             /* connected fd or -1 if disconnected */
 5      int      count;
 6      struct t_call *tcallp;       /* ptr to t_alloc'ed structure */
 7 } *cli;                     /* cli[0], cli[l], ... , cli[ncli-1] are in use */

 8 int
 9 xti_accept(int listenfd, struct netbuf *cliaddr, int rdwr)
10 {
11      int      i, event;
12      u_int    n;
13      char     *ptr;
14      struct t_discon tdiscon;

15      if (ncli == -1) (               initialize first time through */
16          if (cli != NULL)
17              err_quit("already initialized");
18          if ( (ptr = getenv("LISTENQ")) != NULL)
19              n = atoi(ptr);
20          else
21              n = LISTENQ;
22          cli = Calloc(n, sizeof(struct cli));
23          for (i = 0; i < n; i++)
24              cli[i].tcallp = T_alloc(listenfd, T_CALL, T_ALL);
25          ncli = 0;
26      }
```

libxti/xti_accept.c

Figure 30.11 xti_accept function: first part.

Declare static variables

2-7 The counter ncli, another diagnostic counter of aborted connections ndisconn, and a pointer to our array of cli structures are declared as static.

Initialize the first time we are called

15-26 The first time we are called we allocate an array of cli structures, the number of entries in the array being the constant LISTENQ or the value of the environment variable of the same name. This will be the same value as tcp_1isten used in the call to

`t_bind`. For every element in the array we then call `t_alloc` to allocate a `t_call` structure and store the returned pointer in our cli structure.

> In a threads environment this one copy of the `on` array and its `ncli` counter must he protected to allow multiple threads to call `xti_accept` at the same time.

The second half of the function is shown in Figure 30.12.

Wait for a connection

:6-35 If our array is empty, we must call `t_listen` and wait for a connection. This call to `t_listen` is where the listening server spends most of its time. When `t_listen` returns, the client's protocol address and the connection identifier have been saved in the `t_call` structure by `t_listen`. We call `t_open` to create a new endpoint on which the connection will be accepted and bind this endpoint to any local address.

Call `t_accept;` return on success

36-46 We call `t_accept` to accept the connection specified by cli `[ncli-i1,` and if it succeeds we return this connected descriptor to the caller. We also call our `xti_rdwr` function, if the caller want to use `read` and `write,` and optionally return the client's protocol address.

Handle additional pending connections

47-53 If `t_accept` returns `TLOOK` and `t_look` returns `T_LISTEN,` another connection is pending and we must call `t_listen` to receive the information about this new connection. We note that this call will not block, since the endpoint has a `T_LISTEN` event pending. We also call `t_open` and `t_bind` and save the information in the array entry `cli [nciil.`

Handle disconnection of pending connection

54-66 If `t_accept` returns `TLOOK` and `t_look` returns `T_DISCONNECT,` one of the connections that has already been returned by `t_listen` has been aborted by the client_ We call `t_rcvdis` to obtain the information about the aborted connection (i.e., its `sequence)` and then search our array for the matching entry. When we find the matching entry we calculate the number of entries beyond the one being aborted `(n)` and call `memmove` to move the remaining entries. We call `memmove,` instead of `memcpy,` because the former correctly handles overlapping fields, which we will have in this scenario (see Exercise 30.3).

We can test our server from Figure 30.6 using this new version of `xti_accept` and it works as expected with multiple clients. We also want to see how often `t_accept` returns an error because of an additional pending connection. To see this we write a simple function that prints the `count` member of the cli structure, and the `ndisconn` counter (which we describe shortly), and then call this function from the server parent's `SIGINT` signal handler (Figure 30.6). Figure 30.13 (p. 814) shows the results with the server on UnixWare 2.1.2 and the client on Solaris 2.5.1. We vary the number of client children from 1 to 4, always issuing a total of 600 connections from all the children.

libxti/xti_accept.c

```
27      for ( ; ; )
28          if (ncli == 0) {        /* need to wait for a connection */
29              T_listen(listenfd, cli[ncli].tcallp);    /* block here

30                  /* following assumes caller called tcp_listen () */
31              cli[nclil.connfd = T_open(xti_serv_dev, O_RDWR, NULL);
32              Tbind(cli[ncli].connfd, NULL, NULL);
33 cli[ncli] count++;
34              ncli++;
35          }
36          if (t_accept(listenfd, cli[ncli - 1].conafd,
37                       cli[ncli   I] tcallp) == 0) {
38              ncli--;                 /* success */
39              if (rdwr)
40                  Xti_rdwr(cli_{ncli] .connfd);

41              if (cliaddr) (      /* return client's protocol address
42                  n = min(cliaddr->maxlen, cli[ncli].tcallp->addr.len);
43                  memcpy(cliaddr->buf, cli[ncli].tcallp->addr.buf, n);
44                  cliaddr->len = n;
45              }
46              return (cli[ncli].connfd);

47          } else if (t_errno == TLOOK) {
48              if ( (event = T_look(listenfd)) ==  T_LISTEN) {
49                  T_listen(listenfd, cli[ncli].tcallp);    /' won't block */
50                  cli[ncli].conrfd = T_open(xti_serv_dev, 0_RDWR, NULL);
51                  T_bind(cli[ncli].connfd, NULL, NULL);
52                  cli[ncli].count++;
53                  ncli++;

54              } else if (event == T_DISCONNECT) {
55                  T_rcvdis(listenfd, &tdiscon);
56                  for (i = 0; i < ncli; i++) {
57                       if (cli[i].tcallp->sequence == tdiscon.sequence) {
58                          T_close(cli[i].connfd);
59                          ndisconn++;
60                          ncli--;
61                          if ( (n = ncli - i) > 0)
62                              memmove(&cli[i], &cli[i + 1],
63                                      n * sizeof(struct cli));
64                          break;
65                       }
66                  }
67              } else
68                  err_quit("unexpected t_look event %d", event);
69          } else
70              err_xti("unexpected t_accept error");
71      }
72 }
```

libxti/xti_accept.c

Figure 30.12 xti_accept function: second half.

Server counter	#client children			
	1	2	3	4
cli[0].count	600	309	95	102
cli[1].count		291	286	121
cli[2].count			219	235
cli[3].count				142
Total	600	600	600	600

Figure 30.13 Counter of how often t_accept returns T_LISTEN.

Even with only two clients, each establishing 300 connections, one after the other, at about the same time, t_accept returns TLOOK with T_LISTEN about half the time.

> Why do we see multiple completed connections ready for the server to accept so often in this scenario, when we showed earlier (Figure 4.9) that on a busy Web server this is a rare occurrence? One reason is that we are forcing this scenario in Figure 30.13 with all the connections coming from a client on the same LAN. Second, the rate of the connections, *600* in about 12 seconds, corresponds to more than 4 million connections per day. Lastly, we purposely ran this example on a slow *server* (a 75-Mhz Pentium CPU) to test the handling of multiple pending connections with our xti_accept function. We can summarize this scenario as being infrequent enough in the real world so that handling them in a LIFO order by our xti_accept function is fine, but since the scenario can and does occur, the server must handle it.

To test this server with a client that aborts just-established connections we modify our client from Figure 27.4 by changing the innermost loop to the following:

```
    for (j = 0; j < nloops; j++) {
        fd = Tcp_connect(argv[1], argv[2]);

+       if (i == 2 && (j % 3) == 0) {
+           struct linger ling;
+
+           ling.l_onoff = 1;
+           ling.l_linger = 0;
+           Setsockopt(fd, SOL_SOCKET, SO_LINGER, &ling, sizeof(li.ng));
+           Close(fd);
+
+          /* and just continue on for this client connection ... */
+           fd = Tcp_connect(argv[1], argv[2]);
+       }

        Write(fd, request, strlen(request));

        if ( (n = Readn(fd, reply, nbytes)) != nbytes)
                    err_quit("server returned %d bytes", n);

        Close(fd);          /* TIME_WAIT on client, not server */
    }
```

The lines preceded by a plus sign are new. This modification causes the third child (i equals 2) to abort every third just-completed connection. To send the RST we set the SO_LINGER socket option accordingly and close the socket. We then create another

connection and continue with the loop. The effect on the server depends on the timing: some RSTs may arrive between the server's call to `t_listen` and its call to `t_accept` (and we count these with our `ndisconn` counter to verify that the code is exercised). Others may arrive before `t_listen` notifies the server of the connection, and others may arrive after the connection is accepted.

XTI Queue Length versus Listen Backlog

The XTI queue length and the `listen` backlog are similar but not identical. First, there has never been an exact specification of what the `listen` backlog means. We saw in Figure 4.10 that current systems differ in their interpretation.

Posix.lg states that the XTI queue length value specifies the number of "outstanding connect indications" that the provider should support for the endpoint. An out-standing connect indication is one that has been passed to the application by the provider but not yet accepted or rejected. The provider may queue more connect indications than specified but must ensure that there are never more than `glen` delivered to the application that are still outstanding at any given time.

If the implementation passed connect indications to the application when they arrived (i.e., when the client SYN arrives at the server), then this form of application queueing might make sense. But given that a TCP connection is completely established before the application is notified, there is no real need for the application to queue these connections.

As usual, the way to make any sense out of standards is to see what the implementations really provide when we specify different values for the `glen`. We modified Figure E.15 to work with XTI instead of sockets and ran the program on our five systems that support XTI. The results are shown in Figure 30.14.

	AIX 4.2		DUnix 4.0B		HP-UX 10.30		Solaris 2.6		UWare 2.1.2	
request glen	return glen	actual #conns	return glen	actual #conns	return glen	actual #conns	return glen	actual #conns	return glen	actual #conns
0	0	0	0	0	0	0	0	0		0
1	1	3	1	2	1	1	1	1		1
2	2	6	2	4	2	2	2	2		2
3	3	8	3	6	3	3	3	3		3
4	4	11	4	8	4	4	4	4		4
5	5	13	5	10	5	5	5	5		5
6	5	13	6	12	6	6	6	6		6
7	5	13	7	14	7	7	7	7		7
8	5	13	8	16	8	8	8	8		8
9	5	13	9	18	9	9	9	9		9
10	5	13	10	20	10	10	10	10		10
11	5	13	11	22	11	11	11	11		11
12	5	13	12	24	12	12	12			12
13	5	13	13	26	13	13	13	13		13
14	5	13	14	28	14	13	14	14		14

Figure 30.14 Actual number of queued connections for values of XTI glen.

Recall from Section 28.5 that the third argument to `t_bind` is a pointer to a `t_bind` structure that is filled in upon return by the provider. By looking at the `glen` member of this structure we can see what the provider sets this to. (XTI calls this a "negotiated" value.) We note that most systems return the value that was specified, unless a smaller value is supported (AIX). One system (UnixWare) does not return the value.

None of the systems allow any connections when the `glen` is 0 (which differs from a `listen` backlog of 0 in Figure 4.10), and two of the systems returned an error from `t_connect` (HP-UX and Solaris). Some of the implementations queue more connections than specified by `glen` (AIX and Digital Unix), but the remaining three do not.

> Here we are measuring the number of connections queued by the provider, not by the application, but it is this queueing by the provider in which we are most interested.

Setting the Server's Listening Queue Length to 1

One way to avoid the complication involved in accepting XTI connections is to set the `glen` member of the `t_bind` structure to 1. But the problem with this solution is that many implementations will then queue only one client connection (Figure 30.14) and then ignore all other arriving SYNs until this connection is accepted.

We can test this feature with our server from this section. We again run the server on UnixWare 2.1.2, which queues only one connection when the specified `glen is 1.` Like the numbers in Figure 30.13, we vary the number of client children that are issuing connections between 1 and 4, but this time we measure the clock time (in seconds) required for the fixed number of connections (600).

Figure 30.15 Clock time for 600 total connections, varying number of children and listen queue.

Queue length	#client children			
	1	2	3	4
1	10.6	12.2	15.6	13.2
1024	10.6	10.2	10.3	10.4

For a queue length of 1, the clock time increases as the number of children increases. This is caused by many of the client SYNs being ignored by the server, because one connection has filled the queue, and the client must retransmit the SYN. But when the queue length is greater than the number of simultaneous connections, the clock time decreases for the few number of children that we are testing here.

These numbers verify our previous statement: a queue length of 1 is unrealistic for a real-world server.

30.9 Summary

Accepting client connections with XTI is much harder than the same operation with sockets. As we described, the reason for the added complexity is to allow protocols to provide a lazy accept, where the application is notified when the connect request

arrives, and not when the connection establishment is complete. TCP implementations do not provide a lazy accept, and Unix 98 no longer requires t_accept to return an event of T_LISTEN when another connection is pending, but for backward compatibility XTI servers must handle this scenario.

Exercises

30.1 We mentioned in Section 30.2 that some implementations store a pointer in the sequence member of the t_call structure that is filled in by t_iisten. What happens on a 64-bit architecture?

30.2 Why do we add one to PATH_MAX in the declaration of xti_serv_dev in Figure 30.3?

30.3 In Figure 30.12 we call memmove and mention that this is needed since the source and destination overlap. Assume a 4-byte array, with elements x [01 through x [3] (drawn from left to right) and assume that we want to delete x [1] , moving the next two elements "left" by 1 byte, leaving three elements. Draw a picture of the source field and destination field. Then describe what happens if the copy operation starts from the beginning of the source field to the beginning of the destination (copying from right to left). Then describe what happens if the copy operation starts from the end of the source field to the end of the destination (copying from left to right). Does memcpy guarantee in which direction the copy takes place?

30.4 Recode Figure E.15 to use XTI instead of sockets.

30.5 Recode Figures 30.11 and 30.12 to use a linked list of c 1 i structures instead of the fixed-size array that we used for simplicity. Allocate the structures dynamically.

31

XTI: UDP Clients and Servers

31.1 Introduction

XTI provides three functions for connectionless clients and servers: `t_sndudata` to send a datagram, `t_rcvudata` to receive a datagram, and `t_rcvuderr` to obtain information about an asynchronous error. With sockets we had the option of calling `connect` for a UDP application, but XTI does not provide that choice.

31.2 `t_rcvudata` and `t_sndudata` Functions

These two functions are used with connectionless protocols (such as UDP) to receive and send datagrams.

```
#include <xti.h>

int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flagsp) ; int

t_sndudata(int fd, struct t_unitdata *unitdata) ;
```

Both return: 0 if OK, -1 on error

For the `t_sndudata` function the `t_unitdata` structure specifies the destination address, any options, and the actual data to send.

```
struct t_unitdata
  struct   netbuf    addr;      /*  protocol-specific address */
   struct  netbuf    opt;        /* protocol-specific options */
   struct  netbuf    udata;      /* user data */
```

For the t_rcvudata function this structure specifies where to store the sender's protocol address, any received options, and the actual data.

Both of these functions return 0 if everything is OK, or -1 on an error. This differs from most read and write functions that usually return the number of bytes transferred. With t_rcvudata the size of the received datagram is returned as the udata .len member of the t_unitdata structure. t_sndudata does not return the number of bytes written; it just returns 0 on success (i.e., the entire datagram has been copied into the kernel's buffers).

The integer pointed to by *flagsp* is similar to the final argument to t_rcv: it is not a value—result argument because its value is not examined by the function; it is set only on return. The T_MORE flag is returned if another call to t_rcvudata is required to read more of the datagram (i.e., what remains of the datagram exceeds the length of the receive buffer). We show an example of this in Section 31.6.

These two XTI functions correspond to the sendto and recvfrom functions.

31.3 udp_client Function

Before showing a UDP example using XTI we will write a function named udp_client, with the same calling sequence as shown in Section 11.10, that creates an XTI endpoint for a UDP client. This function, shown in Figure 31.1, handles the host-name and service name conversions described in Chapter 29.

Look up hostname and service name

12-19 The calls to getnetpath and netdir_getbyname are similar to the calls described in Figure 29.6.

Open device, bind any local address

22-21 t_open opens the appropriate device and t_bind binds any local address to the endpoint.

Allocate tunitdata structure

22 t_alloc allocates a t_unitdata structure but only the addr structure within, not the opt or udata structures. We do not allocate the opt structure because perdatagram options are rare with UDP (Chapter 32). We do not allocate the udata structure because the range of UDP datagram sizes is large, up to 65507 bytes as' shown in Figure 28.2, but few applications deal with these maximum-sized datagrams. Since most applications deal with smaller datagrams (a few thousand bytes at the most), it makes more sense for the application to allocate the data buffer itself, based on its needs.

Use first returned address for server

23-25 The addr structure is filled in with the first address returned for the server. Figure 31.2 (p. 822) shows the data structures involved, assuming that two netbuf structures are returned for the specified hostname and service name, and assuming that the implementation uses the sockaddr_i.n structure to represent IPv4 addresses.

libxti/udp_client.c

```
 1 #include    "unpxti.h"

 2 int
 3 udp_client(const char *host, const char *serv, void **vptr, socklen_t *lenp)
 4 {
 5     int     tfd;
 6     void    *handle;
 7     struct netconfig *ncp;
 8     struct nd_hostserv hs;
 9     struct nd_addrlist *alp;
10     struct netbuf *np;
11     struct t_unitdata *tudptr;

12     handle = Setnetpath();

13     hs.h_host = (char *) host;
14     hs.h_serv = (char *) serv;

15     while ( (ncp = getnetpath(handle)) != NULL) {
16         if (strcmp(ncp->nc_proto, "udp") != 0)
17             continue;

18         if (netdir_getbyname(ncp, &hs, &alp) != 0)
19             continue;

20         tfd = T_open(ncp->nc_device, O_RDWR, NULL);

21         T_bind(tfd, NULL, NULL);

22         tudptr = T_alloc(tfd, T_UNITDATA, T_ADDR);

23         np = alp->n_addrs;      /* use first server address */
24         tudptr->addr.len = min(tudptr->addr.maxlen, np->len);
25         memcpy(tudptr->addr.buf, np->buf, tudptr->addr.len);

26         endnetpath(handle);
27         netdir_free(alp, ND_ADDRLIST);

28         *vptr = tudptr;          /* return pointer to t_unitdata{} */
29         *lenp = tudptr->addr.maxlen;      /* and size of addresses *!
30         return (tfd);
31     }
32     endnetpath(handle);
33     return (-1);
34 }
```

Figure 31.1 udp_client function for Kn.

l ibxti/ud p_cl ient.c

The addr . maxl en value should be the same as the maxi en values in the structure returned by netdir_getbyname (16 for IPv4), but we use the min macro to be certain we do not overflow the destination of the memcpy. We show four lengths of 0 and two null pointers for the opt and udata structures, since t_alloc initializes these members to these values because we told it to allocate and initialize only the addr structure with the T_ADDR argument.

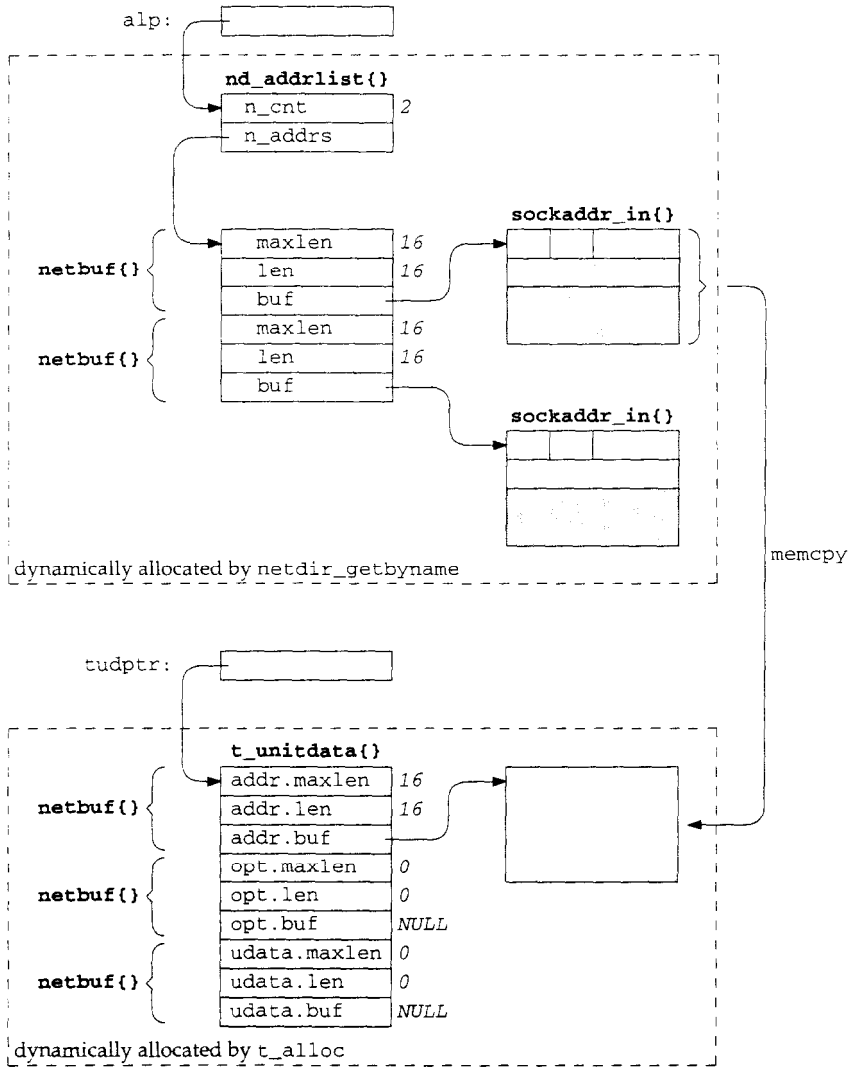

Figure 31.2 Data structures during call to `udp_client`.

Free memory and return

26-33 `endnetpath` frees the memory allocated for the `netconfig` structure and `netdir_free` releases the memory that `netdir_getbyname` allocated (Figure 31.2). The pointer to the `t_unitdata` structure is returned to the caller, along with the size of the protocol addresses and the descriptor for the endpoint. We return the size of the protocol addresses as `addr . maxl en` instead of `addr . l en,` because this value is returned for the caller to use in a call to `malloc.` Should the addresses be variable-length, we should return the maximum size, and not just the size of this address.

Example: Daytime Client

We now use our udp_client function to recode our protocol-independent daytime client from Figure 11.12 to use XTI, which we show in Figure 31.3.

xtiuudp/daytimetldpcl il.c

```
 1 #include      "unpxti.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5      int     tfd, flags;
 6      char    recvline[MAXLINE + 1];
 7      socklen_t addrlen;
 8      struct t_unitdata *sndptr, *rcvptr;

 9      if (argc != 3)
10          err_quit("usage: daytimeudpcli <hostname> <service>");

11      tfd = Udp_client(argv[1], argv[2], (void **) &sndptr, &addrlen);

12      rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);

13      printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));

14      sndptr->udata.maxlen = MAXLINE;
15      sndptr->udata.len = 1;
16      sndptr->udata.buf = recvline;
17      recvline[0] = 0;              /* 1-byte datagram containing null byte */
18      T_sndudata(tfd, sndptr);

19      rcvptr->udata.maxlen = MAXLINE;
20      rcvptr->udata.buf = recvline;
21      T_rcvudata(tfd, rcvptr, &flags);
22      recvline[rcvptr->udata.len] = 0;        /* null terminate */
23      printf("from %s: $s", Xti_ntop_host(&rcvptr->addr), recvline);


24      exit(0);
25 }
```

xtiudp/daytimeudpcl i I .c

Figure 31.3 UDP daytime client using XTI and our udp_client function.

Create endpoint

11-13 We call our udp_client function to create the XTI endpoint and to allocate a t_unitdata structure that we will use for sending datagrams. We allocate another t_unitdata structure that we will use for receiving replies. We call our xti_ntop_host function to print the server's IP address.

Send datagram

14-18 We initialize the udata structure within the t_unitdata structure to point to our recvline buffer and to contain one byte of O. t_sndudata sends the datagram to the server.

> We should be able to send a 0-byte UDP datagram, given our discussion with Figure 28.4, but many implementations of XTI do not allow this.

Read reply

19-23 We initialize the `udata` structure for the receiving `t_unitdata` structure and call `t_rcvudata` to read the server's reply. The reply is null terminated and printed to standard output, along with the server's address.

This example shares all the unreliable UDP properties of our sockets client in Figure 11.12: the client will block forever in the call to `t_rcvudata` if there is no response. If we run the client to a host that is running the server, the output is as we expect.

```
Un xware S daytimeudpclil bsdi daytime
sending to 206.62.226.35
from 206.62.226.35: Fri Feb 28 17:23:40 1997
```

What if we send a datagram to this same host, but to a UDP port that no process has bound? We expect an ICMP port unreachable error to be returned. Recall with our sockets client, if the client does not call `connect`, this error is not returned to the client. There is nothing similar to `connect` for a UDP endpoint with XTI, but we see that the error is returned to our client and our `T_rcvudata` wrapper function prints an error:

```
unixware % daytimeudpclil bsdi 9999
sending to 206.62.226.35
t_rcvudata error: event requires attention
```

When an asynchronous error is received for a UDP endpoint, `t_rcvudata` returns an error of `TLOOK` and we must call `t_rcvuderr` to determine the actual error. We discuss this in the next section.

31.4 `t_rcvuderr` Function: Asynchronous Errors

For a connectionless protocol, errors can be returned asynchronously. That is, a datagram can be correctly transmitted by the protocol stack, only to have an error in it detected somewhere else in the network. Common errors with UDP datagrams are to elicit an ICMP port unreachable from the destination host or an ICMP host unreachable from some intermediate router. When this ICMP error is received at some later time by the provider, it requires some form of notification from the provider to the process and some means for the process to determine the actual error. As shown at the end of the previous section, XTI provides this notification by setting `t_errno` to `TLOOK` when we call `t_rcvudata` to indicate that an error occurred on a previously sent datagram. We can then call the `t_rcvuderr` function to determine what happened and to clear the error status.

```
#include <xci.h>

int t_rcvuderr (int fd, struct t_uderr *uderr);

                                                    Returns: 0 if OK, -1 on error
```

If the *uderr* pointer is nonnull, a t_uderr structure is filled in with information about the error.

```
struct t_uderr {
  struct netbuf  addr;     /* protocol-specific address */
  struct netbuf  opt;      /* protocol-specific Options */
  t_scalar_t     error;    /* protocol-specific error */
};
```

The addr structure contains the destination address of the datagram that caused the error, the opt structure contains any protocol-specific options from the datagram that caused the error, and error contains a protocol-specific error code. For UDP, error is normally one of the errno values from <sys/errno.h>.

If *uderr is* a null pointer, this clears the error status without returning any information.

Example: ICMP Port Unreachable

We now recode our client from Figure 31.3 to handle asynchronous errors. This is shown in Figure 31.4.

Handle asynchronous errors

23-33 What has changed from Figure 11.12 is the call to our wrapper function T_rcvudata is replaced with a call to t_rcvudata, and we handle an asynchronous error by calling t_rcvuderr, printing the returned error code.

If we run this program and send a datagram to a host that does not support the daytime protocol, we receive the ICMP port unreachable error from t_rcvuderr:

```
unixware % daytimeudpcli2 gateway.tuc.noao.edu daytime
sending to 140.252.104.1
error 146 for datagram sent to 140.252.104.1

unixware % grep 146 /usr/include/sys/errno.h
#define ECONNREFUSED    146     /* Connection refused */
```

We see that the error value returned in the t_uderr structure is the errno value corresponding to the ICMP error (Figure A.15).

Unfortunately, as nice as this design feature appears (returning ICMP errors for XTI UDP endpoints), there are still problems. First, there is no requirement that the provider notify the application when these errors occur. With UnixWare 2.1.2, for example, ICMP port unreachables are returned to the application, but ICMP host unreachables are not. Also, if we modify our client to send three datagrams to three different servers and then read all the replies, but two of the datagrams elicit an ICMP port unreachable error, only the first of these two errors is returned to the application by t_rcvuderr. This is because the provider maintains only one error per endpoint. All of these problems are what led us to develop an independent way of notifying a datagram application of asynchronous errors: our icmpd daemon in Section 25.7.

Notice that all we receive is the error code and the destination address of the datagram that caused the error. Another piece of information that is not returned is the source address of who returned the error (e.g., the source address of the ICMP error).

xtiudp/daytimeudpcli2.c

```
 1 #include    "unpxti.h"

 2 int
 3 main (int argc, char **argv)
 4 {
 5      int     tfd,  flags;
 6      char    recvline[MAXLINE +  1];
 7      socklen_t addrlen;
 8      struct t_unitdata *sndptr, *rcvptr;
 9      struct t_uderr *uderr;
10     if (argc != 3)
11         err_quit("usage: a.out <hostname or IPaddress> <service or port#>");

12     tfd = Udp_client(argv[1], argv[2], (void **) &sndptr, &addrlen);

13     rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
14     uderr = T_alloc(tfd, T_UDERROR, T_ADDR);

15     printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));

16     sndptr->udata.maxlen = MAXLINE;
17     sndptr->udata.len = 1;
18     sndptr->udata.buf = recvline;
19     recvline[0] = 0;             /* 1-byte datagram containing null byte */
20     T_sndudata(tfd, sndptr);

21     rcvptr->udata.maxlen = MAXLINE;
22     rcvptr->udata.buf = recvline;
23     if (t_rcvudata(tfd, rcvptr, &flags) == 0) {
24         recvline[rcvptr->udata.len] = 0;    /* null terminate */
25         printf("from %s: %s", Xti_ntop_host(&rcvptr->addr), recvline);
26     } else {
27         if (t_errno == TLOOK) {
28             T_rcvuderr(tfd, uderr);
29             printf("error old for datagram sent to %s\n",
30                    uderr->error, Xti_ntop_host(&uderr->addr));
31         } else
32             err_xti("t_rcvudata error");
33     }
34     exit(0);
35 }
```

31.5 udp_server Function

We can also recode our udp_server function from Figure 11.14 to use XTI, and we show this in Figure 31.5.

libxti/udp_server.c

```
 1 #include    "unpxti.h"

 2 int
 3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int     tfd;
 6     void    *handle;
 7     struct t_bind tbind;
 8     struct t_info tinfo;
 9     struct netconfig *ncp;
10     struct nd_hostserv hs;
11     struct nd_addrlist *alp;
12     struct netbuf *np;

13     handle = Setnetconfig () ;

14     hs.h_host = (host == NULL) ? HOST_SELF:   (char *) host;
15     hs.h_serv = (char *) serv;

16     while ( (ncp = getnetconfig(handle)) != NULL &&
17            strcmp(ncp->nc_proto, "udp") != 0) ;

18     if (ncp == NULL)
19         return (-1);

20     if (netdir_getbyname(ncp, &hs, &alp) != 0)
21         return (-2);
22     np = alp->n_addrs;           /* use first address */

23     tfd = T_open(ncp->nc_device, O_RDWR, &tinfo);

24     tbind.addr = *np;           /*  copy entire netbuf {}
25     tbind.qlen = 0;              /* not used for connectionless server */
26     T_bind(tfd, &tbind, NULL);

27     endnetconfig(handle);
28     netdir_free(alp, ND_ADDRLIST);

29     if (addrlenp)
30         *addrlenp = tinfo.addr; /* size of protocol addresses */
31     return (tfd);
32 }
```

libxti/udp_server.c

Figure **31.5** udp_server function using XTI.

Lookup protocol, host, and service

13-22 The beginning of the function is similar to our `tcp_listen` function (Figure 30.3), finding the protocol by calling `getnetconfig` and then calling `netdir_getbyname` to look up the hostname and service name.

Open device, bind server's IP address and port

23-28 `t_open` opens the correct device and `t_bind` binds the server's IP address (the wildcard address if the `host` argument is a null pointer) and port. The memory allocated by `netdir_getbyname` is returned by `netdir_free`.

Return address length and descriptor

29-31 The size of the protocol's addresses is returned if the final argument is a nonnull pointer, and the descriptor for the endpoint is the return value of the function.

Example: Daytime Server

We can now recode our simple daytime server from Figure 11.15 using XTI. We show this in Figure 31.6.

xtiudpldaytimeudpsrv2.c

```
 1 #include     "unpxti.h"
 2 #include     <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     tfd, flags;
 7     char    buff[MAXLINE);
 8     time_t ticks;
 9     struct t unitdata *tud;

10     if (argc == 2)
11         tfd = Udp_server(NULL, argv[l], NULL);
12     else if (arge == 3)
13         tfd = Udp_server(argv[1], argv[2], NULL);
14     else
15         err_quit("usage: daytimeudpsry [ <host> ] <service or port>");

16     tud = T_alloc(tfd, T_UNITDATA, T_ADDR);

17     for ( ; ; ) {
18         tud->udata.maxlen = MAXLINE;
19         tud->udata.buf = buff;
20         if (t_rcvudata(tfd, tud, &flags) == 0) {
21             printf("datagram from %s\n", Xti_ntop(&tud->addr));
22             ticks = time(NULL);
23             snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24             tud->udata.len = strlen(buff);
25             T_sndudata(tfd, tud);

26         } else if (t_errno == TLOOK)
27             T_rcvuderr(tfd, NULL); /* just clear error */
28         else
29             err_xti("t_rcvudata error");
30     }
31 }
```

Figure 31.6 UDP daytime server using XTI. *xtiudp/daytimeudpsrv2.c*

Create XTI endpoint

10-16 Our udp_server function creates the endpoint and binds the local IP address and port. We allocate a t_unitdata structure by calling t_alloc, specifying the T_ADDR argument, so that it allocates a buffer for only the protocol address.

Read request, send reply

17-30 The program then loops, reading a client request with `t_rcvudata` and sending a reply with `t_sndudata`. If one of our replies generates an asynchronous error, `t_rcvudata` will return an error with `t_errno` set to `TLOOK` and we handle the error by calling `t_rcvuderr`. Notice that the final argument to `t_rcvuderr` is a null pointer to clear the error without returning any information (since there is nothing for us to do when these errors occur). If we did not handle these errors as shown, but aborted if `t_rcvudata` returned an error, then any client could crash our server by sending a datagram to the server and then immediately terminating. When the reply was received by the client's host, it would respond with an ICMP port unreachable, causing the server's `t_rcvudata` to return an error. Therefore the handling of these asynchronous errors is mandatory for a UDP server written using XTI.

31.6 Reading a Datagram in Pieces

Recall our discussion of datagram truncation in Section 20.3 and the different scenarios when a datagram is read on a UDP socket, but the datagram length exceeds the number of bytes requested by the application. XTI handles this scenario differently.

Recall the final *flagsp* argument for `t_rcvudata`. If the application's buffer is not large enough to hold the next datagram on the queue, the number of bytes returned will be `udata.maxlen` and the `T_MORE` bit in the integer pointed to by the *flagsp* argument will be turned on. This flag tells the application to call `t_rcvudata` again to read the remainder of this datagram. The sender's address and options are returned by only the first call to `t_rcvudata` for a given datagram. For all subsequent calls to this function that read the remainder of the datagram, `addr.len` and `opt.len` members will be 0 on return.

We can show the use of this feature by modifying our client from Figure 31.4, as shown in Figure 31.7.

Redefine MAXLINE

2-3 We redefine the constant `MAXLINE` from our `unp.h` header to be 2 bytes, and this is the size of our `recvline` buffer.

Create endpoint, send datagram to server

12-22 This code has not changed from Figure 31.4.

Read reply, 2 bytes at a time

23-41 We now call `t_rvcudata` in a loop, until our `flags` variable does not have the `T_MORE` bit set. We print the server's IP address for only the first piece of the datagram, when the `addr.len` member is nonzero.

xtiudp/daytimeudpcli4.c

```
 1 #include    "unpxti.h"

2 #undef MAXLINE
3 #define MAXLINE 2

4 int
5 main(int argc, char **argv)
6 {
 7     int     tfd, flags;
 8     char    recvline[MAXLINE + 1];
 9     sockler__t addrlen;
10     struct t_unitdata *sndptr, *rcvptr;
11     struct t_uderr *uderr;

12     it (argc != 3)
13        err_quit("usage: a.out <hostname or IPaddress> <service or port#>");

14     tfd = Udp_client(arav[1], argv[2], (void **) &sndptr, &addrlen);

15     rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
16     uderr = T_alloc(tfd, T_UDERROR, T_ADDR);

17     printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));

18     sndptr->udata.maxlen = MAXLINE;
19     sndptr->udata.len = 1;
20     sndptr->udata.buf = recvline;
21     recvline[0] = 0;             /* 1-byte datagram containing null byte */
22     T_sndudata(tfd, sndptr);

23     do {
24         rcvptr->udata.maxlen = MARLINE;
25         rcvptr->udata.buf = recvline;
26         flags = 0;
27         if (t_rcvudata(tfd, rcvptr, &flags) == 0) {
28             recvline[rcvptr->udata.len] = 0;     /* null terminate */
29             if (rcvptr->addr.len > 0)
30                 printf("from %s:     Xti_ntop_host(&rcvptr->addr));
31             printf("%s\n", recvline);
32         } else {
33             if (t_errno == TLOOK) {
34                 T_rcvuderr(tfd, uderr);
35                 printf("error %ld from %s\n",
36                        uderr->error, Xti_ntop_host(&uderr->addr));
37             } else
38                 err_xti("t_rcvudata error");
39             flags = 0;
40         }
41     } while (flags & T_MORE);
42     exit(0);
43 }
```

xtiudp/daytimeudpcli4.c

Figure 31.7 UDP client using XTI that reads returned datagram in pieces.

We now run this client to a daytime server.

```
unixware % daytimeudpcli4 bsdi daytime
sending to 206.62.226.35
from 206.62.226.35: Su
n
Ma
r
 2
 1
1:
53
:5
0
19
97
```

If we remove the newline from the `printf` format string for `recvline` (line 31, which we used only to show how much data was returned by `t_rcvudata),` we get the more familiar output:

```
unixware % daytimeudpcli4 bsdi daytime
sending to 206.62.226.35
from 206.62.226.35: Sun Mar 2 12:04:48 1997
```

31.7 Summary

The two XTI functions `t_rcvudata` and `t_sndudata` are similar to `recvfrom` and `sendto.` One new feature with XTI, which is not provided with sockets, is reading a datagram in pieces, having the `T_MORE` flag returned when there is more to read.

Asynchronous errors are returned with XTI by having `t_rcvudata` and `t_sndudata` return an error of `TLOOK.` We then call `t_rcvuderr` to obtain more protocol-dependent information about the error. This is better than the sockets approach (returning an asynchronous error only if the socket is connected), but even with the XTI approach asynchronous errors can be lost, and our application is still depending on the protocol stack to decide which ICMP errors to return. A better solution is to use a daemon like `icmpd` (Section 25.7) and return all the errors on a separate channel.

32

XTI Options

32.1 Introduction

Another of the mystery areas of XTI has been option processing. The standards and most manuals spend page after page describing the intricacies of option processing and option negotiation, providing no examples, and ending with a statement of the form "the details are protocol dependent."

The term *negotiation* is used heavily with XTI options. An option is not "set"; it is negotiated, meaning the provider may not set the option to exactly what we ask for. When an XTI option is negotiated, the actual value used by the provider is returned, so we can see what that value is.

Figure 32.1 shows all of the standard XTI options, both the generic options (those beginning with XTI_) and those for IPv4.

> Unix 98 prepended T_ to all the INET_, IP_, TCP_, and UDP_ names, but Posix.1g does not do this. For example, the UDP option is called UDP_CHECKSUM by Posix.lg. Unix 98 accepts these Posix.lg names as legacy names, but we will use the newer names in this text.

XTI classifies options as either *end-to-end* or *local*. End-to-end options normally cause some type of information to be sent to the peer across the network. An example is the IPv4 type-of-service field. It can be set by one endpoint (for either TCP or UDP), is carried in the IPv4 header, and is available to the other endpoint. The IPv4 header options and the UDP checksum are the two other end-to-end options in Figure 32.1. An example of a local option is T_IP_REUSEADDR, as this option affects the ability of the calling process to bind a port number that is already in use to its endpoint but has no effect on the data that is sent to the other endpoint.

Level	Name	Datatype	End-to-end	Absolute	Description
XTI_GENERIC	XTI_DEBUG	t_uscalar_t []		•	enable debug tracing
	XTI_LINGER	t_linger ()		•	linger on close if data to send
	XTI_RCVBUF	t_uscalar_t			receive buffer size
	XTI RCVLOWAT	t_uscalar_t			receive buffer low-water mark
	XTI_SNDBUF	t_uscalar_t			send buffer size
	XTI_SNDLOWAT	t_uscalar_t			send buffer low-water mark
T_INET_IP	T_IP_BROADCAST	u_int		•	permit sending of broadcast mesg
	T_IP_DONTROUTE	u_int		•	bypass routing table lookup
	T_IP_OPTIONS	u_char []	•	•	IP header options
	T_IP_REUSEADDR	u_int		•	allow local address reuse
	T_IP_TOS	u_char	•	•	type-of-service and precedence
	T_IP_TTL	u_char		•	time-to-live
T_INET_TCP	T_TCP_KEEPALIVE	t_kpalive {}		•	Periodically test if connection alive
	T_TCP_MAXSEG	t_uscalar_t			TCP MSS (read-only)
	T_TCP_NODELAY	t_uscalar_t		•	disable Nagle algorithm
T_INET_UDP	T_UDP_CHECKSUM	i t_uscalar_t	•	•	enable UDP checksum

Figure 32.1 XTI options.

Some XTI options are classified as an *absolute requirement*, which we also show in Figure 32.1. When setting the value of an option with this property, if the requested value cannot be assigned to the option, failure is returned. If an option does not have this property, and we try to set the option to some value that is not within the range of supported values, the provider will change the requested value to be within the acceptable range. An example of the latter is the receive buffer size, `XTI_RCVBUF`, as most systems enforce a lower limit and an upper limit on this value. If we request a value less than the lower limit or greater than the upper limit, the value will be changed to the appropriate limit and the return is then "partial success."

XTI options are specified and received in the following ways:

1. Calling the `t_optmgmt` function lets us specify any options (end-to-end and local) that we desire. We can also call this function to obtain the current value or the default value of an option.

t:k

2. For a UDP endpoint we can specify our desired options (end-to-end and local) with each call to `t_sndudata`, using the `opt` member of the `t_unitdata` structure.
3. For a UDP endpoint any end-to-end options that arrive with the datagram are returned by `t_rcvudata` through the `opt` member of the `t_unitdata` structure.
4. For a TCP client we can specify our desired options (end-to-end and local) when calling `t_connect`, as the `opt` member of the `t_call` structure.
5. For a TCP server any end-to-end options that arrive with the connection are returned by `t_listen` through the `opt` member of the `t_call` structure.

The t_optmgmt function is a combination of the getsockopt and setsockopt functions. In the sockets API, however, there is no way to specify options when sending or receiving UDP datagrams, or when initiating or accepting TCP connections. The sendmsg and recvmsg functions provide the capability to specify and receive ancillary data, and this is used for IPv6 options.

Figure 32.2 summarizes the sending and receiving of options by the XTI functions.

Endpoint	Function	Return end-to-end only	Return end-to-end and local	Specify end-to-end and local
any endpoint	t_optmgmt		•	•
TCP endpoint	t_accept			•
	t_connect		•	•
	t_listen	•		
	t_rcvconnect		•	
UDP endpoint	t_rcvudata	•		
	t_rcvvudata	•		
	t_rcvuderr		•	
	t_sndudata			•
	t_sndvudata			•

Figure 32.2 XTI functions that can specify and return options.

Figure 32.2 indicates that we can specify options with t_accept. With TCP this is not possible, because the connection is already established when t_listen returns. Hence, any desired end-to-end options that we want to effect the three-way handshake must be specified for the listening endpoint.

32.2 t_opthdr Structure

XTI options are always specified and returned through a netbuf structure named opt that is a member of the t_call, t_optmgmt, t_uderr, and t_unitdata structures (Figure 28.6). The contents of the options buffer is one or more t_opthdr structures, each followed by an optional value.

```
struct t_opthdr {
  t_uscalar_t len;      /* total length of option:
                           sizeof(struct t_opthdr) + length of value */
  t_uscalar_t level;  /* protocol affected */
  t_uscalar_t name;   /* option name */
  t_uscalar_t status; /* status value */
          /* followed by the option value, and then possible padding */
};
```

One difference from TLI to XTI is that TLI said nothing about the format of the options buffer other than it was implementation dependent. Many implementations of TLI used a structure named opthdr, which had only three elements: level, name, and len.

We show two of these XTI structures, pointed to by a netbuf structure that is part of a t_unitdata structure, in Figure 32.3.

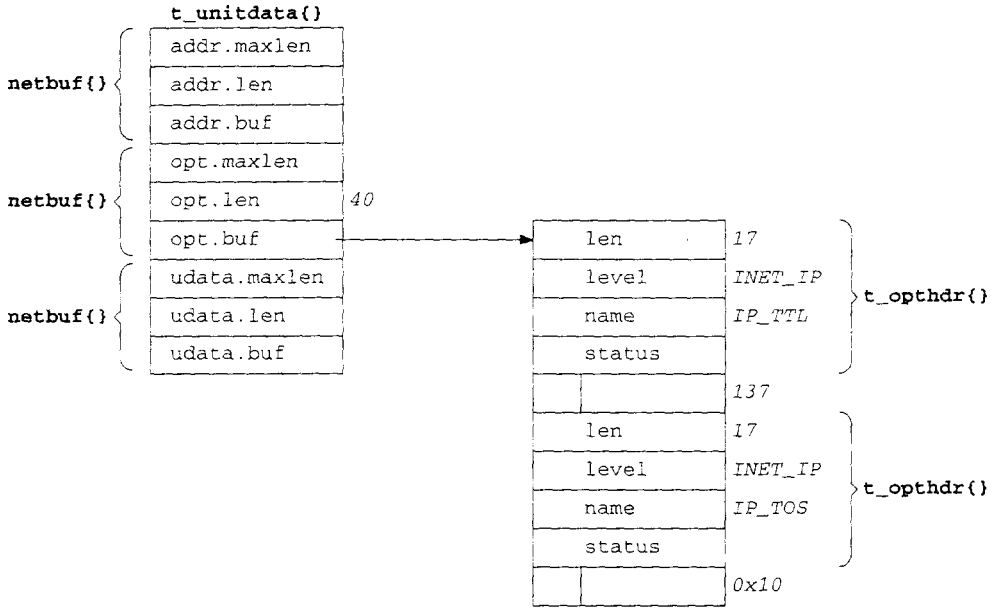

Figure 32.3 Example of two options pointed to by a `netbuf` structure.

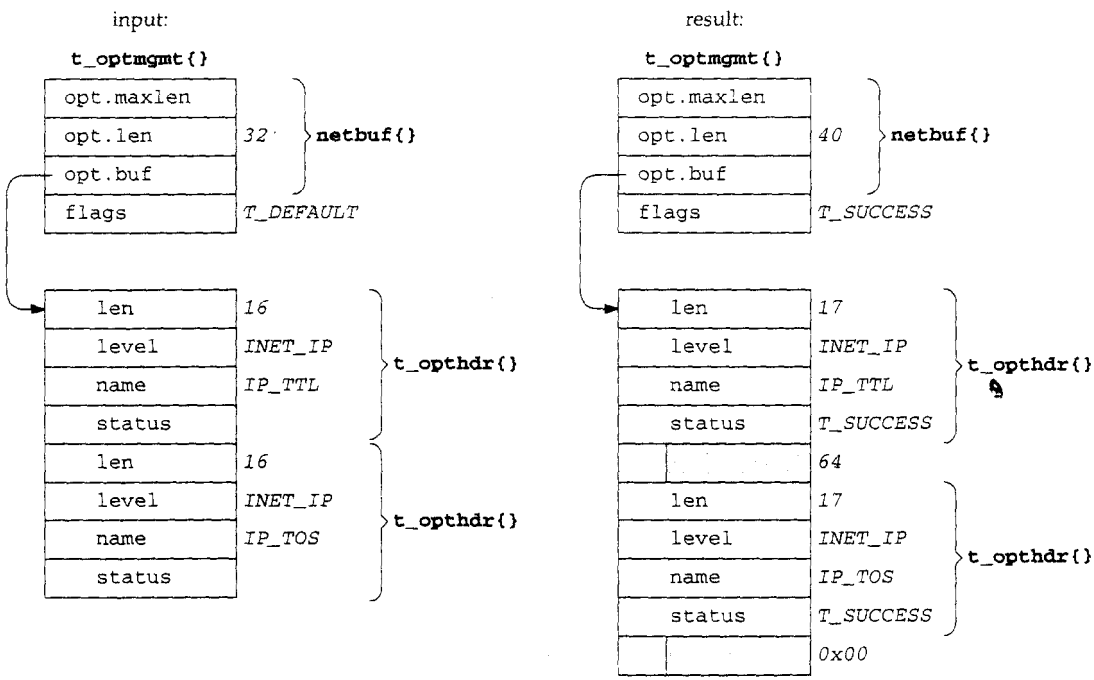

Figure 32.4 Requesting the default value of two options from `t_optmgmt`.

In Figure 32.3 we are specifying the IP TTL as 137 and the IP type-of-service as 0x10 (routine precedence and low delay). Each option value is a 1-byte `u_char` (Figure 32.1) and we show 3 bytes of padding after each value. We also assume here that a `t_uscalar_t` occupies 4 bytes; hence the total size of the option buffer is 40 bytes.

We show another example in Figure 32.4, this time a call to the `t_optmgmt` function, which we describe in Section 32.4. This function takes pointers to two `t_optmgmt` structures: one is the input and the other is the result.

In this example we are asking for the default values of the IP TTL and TOS options (the `flags` of `T_DEFAULT`) so we specify only a `t_opthdr` structure for each option without any value. The result is a copy of the input, with the default values returned after each of the `t_opthdr` structures. In the result structures the `status` member is also filled in.

> Unix 98, but not Posix.lg, defines three macros that can be used when processing a `t_opthdr` structure and the data that follows: `T_OPT_FIRSTHDR`, `T_OPT_NEXTHDR`, and `T_OPT_DATA`. These are similar to the three macros `CMSG_FIRSTHDR`, `CMSG_NXTHDR`, and `CMSG_DATA` used to process ancillary data with sockets, which we described in Section 13.6.

32.3 XTI Options

Most XTI options can be mapped directly to one of the socket options that we described in Chapter 7. Therefore our description of them here is brief. We also note that the header `<xti_inet.h>` must be included to define the constants for all the IP, TCP, and UDP options.

> Note that XTI does not define any way to multicast.

`XTI_DEBUG` Option

This option is similar to the `SO_DEBUG` socket option and normally supported by only TCP. This option is disabled by specifying an option header with no value. By this we mean that the `len` member of the `t_opthdr` structure is just the size of this structure (16 bytes, for example, in Figure 32.4).

`XTI_LINGER` Option

This option is similar to the `SO_LINGER` socket option and is supported by TCP. It specifies what to do when an endpoint is closed. The `t_linger` structure is

```
struct t_linger {
  t_scalar_t l_onoff;     /* T_NO, T_YES */
  t_scalar_t l_linger; /* T_UNSPEC (use default), T_INFINITE, or
                          linger time in seconds */
};
```

We specified in Figure 32.1 that this option is an absolute value, but only the value of

l_onoff is an absolute requirement; the value of l_linger is not an absolute requirement. That is, the implementation can place lower and upper limits on the linger time itself.

Unlike the SO_LINGER socket option, XTI_LINGER is not used to send an RST. t_snddis sends an RST.

XTI_RCVBUF and XTI_RCVLOWAT Options

These two options are similar to the SO_RCVBUF and SO_RCVLOWAT socket options. The first option specifies the size of the endpoint's receive buffer and the second the receive buffer low-water mark used with poll or select.

Figure 32.1 does not consider the XTI_RCVBUF option end-to-end, but with TCP's long fat pipe support (RFC 1323 [Jacobson, Braden, and Borman 19921), this option does indeed have end-to-end significance since it affects TCP's window scale option that is negotiated with the three-way handshake.

XTI_SNDBUF and XTI_SNDLOWAT Options

These two options are similar to the SO_SNDBUF and SO_SNDLOWAT socket options. The first option specifies the size of the endpoint's send buffer and the second the send buffer low-water mark that is used with poll or select.

T_IP_BROADCAST Option

This option is similar to the SO_BROADCAST socket option. The value of this option is either T_YES or T_NO.

T_IP_DONTROUTE Option

This option is similar to the SO_DONTROUTE socket option. The value of this option is either T_YES or T_NO.

T_IP_OPTIONS Option

This option is similar to the IP_OPTIONS socket option. The value of this option is used as the IPv4 header options. An example of these options is provided in Chap-ter 24. The option is disabled if specified with no value (i.e., only a t_opthdr structure).

Calling t_optmgmt with a request of T_CURRENT returns the current value of the IP options that will be used in outgoing datagrams.

T_IP_REUSEADDR Option

This option is similar to the SO_REUSEADDR socket option. The value of this option is either T_YES or T_NO.

T_IP_TOS Option

This option is similar to the IP_TOS socket option. The option value is a combination of the IPv4 precedence field, from the values shown in Figure 32.5, with the IPv4 typeof-service field, from the values shown in Figure 32.6.

Constant	Value
T_ROUTINE	0
T_PRICRITY	1
T_IMMEDIATE	2
T_FLASH	3
T_OVERRIDEFLASH	4
T_CRITIC_ECP	5
T_INETCONTROL	6
T NETCONTROL	7

Figure 32.5 IPv4 precedence values used with T_IP_TOS option.

Constant	Description
T_NOTOS	normal
T_LDELAY	minimize delay
T_HITHRPT	maximize throughput
T_HIREL	maximize reliability
T_LOCOST	minimize cost

Figure 32.6 IPv4 type-of-service values used with T_IP_TOS option.

The macro SET_TOS (defined by including the <xti.h> header) combines its first argument, a precedence value from Figure 32.5, with its second argument, a type-of-service value from Figure 32.6, and the result should be used with this XTI option.

Calling t_optmgmt with a request of T_CURRENT returns the current value of the option that will be used in outgoing datagrams.

T_IP_TTL Option

This option is similar to the IP_TTL socket option. The value of the option is the IPv4 time-to-live field. This option may be set to specify the value used in outgoing datagrams. There is no way, however, to obtain the 1"1'L from a received datagram.

T_TCP_KEEPALIVE Option

This option is similar to the SO_KEEPALIVE socket option and controls the sending of keepalive packets on a TCP connection. This XTI option uses the following structure:

```
struct t_kpalive {

   t_scalar_t  kp_onoff;   /* T_NO (disable), T_YES (enable), or
                              T YES|T GARBAGE (enable & send garbage byte) */
   t_scalar_t  kp_timeout; /* timeout in minutes; T_UNSPEC means default */
```

This option is similar to the XTI_LINGER option in that the value of kp_onoff is an absolute requirement but the value of kp_timeout is not an absolute requirement.

> Sending a garbage byte should not be required and in fact T_GARBAGE was removed from Unix 98. The use of the garbage byte is discussed on p. 335 of TCPv1.

T_TCP_MAXSEG Option

This option is similar to the TCP_MAXSEG socket option. This option is read-only and returns the maximum segment size (MSS) for a TCP endpoint. Since this option is read-only, its value cannot be an absolute requirement.

T_TCP_NODELAY Option

This option is similar to the TCP_NODELAY socket option. The value of this option is either T_YES (disable the Nagle algorithm) or T_NO (the default, the Nagle algorithm is enabled). We say more about the Nagle algorithm in Section 7.9.

T_UDP_CHECKSUM Option

This XTI option is one of the end-to-end options; therefore it is always returned by t_rcvudata if received options are requested (i.e., if opt.maxlen is nonzero). The value of this option is either T_YES or T_NO.

> This option should *never* be enabled and providing the ability for an application to disable the sending of UDP checksums for an endpoint is a mistake. Examples exist of data corruption when UDP checksums are disabled and there is no reason to ever disable UDP checksums. The only use of this option should be to detect whether a peer has UDP checksums enabled.

32.4 t_optmgmt Function

The t_optmgmt function lets us perform the following operations with regard to XTI options:

- check whether one or more options are supported,
- obtain the default value of one or more options,
- obtain the current value of one or more options, and
- negotiate values for one or more options.

```
#include <xti.h>

int t_optmgmt ( int fd, const struct t_optmgmt *request, struct t_optmgmt *result) ;
```

Returns: 0 if OK, -1 on error

We specify our request as a `t_optmgmt` structure, and one of these structures is returned as the result. If we are not interested in the result, we set the `maxlen` member of the structure pointed to by *result* to 0. We showed an example of these two structures in Figure 32.4.

```
struct t_optmgmt {
  struct netbuf opt;      /* one or more t_opthdr structures */
  t_scalar_t      flags; /* action on input, result on output */
};
```

The `flags` member of the *request* structure specifies the action desired by the caller:

`T_CHECK`	check whether the options are supported,
`T_DEFAULT`	return the default values of the options,
`T_CURRENT`	return the current values of the options, and
`T_NEGOTIATE`	negotiate values for the options.

We will examine each of these four operations in the following sections.

We are able to specify multiple options in a single call to `t_optmgmt` as shown in Figure 32.4. But if we do this all options must specify the same `level`. This is OK in Figure 32.4 because the `level` of both options is `T_INET_IP`. There is another complication when negotiating new values for multiple options in a single call to this function: the returned `flags` contains the worst single result, even though each option contains its own `status` return. To avoid these complications, it is simplest to manipulate just one option at a time in each call to `t_optmgmt`.

This XTI function corresponds to the `getsockopt` and `setsockopt` functions.

32.5 Checking If an Option Is Supported and Obtaining the Default

Our first example of XTI options is to check which of the options listed in Figure 32.1 are supported on our system, and for each supported option, to print its default value. Figure 32.7 shows our program.

xtiopt/checkopts.c

```
 1 #include      "unpxti.h"
 2 struct xti_opts {
 3   char             *opt_str;
 4   t_uscalar_t       opt_level;
 5   t_uscalar_t       opt_name;
 6   char           *(*opt_val_str)(struct t_opthdr *);
 7 } xti_opts[]   = {
 8     "XTI_DEBUG",           XTI_GENERIC,      XTI_DEBUG,        xti_str_uscalard,
 9     "XTI_LINGER",          XTI_GENERIC,      XTI_LINGER,       xti_str_linger,
10     "XTI_RCVBUF",          XTI_GENERIC,      XTI_RCVBUF,       xti_str_uscalard,
11     "XTI_RCVLOWAT",        XTI_GENERIC,      XTI_RCVLOWAT,     xti_str_uscalard,
12     "XTI_SNDBUF",          XTI_GENERIC,      XTI_SNDBUF,       xti_str_uscalard,
13     "XTI_SNDLOWAT",        XTI_GENERIC,      XTI_SNDLOWAT,     xti_str_uscalard,
14     "T_IP_BROADCAST",      T_INET_IP,        T_IP_BROADCAST  xti_str_uiyn,
15     "T_IP_DONTROUTE",      T_INET_IP,        T_IP_DONTROUTE,   xti_str_uiyn,
16     "T_IP_OPTIONS",        T_INET_IP,        T_IP_OPTIONS,     xti_str_uchard,
```

```
17      "T_IP_REUSEADDR",       T_INET_IP,          T_IP_REUSEADDR,      xti_str_uiyn,
18      "T_IP_TOS",             T_INET_IP,          T_IP_TOS,            xti_str_ucharx,
19      "T_IP_TTL" ,            T_INET_IP,          T_IP_TTL,            xti_str_uchard,
20      "T_TCP_KEEPALIVE",      T_INET_TCP,         T_TCP_KEEPALIVE      Xti_str_kpalive,
21      "T_TCP_MAXSEG",         T_INET_TCP,         T_TCP_MAXSEG,        xti_str_uscalard,
22      "T_TCP_NODELAY",        T_INET_TCP,         T_TCP_NODELAY,       xti_str_usyn,
23      "T_UDP_CHECKSUM",       T_INET_UDP,         T_UDP_CHECKSUM,      xti_str_usyn,
24      NULL,                   0,                  0,                   NULL
25 };
26 int
27 main (int argc, char **argv)
28 {
29      int    fd;
30      struct t_opthdr *topt;
31      struct t_ooptmgmt *reg, *ret;
32      struct xti:opts *ptr;
33      if (argc != 2)
34         err_quit("usage: checkopts <device>");
35      fd = T_open(argv[1], O_RDWR, NULL);
36      T_bind(fd, NULL, NULL);
37      req = T_alloc(fd, T_OPTMGMT, T_ALL);
38      ret = T_alloc(fd, T_OPTMGMT, T_ALL);
39      for (ptr = xti_opts; ptr->opt_str != NULL; ptr++) {
40          topt = (struct t_opthdr *) req->opt.buf;
41          topt->level = ptr->opt_1evel;
42          topt->name = ptr->opt_name;
43          topt->len = sizeof(struct t_opthdr);
44          req->opt.len = topt->len;
45          req->flags = T_CHECK;
46          printf("%s: ", ptr->opt_str);
47          if (t_optmgmt(fd, req, ret) < 0) {
48              err_xti_ret("t_optmgmt error");
49          } else {
50              topt = (struct t_opthdr *) ret->opt.buf;
51              printf("%s", xti_str_flags(topt->status));
52              if (topt->status == T_SUCCESS || topt->status == T_READONLY) {
53                  req->flags = T_DEFAULT;
54                  if ( t_optmgmt ( fd, req, ret) < 0) {
55                      err_xti_ret("t_optmgmt error for T_DEFAULT"); %
56                  } else {
57                      topt = (struct t_opthdr *) ret->opt.buf;
58                      printf(", default = %s", (*ptr->opt_val_str) (topt));
59 }
60              }
61              printf("\n");
62 }
63      }
64      exit(0);
65 }
```

xtiopt/checkopts.c

Figure 32.7 Check which XTI options are supported.

2-25 We define and initialize a structure defining all the XTI options from Figure 32.1. The final member of each array element is a pointer to a function that prints the value of the option. We need one function for each of the different option types. We do not show the source code for all these functions here.

Open device

35-38 We take the device name as a command-line argument and open the device. This lets us run the program twice, once for `/dev/tcp` and once for `/dev/udp`, as we expect different options to be supported by each provider. We bind any local address to the endpoint, because most calls to `t_optmgmt` require that the endpoint be bound. We also allocate two `t_optmgmt` structures, one for our request and one for the function's reply.

Call `t_optmgmt` for request of `T_CHECK`

39-48 We call `t_optmgmt` for each option in our `xti_opts` array, with a request flag of `T_CHECK`. We fill in our `req` structure, building a single `t_opthdr` structure in the `opt` buffer (Section 32.2). This structure contains just a `t_opthdr` structure, without any data (e.g., similar to the left side of Figure 32.4).

Call `t_optmgmt` for request of `T_DEFAULT`

49-62 If the first call to `t_optmgmt` succeeds, we print the `status` of the option. If the `status` is `T_SUCCESS` or `T_READONLY`, we call `t_optmgmt` again, this time with a request of `T_DEFAULT`. If this second call succeeds, we call the function pointed to by the `opt_val_str` member of our structure in Figure 32.7 to print the default value. When we call `t_optmgmt` the second time, we change only the `flags` member of our request structure. Since the pointer to this structure in the function prototype for `t_optmgmt` has the `const` qualifier, we know the structure was not changed by the first call.

We now run the program two times under AIX 4.2: first for TCP and then for UDP. Notice that AIX uses the device names `/dev/xti/tcp` and `/dev/xti/udp`.

```
aix % checkopts /dev/xti/tcp
XTI_DEBUG: T_SUCCESS, default = 0
XTI_LINGER: T_SUCCESS, default = T_NO, 0 sec
XTI_RCVBUF: T_SUCCESS, default = 16384
XTI_RCVLOWAT: T_SUCCESS, default = 1
XTI_SNDBUF: T_SUCCESS, default = 16384
XTI_SNDLOWAT: T_SUCCESS, default = 4096
T_IP_BROADCAST: T_SUCCESS, default = T_NO
T_IP_DONTROUTE: T_SUCCESS, default = T_NO
T_IP_OPTIONS: T_SUCCESS, default = 0 (length of value)
T_IP_REUSEADDR: T_SUCCESS, default = T_NO
T_IP_TOS: T_SUCCESS, default = 0x00
T_IP_TTL: T_SUCCESS, default = 0
T_TCP_KEEPALIVE: T_SUCCESS, default = T_NO T_UNSPEC
T_TCP_MAXSEG: T_READONLY, default = 512
T_TCP_NODELAY: T_SUCCESS, default = T_NO
T_UDP_CHECKSUM: t_optmgmt error: incorrect option format
```

```
aix % checkopts /dev/xti/udp
XTI_DEBUG: T_SUCCESS, default = 0
XTI_LINGER: T_SUCCESS, default = T_NO, 0 sec
XTI_RCVBUF: T_SUCCESS, default = 41600
XTI_RCVLOWAT: T_SUCCESS, default = 1
XTI_SNDBUF: T_SUCCESS, default = 9216
XTI_SNDLOWAT: T_SUCCESS, default = 4096
T_IP_BROADCAST: T_SUCCESS, default = T_NO
T_IP_DONTROUTE: T_SUCCESS, default = T_NO
T_IP_OPTIONS: T_SUCCESS, default = 0 (length of value)
T_IP_REUSEADDR: T_SUCCESS, default = T_NO
T_IP_TOS: T_SUCCESS, default = 0x00
T_IP_TTL: T_SUCCESS, default = 0
T_TCP_KEEPALIVE: t_optmgmt error: incorrect option format
T_TCP._MAXSEG: t_optmgmt error: incorrect option format
T_TCP_NODELAY: t_optmgmt error: incorrect option format
T_UDP_CHECKSUM: T_NOTSUPPORT
```

The supported values are as we expect, other than the `T_IP_TTL` value. The `T_UDP_CHECKSUM` option that is not supported by TCP, and the three TCP options that are not supported by UDP cause `t_optmgmt` to return an error of `TBADOPT.` The UDP provider understands the `T_UDP_CHECKSUM` option but returns `T_NOTSUPPORT` since it is not supported. The string "(length of value)" that is printed for `T_IP_OPTIONS` indicates that the `len` member that was returned was 0, so there is no value to output.

32.6 Getting and Setting XTI Options

We now show examples of getting and setting XTI options. We define two functions of our own, `xti_getopt` and `xti_setopt,` that have identical calling sequences to `getsockopt` and `setsockopt` (Section 7.2).

```
#include "unpxti.h"

int xti_getopt (int fd, int level, int name, void *optval, socklen_t *optlen);

int xti_setopt(int fd, int level, int name, const void *optval, socklen_t *optlen); i
```

Both return: 0 if OK, on error

These functions can simplify our XTI programs, since each comprises about 20—30 lines of C code.

`xti_getopt` Function

To fetch the current value of an XTI option we call `t_optmgmt` with the `flags` member of the request structure set to `T_CURRENT.` Figure 32.8 shows our `xti_getopt` function.

```
 1 ‘include      "unpxti.h"                                    libxti/xti_getopt.c

 2 int.
 3 xti_getopt(int fd, int level, int name, void *optvai, socklen_t *optlenp)
 4 {
 5      int     rc, len;
 6      struct t_optmgmt_*req, *ref.;
 7      struct t_opthdr *tope;

 8      req = T_alloc(fd, T_OPTMGMT, T_ALL);
 9      ret_ = T_alloc(fd, T_OPTMGMT, T_ALL);
10      if (req->opt.maxlen == 0)
11          err_quit("xti_getopt: opt.maxlen == 0");

12      topt = (struct t_opthdr *) req->opt.buf;
13      topt->level = level;
14      topt->name = name;
15      topf->len = sizeof(struct t_opthdr);     /* just a t_opthdr{} */
16      req->opt.len = topt->len;

17      req->flags = T_CURRENT;
18      if (t_optmgmt(fd, req, ret) < 0) {
19          T_free(req, T_OPTMGMT);
20          T_free(ret, T_OPTMGMT);
21          return (-1);
22      }
23      rc = ret->flags;

24      if (rc == T_SUCCESS || rc == T_READONLY) {
25              /* copy back value and length */
26          topt = (struct t_opthdr *) ret->opt.buf;
27          len = topt->len - sizeof(struct t_opthdr);
28          len = min(len, *optlenp);
29          memcpy(optival, topt + 1, len);
30          *optlenp = len;
31      }
32      T_free(req, T_OPTMGMT);
33      T_free(ret, T_OPTMGMT);

34      if (rc == T_SUCCESS || rc == T_READONLY)
35          return (0);
36      return (-1);                /* T_NOTSUPPORT */
37 }

                                                              libxti/xti_getopt.c
```

Allocate request and reply structures

3-11 We call `t_alloc` to allocate room for a request structure and a reply structure. We also verify that the size of the options buffer is nonzero.

> Older implementations of TLI often used a value of 0 for the size of the TCP options, meaning the application had to allocate its own buffer.

Fill in `t_opthdr` structure

12-16 We fill in a `t_opthdr` structure with the option's *level* and *name*. We do not specify any value in the request structure because this is not required when fetching the current value of an option.

Call `t_optmgmt` and return option value

17-31 We call `t_optmgmt` and then save the `flags` member in the returned structure. If the return value was `T_SUCCESS` or `T_READONLY`, we copy back the value of the option and its size. (The pointer expression `topt+1` points to the returned option value, which immediately follows the `t_opthdr` structure.) The final argument to our function is a value–result argument and we are careful not to overflow the caller's buffer (in case it is too small).

Free memory and return

32-36 We free the memory allocated by `t_alloc` and return 0 on success or -1 on an error.

`xti_setopt` Function

To set the value of an XTI option we call `t_optmgmt` with the `flags` member of the request structure set to `T_NEGOTIATE`. Figure 32.9 shows our `xti_setopt` function. This function is similar to the `xti_getopt` function in Figure 32.8 with a few exceptions.

Copy caller's value

12-19 We copy the caller's option value into the buffer that we build, placing it immediately following the `t_opthdr` structure.

Call t_opmgmt

20-26 The request flags is now `T_NEGOTIATE` for `t_optmgmt`.

Free memory and return

27-31 If the option value is not an absolute requirement the return value is `T_PARTSUCCESS`, which is OK.

XTI lets us set an option and fetch its value in a single call to `t_optmgmt`. This might be useful for an option whose value is not an absolute requirement (e.g., the send and receive buffer sizes). Using our functions requires a call to `xti_setopt` followed by a call to `xti_getopt`. We could have defined a function that does both, but the extra call to `xti_getopt` would rarely be the bottleneck of an application.

libxti/xti_setopt.c

```
 1 #include    "unpxti.h"

 2 int
 3 xti_setopt(int fd, int level, int name, void *optval, socklen_t optlen)
 4 {
 5     int     rc;
 6     struct t_optmgmt *req, *ret;
 7     struct t_opthdr *topt;

 8     req = T_alloc(fd, T_OPTMGMT, T_ALL);
 9     ret = T_alloc(fd, T_OPTMGMT, T_ALL);
10     if (req->opt.maxlen == 0)
11         err_quit("xti_setopt: req.opt.maxlen == 0");

12     topt = (struct t_opthdr *) req->opt.buf;
13     topt->level = level;
14     topt->name = name;
15     topt->len = sizeof(struct t_opthdr) + optlen;
16     if (topt->len > req->opt.maxlen)
17         err_quit("optlen too big");
18     req->opt.len = topt->len;
19     memcpy(topt + 1, optval, optlen);    /* copy option value */

20     req->flags = T_NEGOTIATE;
21     if (t_optmgmt(fd, req, ret) < 0) {
22         T_free(req, T_OPTMGMT);
23         T_free(ret, T_OPTMGMT);
24         return (-1);
25     }
26     rc = ret->flags;

27     T_free(req, T_OPTMGMT);
28     T_free(ret, T_OPTMGMT);

29     if (rc == T_SUCCESS || rc == T_PARTSUCCESS)
30         return (0);
31     return (-1);               /* T_FAILURE, T_NOTSUPPORT, T_READONLY */
32 }
```

Figure 32.9 xti_setopt function: set the value of an XTI option.

libxti/xti_setopt.c

Example

We now use the two functions that were just shown. The program in Figure 32.10 fetches the current value of TCP's maximum segment size, sets the size of the send buff-er to 65536, and then fetches and prints the size of the send buffer. If we compile and execute this program, its output is

```
aix % getsetopt
TCP mss = 512
send buffer size = 65536
```

xtiopt/getsetopt.c

```
 1 #include    "unpxti.h"
```

```
1 #include    “unpxti.h”

2 int
3 main(int argc, char **argv)
4 {
5       int     fd;
6       socklen_t optlen;
7       t_uscalar_t mss, sendbuff;

 3      fd = T_open(XTI_TCP, O_RDWR, NULL);
 9      T_bind(fd, NULL, NULL);

10      optlen = sizeof(mss);
11      Xti_getopt(fd, T_INET_TCP, T_TCP_MAXSEG, &mss, &optlen);
12      printf("TCP mss = %d\n", mss);

13      sendbuff = 65536;
14      Xti_setopt(fd, XTI_GENERIC, XTI_SNDBUF, &sendbuff, sizeof(sendbuff));

15      optlen = sizeof(sendbuff);
16      Xti_getopt(fd, XTI_GENERIC, XTI_SNDBUF, &sendbuff, &optlen);
17      printf("send buffer size = %d\n", sendbuff);

18      exit(0);
19 }
```

xtiopt/getsetopt.c

Figure 32.10 Example of our xti_getopt and xti_getopt functions.

321 Summary

XTI options are negotiated with the possibility that the provider returns a different value than we asked for. Although XTI option processing is very general, the simplest approach is to define two basic functions that look like getsockopt and setsockopt and call them from our application.

33

Streams

33.1 Introduction

Before describing some of the additional features of XTI, such as signal-driven I/O and out-of-band data, we need to understand some implementation details. XTI and the networking protocols are normally implemented using the streams system, as is the terminal I/O system on most SVR4-derived kernels.

In this chapter we provide an overview of the streams system and the functions used by an application to access a stream. Our goal is to understand the implementation of networking protocols within the streams framework. We also develop a simple TCP client using TPI, the interface into the transport layer that both XTI and sockets normally use on a system based on streams. Additional information on streams, including information on writing kernel routines that utilize streams, can be found in [Rago 1993].

Streams were designed by Dennis Ritchie [Ritchie 1984] and first made widely available with SVR3 in 1986. They have never been standardized by Posix. The basic streams functions are required by Unix 98: `getmsg, getpmsg, putmsg, putpmsg, fattach,` and all of the streams `ioctl` commands. XTI is often implemented using streams. Any system derived from System V should provide streams, but the various 4.xBSD releases do not provide streams.

The streams system is often written as STREAMS, but it is not even an acronym, so we write it as just *streams.*

Be careful to distinguish between the stream I/O system that we are describing in this chapter, versus "standard I/O streams." The latter term is used *when* talking about the standard I/O library (e.g., functions such as `fopen, (gets, print f,` and the like).

33.2 Overview

Streams provide a full-duplex connection between a process and *a driver*, as shown in Figure 33.1. Although we describe the bottom box as a driver, this need not be associated with a hardware device; it can also be a pseudo-device driver (e.g., a software driver).

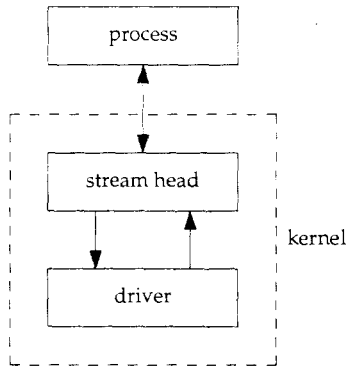

Figure 33.1 A stream shown between a process and a driver.

The *stream head* consists of the kernel routines that are invoked when the application makes a system call for a streams descriptor (e.g., read, putmsg, ioctl, and the like).

A process can dynamically add and remove intermediate processing *modules* between the stream head and the driver. A module performs some type of filtering on the messages going up and down a stream. We show this in Figure 33.2.

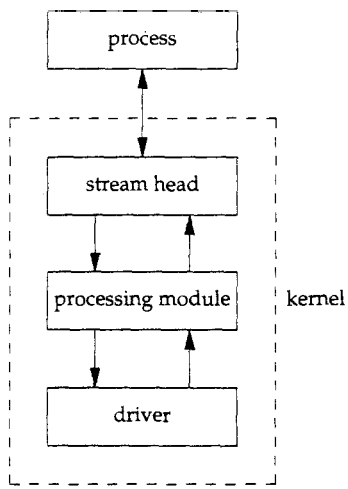

Figure 33.2 A stream with a processing module.

Any number of modules can be pushed onto a stream. When we *say push*, we mean that each new module gets inserted just below the stream head.

A special type of pseudo-device driver is a *multiplexor*, which accepts data from multiple sources. A streams-based implementation of the TCP/IP protocol suite, as found on SVR4 for example, could be as shown in Figure 33.3.

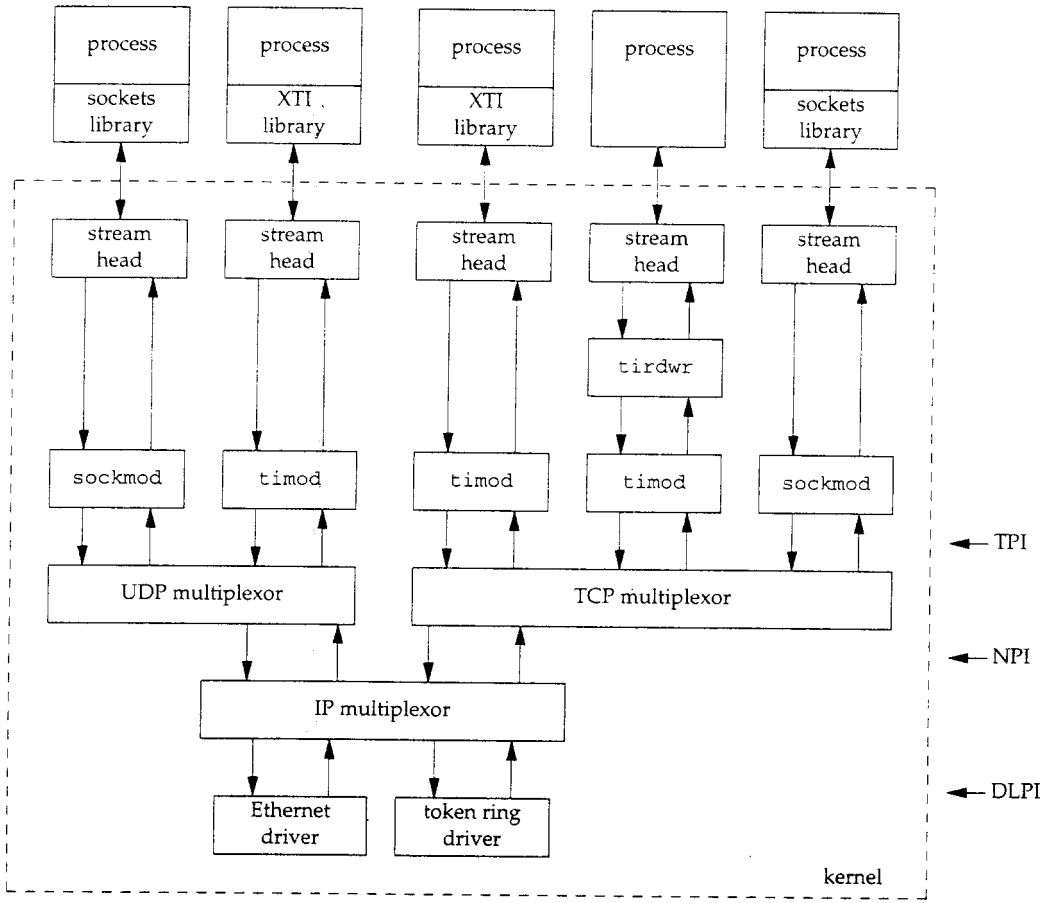

Figure 33.3 Simplified streams implementations of TCP/IP using streams.

- When a socket is created, the module `sockmod` is pushed onto the stream by the sockets library. It is the combination of the sockets library and the `sockmod` streams module that provides the sockets API to the process.
- When an XTI endpoint is created, the module `timod` is pushed onto the stream by the XTI library. It is the combination of the XTI library and the `timod` streams module that provides the XTI API to the process.
- We mentioned in Section 28.12 that the streams module `tirdwr` must normally be pushed onto a stream to use `read` and `write` with an XTI endpoint. The middle process using TCP in Figure 33.3 has done this. This process has probably abandoned the use of XTI by doing this, so we have removed the XTI library.

• Three service interfaces define the format of the networking messages exchanged up and down a stream. TPI, the *Transport Provider Interface* [Unix International 1992b], defines the interface provided by a transport-layer provider (e.g., TCP and UDP) to the modules above it. NPI, the *Network Provider Interface* [Unix International 1992a], defines the interface provided by a network-layer provider (e.g., IP). DLPI is the *Data Link Provider Interface* [Unix International 1991]. An alternate reference for TPI and DLPI, which contains sample C code, is [Rago 1993].

> The claim is regularly made to Usenet that "in a streams environment sockets are implemented on top of TLI (XTI)." This is false. As we can see in Figure 33.3, both sockets and XTI are implemented on top of TPI. This claim is often followed with "therefore TLI (XTI) is faster than sockets." This is also false. The TCP, UDP, and IP layers are the same, regardless of whether XTI or sockets are used. What changes is the user library and whether cimod or sockmod is on the stream. But the author is not aware of any numbers comparing these libraries and modules, For the bottleneck of most applications (data transfer), the code path is probably similar for XTI and sockets, unless special optimizations have been applied to one and not the other.

Each component in a stream—the stream head, all processing modules, and the driver—contain at least one pair of *queues:* a write queue and a read queue. We show this in Figure 33.4.

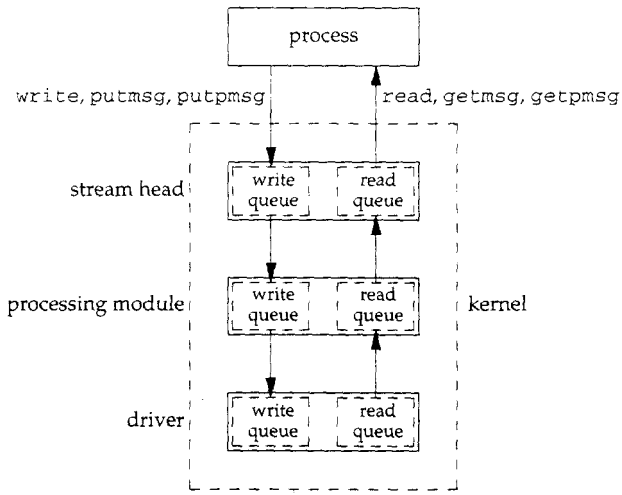

Figure 33.4 Each component in a stream has at least one pair of queues.

Message Types

Streams messages can be categorized as *high priority*, *priority band*, or *normal*. There are 256 different priority bands, between 0 and 255, with normal messages in band 0. The priority of a streams message is used for both queueing and flow control. By convention, high-priority messages are unaffected by flow control.

Figure 33.5 shows the ordering of the messages on a given queue.

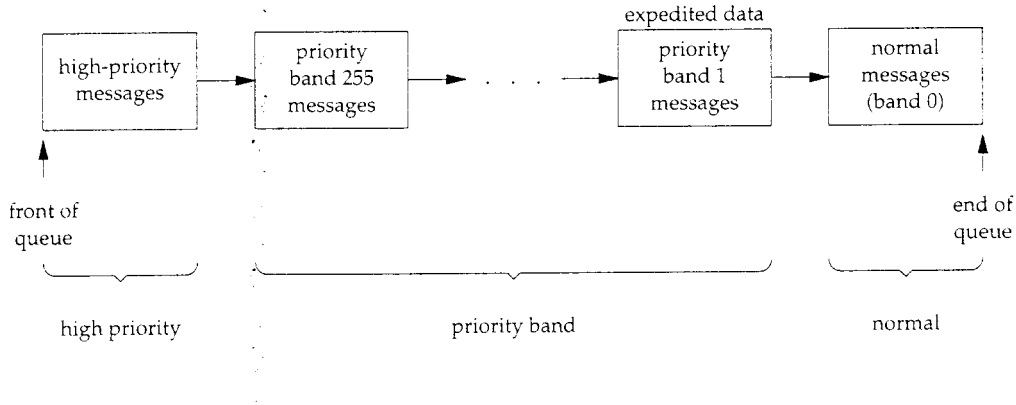

Figure 33.5- Ordering of streams messages on a queue, based on priority.

Although the streams system supports 256 different priority bands, networking protocols often use band 1 for expedited data and band 0 for normal data.

TCP's out-of-band data is not considered true expedited data by TPI. Indeed, TCP uses band 0 for both normal data and its out-of-band data (as we will verify in Figure C.4). The use of band 1 for expedited data is for protocols in which the expedited data (not just the urgent pointer, as in TCP) is sent ahead of normal data.

Beware of the.term *normal.* In releases before SVR4 there were no priority bands; there were just normal messages and priority messages. SVR4 implemented priority bands, requiring the `getpmsg` and `putpmsg` functions, which we describe shortly. The older priority messages were renamed, high priority. The question is what to call the new messages, with priority bands between 1 and 255. Common terminology [Rago 1993] is to refer to everything other than high-priority messages as normal-priority messages and then subdivide these normal-priority messages into priority bands. The term *normal message* should always refer to a message with a band of O.

Although we talk about normal-priority messages and high-priority messages, there are about a dozen normal-priority message types and around 18 high-priority message types. From an application's perspective, and the `getmsg` and `putmsg` functions that we are about to describe, we are interested in only three different types of messages: `M_DATA`, `M_PROTO`, and `M_PCPROTO` (`PC` stands for "priority control" and implies a high-priority message). Figure 33.6 shows how these three different message types are generated by the `write` and `putmsg` functions.

Function	Control?	Data?	Flags	Message type generated
`write`		yes		`M_DATA`
`putmsg`	no	yes	0	`M_DATA`
`putmsg`	yes	don't care	0	`M_PROTO`
`putmsg`	yes	don't care	`MSG_HIPRI`	`M_PCPROTO`

Figure 33.6 Streams **message types** generated by `write` and `putmsg`.

We will see what we mean by control, data, and flags in our description of the putmsg function in the next section.

33.3 getmsg and putmsg Functions

The data transferred up and down a stream consists of messages and each message contains *control, data*, or both. If we use read and write on a stream, these transfer only data. To allow a process to read and write both data and control information, two new functions were added.

```
#include <stropts.h>

int getmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);

int putmsg (int fd, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int flags);
```

Both return: nonnegative value if OK (see text), -1 on error

Both the control and data portions of the message are described by a strbuf structure:

```
struct strbuf {
  int    maxlen;     /* maximum size of buf */
  int     len;       /* actual amount of data in buf */
  char   *buf;       /* data */
```

Note the similarity between the strbuf structure and the netbuf structure. The names of the three elements in each structure are identical.

But the two lengths in the netbuf structure are unsigned integers, while the two lengths in the strbuf structure are signed integers. The reason is that some of the streams functions use a len or maxlen value of—1 to indicate something special.

We can send only control information, only data, or both using putmsg. To indicate the absence of control information we can either specify *ctlptr* as a null pointer, or set *ctlptr->len* to -1. The same technique is used to indicate no data.

If there is no control information, an M_DATA message is generated by putmsg (Figure 33.6); otherwise either an M_PROTO or an M_PCPROTO message is generated, depending on the *flags*. The *flags* argument to putmsg is 0 for a normal message or RS_HIPRI for a high-priority message.

The final argument to getmsg is a value—result argument. If the integer pointed to by *flagsp is* 0 when the function is called, the first message on the stream is returned (which can be normal or high priority). If the integer value is RS_HIPRI when the function is called, the function waits for a high-priority message to arrive at the stream head. In both cases the value stored in the integer pointed to by *flagsp* will be 0 or RS_HIPRI, depending on the type of message returned.

Assuming we pass nonnull *ctlptr* and *dataptr* values to getmsg, if there is no control information to return (i.e., an M_DATA message is being returned), this is indicated by

setting *ctlptr->len* to -1 on return. Similarly, *dataptr->len is* set to -1 if there is no data to return.

The return value from putmsg is 0 if all is OK, or -1 on an error. But getmsg returns 0 only if the entire message was returned to the caller. If the control buffer is too small for all the control information, the return value is MORECTL (which is guaranteed to be nonnegative). Similarly if the data buffer is too small, MOREDATA can be returned. If both are too small, the logical OR of these two flags is returned.

33.4 getpmsg and putpmsg Functions

When support for different priority bands was added to streams with SVR4, the following two variants of getmsg and putmsg were added.

```
#include <stropts.h>

int getpmsg ( int fd, struct strbuf *ctlptr,
              struct strbuf *dataptr, int *bandp, int *flagsp) ;

int putpmsg(int fd, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags) ;
```

Both return: nonnegative value if OK, -1 on error

The *band* argument to putpmsg must be between 0 and 255, inclusive. If the *flags* argument is MSG_BAND, then a message is generated in the specified priority band. Setting *flags* to MSG_BAND and specifying a band of 0 is equivalent to calling putmsg. *If flags is* MSG_HIPRI, *band* must be 0, and a high-priority message is generated. (Note that this flag is named differently from the RS_HIPRI flag for putmsg.)

The two integers pointed to by *bandp* and *flags p* are value-result arguments for getpmsg. The integer pointed to by *flagsp* for getpmsg can be MSG_HIPRI (to read a high-priority message), MSG_BAND (to read a message whose priority band is at least equal to the integer pointed to by *bandp)*, or MSG_ANY (to read any message). On return the integer pointed to by *bandp* contains the band of the message that was read, and the integer pointed to by *flagsp* contains MSG_HIPRI (if a high-priority message was read) or MSG_BAND (if some other message was read).

33.5 ioctl Function

With streams we again encounter the ioctl function that we described in Chapter 16.

```
#include <stropts.h>

int ioctl(int fd, int request, ... /* void *arg */ );
```

Returns: 0 if OK, -1 on error

The only change from the function prototype shown in Section 16.2 is the headers that must be included when dealing with streams.

There are about 30 requests that affect a stream head. Each request begins with `I_` and they are normally documented on the `streamio` manual page. We showed the `I_PUSH` request in Figure 28.14 when we pushed the `tirdwr` module onto a stream.

When we discuss signal-driven I/O with XTI in Section 34.11 we discuss the `I_SETSIG` request.

33.6 TPI: Transport Provider Interface

In Figure 33.3 we showed that TPI is the service interface into the transport layer from above. Both sockets and XTI use this interface in a streams environment. In Figure 33.3 it is a combination of the sockets library and `sockmod`, along with a combination of the XTI library and `timod` that exchange TPI messages with TCP and UDP.

TPI is *a message-based* interface. It defines the messages that are exchanged up and down a stream between the application (e.g., the XTI or sockets library) and the trans-port layer: the format of these messages and what operation each message performs. In many instances the application sends a request to the provider (such as "bind this local address") and the provider sends back a response ("OK" or "error"). Some events occur asynchronously at the provider (the arrival of a connection request for a server), causing a message or a signal to be sent up the stream.

We are able to bypass both XTI and sockets and use TPI directly. In this section we rewrite our simple daytime client using TPI, instead of sockets (Figure 1.5) or XTI (Figure 28.13). Using programming languages as an analogy, using sockets or XTI is like programming in a high-level language such as C or Pascal, while using TPI directly is like programming in assembler. We are not advocating the use of TPI directly in real applications. But examining how TPI works and developing this example gives us a better understanding of how the sockets library and the XTI library work in a streams environment.

Figure 33.7 is our `tpi_daytime.h` header.

streams/tpi_daytime.h

```
1 #include     "unpxti.h"
2 #include     <sys/stream.h>
3 #include     <sys/tihdr.h>

4 void      tpi_bind(int, const void *, size_t);
5 void      tpi_connect(int, const void *, size_t);
6 ssize_t   tpi_read(int, void *, size_t);
7 void      tpi_close(int);
```

streams/tpi_daytime.h

Figure 33.7 Our tpi_daytime.h header.

We need to include one additional streams header along with `<sys/tihdr.h>`, which contains the definitions of the structures for all the TPI messages.

Figure 33.8 is the `main` function for our daytime client.

streams/tpi_daytime.c

```
1 #include     "tpi_daytime.h"
2 int
3 main(int argc, char **argv)
4 {
5       int     fd, n;
6       char    recvline[MAXLINE + 1];
7       struct sockaddr_in myaddr, servaddr;
8       if (argc != 2)   .
9           err_quit("usage: tpi_daytime <IPaddress>");
10      fd = Open(XTI_TCP, 0_RDWR, 0);
11          /* bind any local address */
12      bzero(&myaddr, sizeof(myaddr));
13      myaddr.sin_family = AF_INET;
14      myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15      myaddr.sin_port = htons(0);
16      tpi_bind(fd, &myaddr, sizeof(struct sockaddr_in));
17          /* fill in server's address */
18      bzero(&servaddr, sizeof(servaddr));
19      servaddr.sin_family = AF_INET;
20      servaddr.sin_port = htons(13); /* daytime server */
21      Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
22      tpi_connect(fd, &servaddr, sizeof(struct sockaddr_in));
23      for ( ; ; ) {
24          if ( (n = tpi_read(fd, recvline, MAXLINE)) <= 0) {
25              if (n == 0)
26                  break;
27              else
28                  err_sys("tpi_read error");
29          }
30          recvline[n] = 0;        /* null terminate */
31          fputs(recvline, stdout);
32      }
33      tpi_close(fd);
34      exit(0);
35      }
```

streams/tpi_daytime.c

Figure 33.8 main function for our daytime client written to TPI.

Open transport provider, bind local address

10-16 We open the device corresponding to the transport provider (normally `/dev/tcp`). We fill in an Internet socket address structure with `INADDR_ANY` and a port of 0, telling TCP to bind any local address to our endpoint. We call our own function `tpi_bind` (shown shortly) to do the bind.

Fill in server's address, establish connection

17-22 We fill in another Internet socket address structure with the server's IP address (taken from the command line) and port (13). We call our `tpi_connect` function to establish the connection.

Read data from server, copy to standard output

23-33 As in our other daytime clients, we just copy data from the connection to standard output, stopping when we receive the end-of-file from the server (e.g., the FIN). We have written this loop to look like our sockets client (Figure 1.5) instead of our XTI client (Figure 28.13), because our `tpi_read` function will convert an orderly release from the server into a return of 0. We then call our `tpi_close` function to close our endpoint.

Our `tpi_bind` function is shown in Figure 33.9.

Fill in `T_bind_req` structure

16-20 The `<sys/tihdr.h>` header defines the `T_bind_req` structure:

```
struct T_bind_req {
  long           PRIM_type;      /* T_BIND_REQ */
  long           ADDR_length;    /* address length */
  long           ADDR_offset;    /* address offset */
  unsigned long CONIND_number; /* connect indications requested */
      /* followed by the protocol address for bind */
};
```

All TPI requests are defined as a structure that begins with a long integer type field. We define our own `bind_req` structure that begins with the `T_bind_req` structure, followed by a buffer containing the local address to be bound. TPI says nothing about the contents of this buffer; it is defined by the provider. TCP providers expect this buffer to contain a `sockaddr_in` structure.

We fill in the `T_bind_req` structure, setting the `ADDR_length` member to the size of the address (16 bytes for an Internet socket address structure) and `ADDR_offset` to the byte offset of the address (it immediately follows the `T_bind_req` structure). We are not guaranteed that this location is suitably aligned for the `sockaddr_in` structure that is stored there, so we call `memcpy` to copy the caller's structure into our `bind_req` structure. We set `CONIND_number` to 0, because we are a client, not a server.

Call `putmsg`

21-23 TPI requires that the structure that we just built be passed to the provider as one `M_PROTO` message. We therefore call `putmsg` specifying our `bind_req` structure as the control information, with no data and with a flag of 0.

Call `getmsg` to read high-priority message

24-30 The response to our `T_BIND_REQ` request will be either a `T_BIND_ACK` message or a `T_ERROR_ACK` message. These acknowledgment messages are sent as high-priority messages (`M_PCPROTO`) so we read them using `getmsg` with a flag of `RS_HIPRI`. Since the reply is a high-priority message, it will bypass any normal-priority messages on the stream.

streams/tpi_bind.c

```
1 #include    "tpi_daytime.h"

2 void
3 tpi_bind(int fd, const void *addr, size_t addrlen)
4 {
5     struct {
6         struct T_bind_req msg_hdr;
7         char    addr[128];
8     } bind_req;
9     struct {
10        struct T_bind_ack msg_hdr;
11        char    addr[128];
12    } bind_ack;
13    struct strbuf ctlbuf;
14    struct T_error_ack *error_ack;
15    int     flags;

16    bind_req.msg_hdr.PRIM_type = T_BIND_REQ;
17    bind_req.msg_hdr.ADDR_length = addrlen;
18    bind_req.msg_hdr.ADDR_offset = sizeof(struct T_bind_req);
19    bind_req.msg_hdr.CONIND_number = 0;
20    memcpy(bind_req.addr, addr, addrlen);   /* sockaddr_in{} */

21    ctlbuf.len = sizeof(struct T_bind_req) + addrlen;
22    ctlbuf.buf = (char *) &bind_req;
23    Putmsg(fd, &ctlbuf, NULL, 0);

24   ctlbuf.maxlen = sizeof(bind_eck);
25   ctlbuf.len = 0;.
26   ctlbuf.buf = (char *) &bind_ack;
27   flags = RS_HIPRI;
28   Getmsg(fd, &ctlbuf, NULL, &flags);
29   if (ctlbuf.len a (int) sizeof(long))
30       err_quit("bad length from getmsg");

31   switch (bind_ack.msg_hdr.PRIM_type) {
32   case T_BIND_ACK:
33       return;

34   case T_ERROR_ACK:
35       if (ctlbuf.len < (int) sizeof(struct T_error_ack))
36           err_quit("bad length for T_ERROR_ACK");
37       error_ack = (struct T_error_ack *) &bind_ack.msg_hdr;
38       err_quit(“T_ERROR_ACK from bind (%d, %d)*,
39                   error_ack->TLI_error, error_ack->UNIX_error);
40   default:
41       err_quit("unexpected message type: %d", bind_ack.msg_hdr.PRIM_type);
42   }
43 }
```

streams/tpi_bind.c

Figure 33.9 `tpi_bind` function: bind a local address to an endpoint.

These two messages are

```
   struct T_bind_ack {
      long             PRIM_type;     /* T_ERROR_ACK */
      long             ADDR_length;   /* address length */
      long             ADDR_offset;   /* address offset */
      unsigned long    CONIND_number; /* connect ind to be queued */
        /* followed by the bound address */
};

   struct T_error_ack {
      long            PRIM_type;      /* T_ERROR_ACK */
      long            ERROR_prim      /* primitive in error */
      long            TLI_error;      /* TLI error code */
      long            UNIX_error;     /* UNIX error code */
};
```

All these messages begin with the type, so we can read the reply assuming it is a T_BIND_ACK message, look at the type, and process the message accordingly. We do not expect any data from the provider, so we specify a null pointer as the third argument to getmsg.

> When we verify that the amount of control information returned is at least the size of a long integer, we must be careful to cast the sizeof value to an integer. The sizeof operator returns an unsigned integer value but it is possible for the returned len field to be -1. But since the less-than comparison is comparing a signed value on the left to an unsigned value on the right, the compiler casts the signed value to an unsigned value. On a twos-complement architecture, -1 considered as an unsigned value is very large, causing -1 to be greater than 4 (if we assume a long integer occupies 4 bytes).

Process reply

31-33 If the reply is T_BIND_ACK, the bind was successful, and we return. The actual address that was bound to the endpoint is returned in the addr member of our bind_ack structure, which we ignore.

34-39 If the reply is T_ERROR_ACK, we verify that the entire message was received and then print the three return values in the structure. In this simple program we terminate when an error occurs; we do not return to the caller.

We can see these errors from the bind request by changing our main functV to bind some port other than 0. For example, if we try to bind port 1 (which requires superuser privileges, since it is a port less than 1024) we get

```
aix % tpi_daytime 206.62.226.33
T_ERROR_ACK from bind (3, 0)
```

The error EACCES has the value of 3 on this system. If we change the port to a value greater than 1023, but one that is currently in use by another TCP endpoint, we get

```
aix % tpi_daytime 206.62.226.33
T_ERROR_ACK from bind (23, 0)
```

The error EADDRBUSY has a value of 23 on this system.

This error is new with the TPI to support XTI. Older versions of TPI that support TLI would bind another unused port if the requested one was busy. This meant that a server binding a well-known port would have to compare the returned address (from the T_bind_ack message, which is returned by t_bind if the third argument is a nonnull pointer) to the requested address, and abort if they were not equal.

The next function, shown in Figure 33.10, is `tpi_connect`, which establishes the connection with the server.

Fill in request structure and send to provider

13-26 TPI defines a `T_conn_req` structure that contains the protocol address and options for the connection:

```
struct T_conn_req {
   long     PRIM_type;     /* T_CONN_REQ */
   long     DEST_length;  /* destination address length */
   long     DEST_offset;  /* destination address offset */
   long     OPT_length;   /* options length */
   long     OPT_offset;   /* options offset */
       /* followed by the protocol address and options for connection */
};
```

As in our `tpi_bind` function, we define our own structure named `conn_req` that includes a `T_conn_req` structure along with room for the protocol address. We fill in our `conn_req` structure, setting the two members dealing with options to 0. We call `putmsg` with only control information and a flag of 0 to send an `M_PROTO` message down the stream.

streams/tpi_connect.c

```
 1 #include  "tpi_daytime.h"

 2 void
 3 tpi_connect(int fd, const void *addr, size_t addrlen)
 4 {
 5       struct {
 6         struct T_conn_req msg_hdr;
 7         char       addr[128];
 8       } conn_req;
 9       struct {
10           struct T_conn_con msg_hdr;
11           char     addr[128];
12       } conn_con;
13       struct strbuf ctlbuf;
14       union T_primitives rcvbuf;
15       struct T_error_ack *error_ack;
16       struct T_discon_ind *discon_ind;
17       int     flags;

18       conn_req.msg_hdr.PRIM_type = T_CONN_REQ;
19       conn_req.msg_hdr.DEST_length = addrlen;
20       conn_req.msg_hdr.DEST_offset = sizeof(struct T_conn_req);
21       conn_req.msg_hdr.OPT_length = 0;
22       conn_req.msg_hdr.OPT_offset = 0;
23       memcpy(conn_req.addr, addr, addrlen);  /*  sockaddr_in{} */
```

```
24      ctlbuf.len = sizeof(struct T_conn_req) + addrlen;
25      ctlbuf.buf = (char *) &conn_req;
26      Putmsg(fd, &ctlbuf, NULL, 0);

27      ctlbuf.maxlen = sizeof(union T_primitives);
28      ctlbuf.len = 0;
29      ctlbuf.buf = (char *) &rcvbuf;
30      flags = RS_HIPRI;
31      Getmsg(fd, &ctlbuf, NULL, &flags);
32      if (ctlbuf.len < (int) sizeof(long))
33          err_quit("tpi_connect: bad length from getmsg");

34      switch (rcvbuf.type) {
35      case T_OK_ACK:
36          break;

37      case T_ERROR_ACK:
38          if (ctlbuf.len < (int) sizeof(struct T_error_ack))
39              err_quit("tpi_connect: bad length for T_ERROR_ACK");
40          error_ack = (struct T_error_ack *) &rcvbuf;
41          err_quit("tpi_connect: T_ERROR_ACK from conn (%d, %d)",
42                   error_ack->TLI_error, error_ack->UNIX_error);

43      default:
44          err_quit("tpi_connect: unexpected message type: %d", rcvbuf.type):
45      }

46      ctlbuf.maxlen = sizeof(conn_con);
47      ctlbuf.len = 0;
48      ctlbuf.buf = (char *) &conn_con;
49      flags = 0;
50      Getmsg(fd, &ctlbuf, NULL, &flags);
51      if (ctlbuf.len < (int) sizeof(long))
52          err_quit("tpi_connect2: bad length from getmsg");

53      switch (conn_con.msg_hdr.PRIM_type) {
54      case T_CONN_CON:
55          break;

56      case T_DISCON_IND:
57          if (ctlbuf.len < (int) sizeof(struct T_discon_ind))
58              err_quit("tpi_connect2: bad length for T_DISCON_IND");
59          discon_ind = (struct T_discon_ind *) &conn_con.msg_hdr;
60          err_quit("tpi_connect2: T_DISCON_IND from comm (%d)",
61                   discon_ind->DISCON_reason);

62      default:
63          err_quit("tpi_connect2: unexpected message type: %d",
64                   conn_con.msg_hdr.PRIM_type);
65      }
66  }
```

streams/tpi_connect.c

Figure 33.10 `tpi_connect` function: establish connection with server.

Read response

27-45 We call getmsg expecting to receive either a `T_OK_ACK` message

```
struct T_ok_ack {
  long      PRIM_type;     /* T_0K_ACK    */
  long      CORRECT prim;  /* correct primitive */
};
```

if the connection establishment was started, or a T_ERROR_ACK message (which we showed earlier). In the case of an error, we terminate. Since we do not know what type of message we will receive, a union named T_primitives is defined as the union of all the possible requests and replies, and we allocate one of these that we use as the input buffer for the control information when we call getmsg.

Wait for connection establishment to complete

46-65 The successful T_OK_ACK message that was just received only tells us that the connection establishment was started. We must now wait for a T_CONN_CON message to tell us that the other end has confirmed the connection request.

```
structT_conn_con {
  long   PRIM_type;        /* T_CONN_CON */
  long   RES_length;       /* responding address length */
  long   RES_offset;       /* responding address offset */
  long   OPT_length;       /* option length */
  long   OPT_offset;       /* option offset */
      /* followed by peer's  protocol address and options */
};
```

We call getmsg again, but the expected message is sent as an M_PROTO message, not an M_PCPROTO message, so we set the flags to 0. If we receive the T_CONN_CON message, the connection is established, and we return, but if the connection was not established (either the peer process was not running, a timeout, or whatever), a T_DISCON_IND message is sent up the stream instead:

```
struct T_discon_ind {
  long    PRIM_type;      /* T_DISCON_IND */
  long    DISCON_reason;  /* disconnect reason */
  long    SEQ_number;     /* sequence number */
```

We can see the different errors that are returned by the provider. We first specify the IP address of a host that is not running the daytime server:

```
solaris26 % tpi_daytime 140.252.1.4
tpi_connect2: T_DISCON_IND from conn (146)
```

The error of 146 corresponds to ECONNREFUSED. Next we specify an IP address that is not connected to the Internet:

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (145)
```

The error this time is ETIMEDOUT. But if we run our program again, specifying the same IP address, we get a different error:

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (148)
```

The error this time is EHOSTUNREACH. The difference in the last two results is that the first time no ICMP host unreachable errors were returned, while the next time this error was returned.

The next function is tpi_read, shown in Figure 33.11. It reads data from a stream.

streams/tpi_read.c

```
1 #include   "tpi_daytime.h"

2 ssize_t
3 tpi_read(int fd, void *buf, size_t len)
4 {
5     struct strbuf ctlbuf;
6     struct strbuf datbuf;
7     union T_primitives rcvbuf;
8     int     flags;

9     ctlbuf.maxlen = sizeof(union T_primitives);
10    ctlbuf.buf = (char *) &rcvbuf;

11    datbuf.maxlen = len;
12    datbuf.buf = buf;
13    datbuf.len = 0;

14    flags = 0;
15    Getmsg(fd, &ctlbuf, &datbuf, &flags);

16    if (ctlbuf.len >= (int) sizeof(long)) {
17        if (rcvbuf.type == T_DATA_IND)
18            return (datbuf.len);
19        else if (rcvbuf.type == T_ORDREL_IND)
20            return (0);
21        else
22            err_quit("tpi_read: unexpected type %d", rcvbuf.type);
23    } else if (ctlbuf.len == -1)
24        return (datbuf.len);
25    else
26        err_quit("tpi_read: bad length from getmsg");
27 }
```

streams/tpi_read.c

Figure 33.11 tpi_read function: read data from a stream.

Read control and data; process reply

9-26 This time we call getmsg to read both control information and data. The strbuf structure for the data points to the caller's buffer. Four different scenarios can occur on the stream.

- The data can arrive as an M_DATA message, and this is indicated by the returned control length set to -1. The data was copied into the caller's buffer by getmsg, and we just return the length of this data as the return value of the function.
- The data can arrive as a T_DATA_IND message, in which case the control information will be a T_data_ind structure:

```
struct T_data_ind {
  long    PRIM_type; /* T_DATA_IND */
  long    MORE_flag; /* more data */
};
```

If this message is returned, we ignore the MORE_flag member (it will never be set for a stream protocol such as TCP) and just return the length of the data that was copied into the caller's buffer by getmsg.

- A T_ORDREL_IND message is returned if all the data has been consumed and the next item is a FIN:.

```
struct T_ordrel_ind {
 long    PRIM_type:    /* T_ORDREL_IND
*/ };
```

This is the orderly release that we described in Section 28.9. We just return 0, indicating to the caller that the end-of-file has been encountered on the connection.

- A T_DISCON_IND message is returned if a disconnect has been received. We discussed this in Section 28.10 and said this occurs in the case of TCP if an RST is received on an existing connection. We do not handle this scenario in this simple example, but we did handle it in Figure 28.13.

We can now explain the two different scenarios that we saw in Section 28.12 when we called read but had not pushed the tirdwr module onto the stream. In the first example, which generated the error "read error: Not a data message," the provider had sent a T_DATA_IND message up the stream as an M_PROTO message (since it had control and data). But read handles only M_DATA messages, hence the error.

In the second example the error was "read error: Bad message" but this appeared after the server's expected reply was received and printed. On this implementation the provider sent the data up the stream as an M_DATA message, so it was handled by read correctly. But the next message up the stream was a T_ORDREL_IND message, which read cannot handle.

Our final function is tpi_close, shown in Figure 33.12.

Send orderly release to peer

7-10 We build a T_ordrel_req structure

```
struct T_ordrel_req {
  long PRIM_type;    /* T_ORDREL_REQ */
};
```

and send it as an M_PROTO message using putmsg. This corresponds to the XTI t_sndrel function.

This example has given us a flavor for TPI. The application sends messages down a stream to the provider (requests) and the provider sends messages up the stream (replies). Some exchanges are a simple request–reply scenario (binding a local address) while others may take a while (establishing a connection), allowing us to do something

streams/tpi_close.c

```
1 #include     "tpi_daytime.h"

2 void
3 tpi_close(int fd)
4 {
5       struct T_ordrel_req ordrel_req;
6       struct strbuf ctlbuf;

7       ordrel_req.PRIM_type = T_ORDREL_REQ;

8       ctlbuf.len = sizeof(struct T_ordrel_req);
9       ctlbuf.buf = (char *) &ordrel_req;
10      Putmsg(fd, &ctlbuf, NULL, 0);

11      Close (fd);
12 }
```

streams/tpi_close.c

Figure **33.12** `tpi_close` function: send an orderly release to peer.

while we wait for the reply. Our choice of writing a TCP client using TPI was done for simplicity; writing a TCP server and handling connections as we described in Section 30.7 becomes much harder.

It should be obvious that the mapping from the XTI functions to TPI is very close. On the other hand, the mapping from sockets to TPI is not as close. Nevertheless, both the XTI and socket libraries handle lots of the details required by TPI, simplifying our applications.

> We can compare the number of system calls required for the network operations that we have seen in this chapter, when using TPI versus a kernel that implements sockets within the kernel. Binding a local address takes two system calls with TPI, but only one with kernel sockets (TCPv2, p. 454). To establish a connection on a blocking descriptor takes three system calls with TPI, but only one with kernel sockets (TCPv2, p. 466).

33.7 Summary

XTI is often implemented using streams. Four new functions are provided to access the streams subsystem, `getmsg`, `getpmsg`, `putmsg`, and `putpmsg`, and the existing `ioctl` function is heavily used by the streams subsystem also.

TPI is the SVR4 streams interface from the upper layers into the transport layer. It is used by both XTI and sockets, as shown in Figure 33.3. We developed a version of our daytime client using TPI directly, as an example to show the message-based interface that TPI uses.

Exercises

33.1 In Figure 33.12 we call putmsg to send the orderly release request down the stream and then immediately close the stream. What happens if our orderly release request is lost by the streams subsystem when the stream is closed?

34

XTI: Additional Functions

34.1 Introduction

In the previous chapters we have covered the XTI functions for

- TCP clients,
- hostname and service **name** lookups,
- TCP servers,
- UDP clients and servers,
- options, and
- the common streams implementation.

This chapter covers the remaining XTI functions.

34.2 Nonblocking I/O

An endpoint can be put into a nonblocking mode. This is done by specifying the `O_NONBLOCK` flag in the call to `t_open` when the endpoint is created, or at a later time with the `fcntl` function (as shown in Section 7.10).

The operation of some of the XTI functions changes when the endpoint is nonblocking.

- `t_connect` returns immediately with a return of -1 and `t_errno` set to `TNODATA`. With TCP this call initiates the three-way handshake, and we must call `t_rcvconnect` (Section 34.3) to wait for the connection establishment to complete.

- `t_rcvconnect` returns -1 with `t_errno` set to `TNODATA` if a connection is in progress but has not yet completed.
- `t_listen` returns immediately with a return of -1 and `t_errno` set to `TNODATA` when there are no connections ready for the application to call `t_accept`.
- The four receive functions, `t_rcv`, `t_rcvudata`, `t_rcvv`, and `t_rcvvudata`, return -1 with `t_errno` set to `TNODATA` if there is no data available. If some data is available, that data is returned, even though it may be less than asked for by the application. (The last two functions mentioned are new; we describe them in Section 34.8.)
- The four send functions, `t_snd`, `t_sndudata`, `t_sndv`, and `t_sndvudata`, return -1 with `t_errno` set to `TFLOW` if the provider is not able to accept any data. If some data can be accepted, then the return value might be less than the amount requested for `t_snd` and `t_sndv`. The two datagram functions write a complete datagram, or they return an error. (The last two of the four functions listed are new; we describe them in Section 34.9.)

34.3 `t_rcvconnect` Function

In the previous section we mentioned initiating a connection in the nonblocking mode and then waiting for the connection to complete by calling `t_rcvconnect`.

```
#include <xti.h>

int t_rcvconnect(int fd, struct t_call *recvcall);
```

Returns: 0 if OK, -1 on error

The sequence of steps typically used with this function are as follows:

1. An endpoint is created using `t_open` and set nonblocking.
2. `t_connect` initiates the connection establishment. Since the endpoint is in the nonblocking mode, this function returns immediately with a value of -1 and `t_errno` set to `TNODATA`.
3. At some later time the process calls `t_rcvconnect` to determine if the connection has completed. If the endpoint is no longer in a nonblocking mode (the process has turned off the nonblocking flag since calling `t_connect` in step 2), then `t_rcvconnect` blocks until the connection is established. If the endpoint is still in a nonblocking mode, then this call to `t_rcvconnect` either (a) returns immediately with a return value of 0 if the connection is established, or (b) returns a value of -1 with `t_errno` set to `TNODATA` if the connection is not vet established.

Note in the case of a blocking `t_connect` (the default), the provider returns the information in the `t_call` structure that is pointed to by the third argument to `t_connect`. But with a nonblocking `t_connect`, this information is returned in the `t_call` structure that is pointed to by the second argument to `t_rcvconnect`.

Unless the application converts the endpoint from nonblocking to blocking between the calls to `t_connect` and `t_rcvconnect` (steps 2 and 3 above), calling `t_rcvconnect` to determine when a nonblocking connection establishment completes is a waste.of time, because the application must call `t_rcvconnect` in a loop of some form, waiting for the connection to complete (or an error to be returned). This is called *polling*. Better techniques for waiting for a nonblocking connection establishment to complete are to call either `select` or `poll` (Chapter 6), or to use signal-driven I/O (Section 34.11).

Recall our discussion of an interrupted connection establishment at the end of Section 15.4. With XTI, if a call to `t_connect` on a blocking endpoint is interrupted, we just call `t_rcvconnect` to wait for the connection establishment to complete.

34.4 t_getinfo Function

Recall the `t_info` structure that is returned by the `t_open` function (Section 28.2). The following function returns the same information to the caller.

```
#include <xti.h>

int t_getinfo(int fd, struct t_info *info);
```

Returns: 0 if OK, -1 on error

This function is called, for example, by `t_alloc`, to obtain the information about an endpoint that is already open for `t_alloc` to obtain the required buffer sizes.

34.5 t_getstate Function

Every transport endpoint has *a current state* associated with it. The following function returns the current state (an integer value) to the caller.

```
#include <xti.h>

int t_getstate(int fd);
```

Returns: current state if OK, -1 on error

The current state is specified by one of the constants shown in Figure 34.1. The final three columns indicate which states are valid for the different service types (Figure 28.3).

State	Description	T_COTS	T_COTS_ORD	T_CLTS
T_DATAXFER	data transfer	•	•	
T_IDLE	bound, but idle	•	•	•
T_INCON	incoming connection pending for passive endpoint	•	•	
T_INREL	incoming orderly release		•	
T_OUTCON	outgoing connection pending for active endpoint	•	•	
T_OUTREL	outgoing orderly release		•	
T_UNBND	unbound	•	•	•
T_UNINIT	uninitialized: starting and final state	•	•	•

Figure 34.1 Possible states of an XTI endpoint.

A state transition diagram can be developed to show exactly how the state of a transport endpoint changes as different XTI functions are called and as different events occur at the endpoint. This diagram would also show which XTI functions are allowed in the different states. For example, the only function call allowed in the `T_UNINIT` state is `t_open` and the new state becomes `T_UNBND.` Four events can occur in the `T_UNBND` state:

1. A successful return from `t_close` changes the state to `T_UNINIT.`
2. Calling `t_optmgmt` is allowed but does not change the state. (What this state transition diagram cannot show, however, is that the option processing might change depending on the state. For example, the `T_UDP_CHECKSUM` option behaves differently in the `T_UNBND` state, versus other states.)
3. A successful return from `t_bind` changes the state to `T_IDLE.`
4. Passing a connection to the endpoint (by `t_accept)` is allowed, and changes the state to `T_DATAXFER.`

Once we get past these first two states, however, the diagram becomes unwieldy, so we will not attempt to show it.

34.6 t_sync Function

Historically TLI was implemented as a library of functions in SVR3. Consider % pro-gram using TLI that calls exec as shown in Figure 34.2. Perhaps the program on the left is a listening server that waits for a connection to arrive and be accepted and then execs the program on the right to handle the client. (Remember that the process ID does not change across an exec, but the caller's memory is replaced with the new pro-gram that then begins execution at its `main` function.)

The problem encountered with this scenario in SVR3 was that state information is maintained in both the TLI library within the process and in the provider within the kernel. After an exec all of the state information in the library is discarded, and the library in the new program starts off fresh. The purpose of the `t_sync` function was to allow the new program (on the right in Figure 34.2) to synchronize the state of its library with the provider in the kernel.

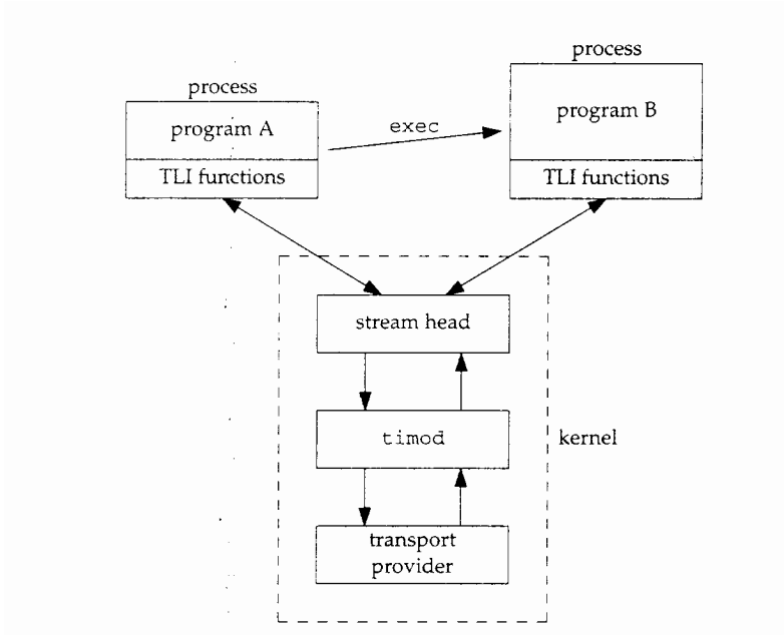

Figure 34.2 TLI implementation when a process calls exec.

With SVR4, however, the need to call t_sync for the scenario that we show in Figure 34.2 disappeared. The TLI library functions could detect the need for performing a synchronization themselves. For example, if the library has a variable declared as

```
static int synced;      /* initialized to 0 when program starts */
```

then each function can begin with the sequence similar to the following:

```
int
t_connect(fd, ... )
{
    if (synced == 0)
        t_sync(fd); /* also sets synced = 1 */
    ...

}
```

While this handles the case of a process calling exec, there is still one scenario, albeit rare, where t_sync is required: when multiple processes are sharing an XTI endpoint. In this scenario it is assumed that the processes are cooperating with each other and that each calls t_sync when it deems necessary (which depends, of course, on the specifics of the application). One example where t_sync might be necessary is in a parent—child relationship, if the parent calls t_l is t en and then the child calls t_accept. The parent would need to call t_sync to update its library copy of the endpoint state, which might change after the child calls t_accept.

```
*include <xti.h>
int t_sync(int fd);
```

Returns: current state if OK, -1 on error

The successful return from this function is one of the states shown in Figure 34.1.

There is no operation similar to this function in the sockets API.

34.7 t_unbind Function

The effect of the `t_bind` function is undone by `t_unbind`.

```
*include <xti.h>
int t_unbind(int fd);
```

Returns: 0 if OK, -1 on error

This function disables the transport endpoint specified by *fd*. No further data will be accepted for this endpoint. `t_bind` may be called, however, to bind another local address to the endpoint.

With sockets this operation can be performed only on a connected UDP socket by calling `bind` with an invalid address.

34.8 t_rcvv and t_rcvvudata Functions

These two functions extend the `t_rcv` and `t_rcvudata` functions to operate on a *vector* of buffers, instead of just a single buffer. They provide *a scatter read* capability.

These two functions and the two described in the next section were introduced with Posix.lg.

The concept of operating on a vector of buffers comes from the `ready` and `write` functions, along with the `recvmsg` and `sendmsg` functions.

```
*include <xti.h>
int t_rcvv(int fd, struct t_iovec *iov, unsigned int iovcnt, int *flags);
int t_rcvvudata(int fd, struct t_unitdata *unitdata,
          struct t_iovec *iov, unsigned int iovcnt, int *flags);
```

Both return: number of bytes read or written if OK, -1 on error

The *iov* argument to both functions is a pointer to an array of `t_iovec` structures:

```
struct t_iovec {
  void    *iov_base; /* starting address of buffer */
  size_t iov_1en;     /* length of buffer in bytes */
}
```

The number of entries in the array is specified by the *iovcnt* argument. The limit on the number of entries in the array is given by the constant T_I0V_MAX, defined by including the <xti.h> header, whose value must be at least 16.

Comparing these new functions to their earlier counterparts we see the following:

- The buffer pointer and its length are the middle two arguments to t_rcv.
- The buffer pointers and their lengths are in an array of t_iovec structures for t_rcvv, and the middle two arguments for this function point to this array of structures and specify the number of entries in the array.

- The buffer pointer and its length are in the udata member of the t_unitdata structure for t_rcvudata. Also, this function returns 0 upon success, with the actual length of the received datagram in the udata.len member of the t_unitdata structure.

- The buffer pointers and their lengths are in an array of t_iovec structures for t_rcvvudata. The third argument to this function is a pointer to this array of structures and the fourth argument is the number of entries in the array. A pointer to a t_unitdata structure is the second argument to this function and the addr and opt member are still used (for the sender's protocol address and any received options), but the udata member is ignored. This function returns the number of bytes in the datagram as its return value, not 0.

34.9 t_sndv and t_sndvudata Functions

These two functions extend the t_snd and t_sndudata functions to operate on a *vector* of buffers, instead of just a single buffer. They are the send counterparts of the two functions described in the previous section and provide *a gather write* capability.

```
#include <xti.h>

int t_sndv (int fd, struct t_iovec *iov, unsigned int iovcnt, int flags) ;
```

Returns: number of bytes read or written if OK, -1 on error

```
int t_sndvudata(int fd, struct t_unitdata *unitdata, struct
                t_iovec *iov, unsigned int iovcnt) ;
```

Returns: 0 if OK, -1 on error

The *iov* argument to both functions is a pointer to an array of t_iovec structures, which we showed in the previous section. The number of entries in the array is specified by the *iovcnt* argument.

The output buffers are specified by the two middle arguments to `t_sndv`, replacing the two middle arguments to `t_snd`. For the datagram functions, the output buffer is specified by the `udata` member of the `t_unitdata` structure with `t_sndudata` but by the iov vector with `t_sndvudata`. The `udata` member of the `t_unitdata` structure is ignored by `t_sndvudata`.

34.10 `t_rcvreldata` and `t_sndreldata` Functions

If we send an orderly release with `t_sndrel` (Section 28.9) we cannot send data with the orderly release notification (the only argument to the function is a descriptor), but if we send a disconnect with `t_snddis` (Section 28.10), we can send data (the `udata` member of the `t_call` structure). We find the same limitation for `t_rcvrel`, compared to `t_rcvdis`. To get around this limitation XTI invented two new functions that send and receive data with an orderly release.

```
#include <xti.h>

int t_sndreldata(int fd, const struct t_discon *discon)

int t_rcvreldata(int fd, struct t_discon *discon);
```

Both return: 0 if OK, -1 on error

The difference between these two functions and `t_sndrel` and `t_rcvrel` is the addition of the second argument (a pointer to a `t_discon` structure).

These functions are useful only when the provider supports the sending of data with an orderly release, as indicated by the `T_ORDRELDATA` flag in the `flag` member of the `t_info` structure (Figure 28.4). If supported, the amount of orderly release data is limited to the value of the `discon` member of the `t_info` structure.

TCP does not support this optional feature.

34.11 Signal-Driven I/O

Signal-driven I/O is provided by the streams system, not XTI. The signal name is `SIGPOLL` and the signal is not delivered just because the process installs a signal handling function for the signal. The process must also tell the kernel that it wants to receive the signal by issuing the `I_SETSIG` streams `ioctl` request, specifying which conditions should generate the signal. This is similar to what we must do to receive the `SIGIO` and `SIGURG` signals that we described for the sockets API.

The third argument to `ioctl` is an integer value that specifies the conditions for which a `SIGPOLL` signal should be generated. If this value is 0, the process will no longer receive the `SIGPOLL` signal for the stream. This value can also be formed as the logical OR of the following constants:

S_BANDURG	If this flag is specified in conjunction with S_RDBAND, the SIGURG signal will be generated instead of SIGPOLL when a message in a priority band greater than 0 can be read.
S_ERROR	The stream is in error.
S_HANGUP	A hangup message has reached the stream head.
S_HIPRI	A high-priority message can be read.
S_INPUT	This is equivalent to S_RDNORM S_RDBAND and means that a message with any band (including 0) can be read.
S_OUTPUT	The write queue just below the stream head is no longer flow controlled for normal messages (band 0).
S_MSG	A streams signal message is at the front of the stream's read queue
S_RDNORM	A normal message (band 0) can be read.
S_RDBAND	A message in a priority band greater than 0 can be read
S_WRNORM	Equivalent to S_OUTPUT.
S_WRBAND	The write queue is no longer flow controlled for messages in a priority band greater than 0.

The S_BANDURG flag is used by the sockets API when it is implemented using streams.

There is no output equivalent for S_HIPRI. This is because putmsg and putpmsg do not block when sending a high-priority message: these messages are not flow controlled.

The streams signal SIGPOLL is used for both signal-driven I/O and for the notification of the arrival of out-of-band data. This corresponds to the two signals SIGIO and SIGURG that we described with the sockets API.

The default action for SIGPOLL is to terminate the process, so when using this signal we must establish the signal handler and then call ioctl to enable the signal.

There is a conflict between the default action of SIGPOLL, which we just said terminates the process, and SIGIO. Posix.lg specifies that the default action of SIGIO is to be ignored. Since SVR4 systems define these two signals to be the same, these systems will have to change the default action for SIGPOLL to be ignored, to be Posix.lg compliant.

Even though the SIGPOLL signal can be generated for numerous conditions, typical applications that do not deal with out-of-band data are interested in only S_RDNORM and S_WRNORM.

34.12 Out-of-Band Data

Out-of-band data is called *expedited data* by XTI. Support for this feature is provided by the transport provider and the streams system. We mentioned in Chapter 33 that

out-of-band data is often implemented as normal-priority data in priority band 1. Normal data (i.e., not out-of-band data) is in priority band 0.

> We also mentioned in Chapter 33 that since TCP's out-of-band data is not true expedited data (in the XTI sense), it is actually implemented in band 0, not band 1.

Everything that we said in Chapter 21 about the support for out-of-band data with TCP, and the mapping of TCP's urgent mode into out-of-band data, applies to XTI just like sockets.

Out-of-band data is sent with the XTI `t_snd` function by specifying *a flags* argument of `T_EXPEDITED`. This flag value is also returned to the caller by the `t_rcv` function.

We cannot use the `read` and `write` functions when writing an application that deals with out-of-band data (recall our `xti_rdwr` function in Section 28.12). We must use `t_snd` and `t_rcv`.

Since TCP's out-of-band data corresponds to normal-priority messages in band 0, to receive `SIGPOLL` when out-of-band data arrives requires that we specify `S_RDNORM` when we call `ioctl` with a request of `I_SETSIG` (Section 34.11). Since this also generates the signal when normal data arrives, if we want to differentiate between normal data and out-of-band data, we must call `t_look` from our signal handler and check for either `T_DATA` or `T_EXDATA` (Figure 28.9). XTI sets the event `T_EXDATA` as soon as a TCP segment with an urgent pointer is received, and this event remains set until all data up through the urgent pointer has been received.

> `SIGPOLL` corresponds to the `SIGURG` signal for sockets.

If using `poll` to await the arrival of out-of-band data (Section 6.10), the `events` member of the `po11fd` structure must be set to `POLLRDNORM` since it appears as normal data in band 0. We will verify that TCP's out-of-band data appears as normal data to `poll` in Figures 34.4 and C.5. Also note that this treatment of out-of-band data with XTI differs from sockets, which considers out-of-band data as belonging to a priority band.

> Using `poll` corresponds to calling select and waiting for an exception condition, but `poll` does not tell us what type of data arrived: we must call `t_look` and `t_rcv`.
>
> Recall that by default the sockets API removes a received out-of-band byte from%he normal stream of data, placing it into its own special 1-byte buffer that the application reads using recv with the `MSG_00B` flag. There is nothing similar to this mode with XTI: TCP's out-ofband data is always received inline, something we have to enable with sockets using the `SO_OOBINLINE` socket option.

We now look at a few examples to see how signal-driven I/O and `poll` work with XTI's out-of-band data.

Example Using sIGPoLL

Figure 34.3 is a program that uses `SIGPOLL` to be notified when data is available on an XTI endpoint.

xtioob/tcprecv0l.c

```
 1 #include    "urpxti.h"
 2 #define NREAD    100
 3 int      listenfd, connfd;
 4 void     sig_poii(int);
 5 int
 6 main(int argc, char **argv)
 7 {
 8     int      n, flags;
 9     char     buff[NREAD + 1];       /* +1 for null at end */
10     if (argc == 2)
11          listenfd = Tcp_listen(NULL, argv[1], NULL);
12     else if (argc == 3)
13          listenfd = Tcp_listen(argv[1], argv[2], NULL);
14     else
15          err_quit("usage: tcprecvtl [ <host> ] <port#>");
16     connfd = Xti_accept(listenfd, NULL, NULL);
17      Signal(SIGPOLL, sig_poll);
18      Ioctl(connfd, I_SETSIG, S_RDNORM);
19      for ( ; ; ) {
20          flags = 0;
21          if ( (n = t_rcv(connfd, buff, NREAD, &flags)) < 0) {
22              if (t_errno == TLOOK) {
23                  if ( (n = T_look(connfd)) == T_ORDREL) {
24                      printf("received T_ORDREL\n");
25                      exit(0);
26                  } else
27                      err_quit("unexpected event after t_rcv: %d", n);
28              }
29              err_xti("t_rcv error");
30          }
31          buff[n] = 0;             /* null terminate */
32          printf("read %d bytes: %s, flags = %s\n",
33                 n, buff, Xti_flags_str(flags));
34     }
35 }
36 void
37 sig_poll(int signo)
38 {
39      printf("SIGPOLL received, event = %s\n", Xti_tlook_str(connfd));
40 }
```

xtioob/tcprecvOl .c

Figure 34.3 Receive normal and out-of-band data using `SIGPOLL` on an XTI endpoint.

Create listening endpoint and wait for connection

10-16 We call our `tcp_listen` function to create a listening endpoint and then our `xti_accept` function waits for a connection to arrive and accepts it.

Establish signal handler

17-18 We call `signal` to establish a signal handler for `SIGPOLL` and then call `ioctl` to enable the signal to be generated when normal data arrives for the endpoint.

Loop, reading data

19-34 We call `t_rcv` to receive the data, handling an orderly release when the peer closes the connection. We print the data bytes that are received, along with the `flags` returned by `t_rcv`. Our function `xti_flags_str` returns a pointer to a message describing the flags that are passed as an argument.

Signal handler

36-40 Our signal handler just prints a message that includes the current event for the endpoint. Our function `xti_tlook_str` calls `t_look` and returns a pointer to a message describing the current event for the endpoint.

We start this program and then run the program from Figure 21.3 as the client. Here is the output from our server:

```
unixware % tcprecvOl 9999
read 3 bytes: 123, flags = 0
SIGPOLL received, event = T_EXDATA
read 1 bytes: 4, flags = T_EXPEDITED
SIGPOLL received, event = T_DATA
read 2 bytes: 56, flags = 0
SIGPOLL received, event = T_EXDATA
read 1 bytes: 7, flags = T_EXPEDITED
SIGPOLL received, event = T_DATA
read 2 bytes: 89, flags = 0
SIGPOLL received, event = T_ORDREL
received T_ORDREL
```

The first 3 bytes are received as normal data but `SIGPOLL` is not generated. This is a timing issue. Recall from Figure 2.5 that the client's `connect` (or `t_connect` if XTI is being used) returns one-half an RTT before the server's `accept` (or `t_accept`), given how the TCP three-way handshake operates. This gives the client a head start in sending its first segment of data, and we see in this example that the first 3 bytes arrive before we establish our signal handler.

We then receive `SIGPOLL` and the event is `T_EXDATA`. `t_rcv` returns a flag of `T_EXPEDITED`. We can see from the remaining lines of output that each time a TCP segment arrives, `SIGPOLL` is generated, and we must call `t_look` to see what event has occurred.

When we have read all the data and are notified that the client closed its end of the connection, the signal is generated and the event is `T_ORDREL`, as expected.

Example Using `poll`

Our next example, shown in Figure 34.4, uses the `poll` function.

xtioob/tcprecv03.c

```
1 *include     "unpxti.h"

2 #define NREAD    100
3 int      listenfd, connfd;

4 int
5 main(int argc, char **argv)
6 {
7      int     n, flags;
8      char    buff[NREAD + 1];       /* +1 for null at end */
9      struct pollfd pollfd[l];

10     if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], NULL);
12     else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], NULL);
14     else
15        err_quit("usage: tcprecv03 [ <host> ] <port#>");

16     connfd = Xti_accept(listenfd, NULL, NULL);

17    pollfd[0].fd = connfd;
18    pollfd[0].events = POLLIN;

19    for ( ; ; ) {
20        Poll(pollfd, 1, INFTIM);

21        printf("revents = %x\n", pollfd[0].revents);
22        if (pollfd[0].revents & POLLIN) {
23            flags = 0;
24            if ( (n = t_rcv(connfd, buff, NREAD, &flags)) < 0) {
25                if (t_errno == TLOOK) {
26                    if ( (n = T_look(connfd)) == T_ORDREL) {
27                        printf("received T_ORDREL\n");
28                        exit(0);
29                    } else
30                        err_quit("unexpected event after t_rcv: %d", n);
31            }
32            err_xti("t_rcv error");
33        }
34        buff[n] = 0;    /* null terminate */
35        printf("read %d bytes: %s, flags = %s\n",
36              n, buff, Xti_flags_str(flags));
37        }
38   }
39 }
```

xtioob/tcprecv03.c

Figure 34.4 Receive normal and out-of-band data using `poll` on an `XTI` endpoint.

Wait for client connection

10-16 The creation of the listening endpoint and accepting the client's connection have not changed from Figure 34.3.

Prepare for poll

17-18 We allocate a 1-element pollfd array and initialize it to tell us when normal or priority data arrives for our connected endpoint.

Call poll

19-38 We call poll with an infinite time limit. When it returns, we print the revents member of our pollfd structure to see what type of data has arrived. If the event is POLLIN, we call t_rcv to read the data and print the data and the returned flags.

We run this program with Figure 21.3 as the client (the same client as in the previous example).

```
unixware % tcprecv03 7777
revents = 1
read 3 bytes: 123, flags = 0
revents = 1
read 1 bytes: 4, flags = T_EXPEDITED
revents = 1
read 2 bytes: 56, flags = 0
revents = 1
read 1 bytes: 7, flags = T_EXPEDITED
revents = 1
read 2 bytes: 89, flags = 0
revents = 1
received T_ORDREL
```

Each time poll returns, the event is 1, which is POLLIN on this system. t_rcv tells us the type of data being returned through the returned flags.

34.13 Loopback Transport Providers

Many implementations of XTI provide a loopback transport provider. The names in the netconfig file are normally ticlts, ticots, and ticotsord for the three types of XTI providers (Figure 28.3). The ti prefix stands for "transport independent." These three names are also the filenames in the / dev directory for t_open.

Unix domain sockets are often implemented on streams-based systetßts using two of these three providers: ticlts is used for SOCK_DGRAM sockets, and either ticots or ticotsord is used for SOCK_STREEAM sockets, depending which of the two appears first in the netconfig file.

One point to be aware of, when using these providers directly with XTI, is that the addresses used are called *flex addresses,* which are just arbitrary strings of one or more bytes. These addresses are not null terminated; their length is specified by the len member of the netbuf structure containing the address. But the addr member of the t_info structure can be returned as -1 (T_INFINITE), causing t_alloc not to allocate a buffer for the address.

One of the differences between TLI and XTI is the handling of T_INFINITE by t_alloc. TLI would allocate a 1024-byte buffer by default, while XTI does not allocate a buffer.

34.14 Summary

Nonblocking I/O is enabled for an XTI endpoint by specifying the `O_NONBLOCK` in the call to `t_open`, or anytime later with `fcntl`. The changes are similar to the changes with sockets when an endpoint is made nonblocking, with the exception of a nonblocking `t_connect`. We can wait for this to complete with the `t_rcvconnect` function.

Four new I/O functions are defined by XTI to operate on vectors of buffers: `t_rcvv`, `t_rcvvudata`, `t_sndv`, and `t_sndvudata`.

Signal-driven I/O is enabled for an XTI endpoint using `ioctl`, and we also must specify all the conditions under which the signal is to be generated: one of the `S_`*xxx* flags. Out-of-band data is sent by `t_snd` when the `T_EXPEDITED` flag is set. Nothing special need be done to receive out-of-band data: `t_rcv` returns a flag of `T_EXPEDITED`. A signal can also be generated when out-of-band data arrives.

Index

Networking is a field that is pockmarked with acronyms. Rather than provide a separate glossary (with most of the entries being acronyms), this index also serves as a glossary for all the acronyms used in the book. The primary entry for the acronym appears under the acronym name. For example, all references to the Internet Control Message Protocol appear under ICMP. The entry under the compound term "Internet Control Message Protocol" refers back to the main entry under ICMP.

The'notation "definition of" appearing with a C function refers to the boxed function prototype for that function, its primary description. The "definition of" notation for a structure refers to its primary definition. Some functions also contain the notation "source code" if the source code implementation for that function appears in the text.