FUJITSU COMPUTERS SIEMENS

Reliant UNIX ONLINE Documentation

Reliant UNIX 5.45

Network Programming Interfaces

Edition March 1999

Identification: U42311-J-Z915-1-76

Copyright and Trademarks

1 Preface

This manual is part of the user documentation that comes with the Reliant UNIX 5.45 operating system. It describes how to write application programs that use the Reliant UNIX network functions.

1.1 Target group

The manual is aimed at application programmers who want to develop software for use in networks. It is assumed that you have practical knowledge of Reliant UNIX and the C programming language. You should also be familiar with the OSI/ISO reference model.

1.2 Summary of contents

The Chapter "Transport Level Interface" (...) begins with a description of the OSI/ISO reference model and then provides you with detailed information on the Reliant UNIX transport level interface (TLI). You can use the TLI to create applications that run on a large number of networks without being modified.

The Chapter "The sockets interface" provides an introduction to the use of the socket routines and describes some of the functions you can use to write applications that are to be used in a network. In addition, the client/server model is discussed with the help of examples. More experienced users will find information on complex network programming questions.

The Chapter "Remote procedure calls" (...) provides an overview of the RPC programming functions and protocols. In the Section "rpcgen" you learn how to use the *rpcgen* compiler to create C programs that use RPC. The RPC programming language is described here. The Section "Programming with RPC" describes the C interface to the RPC environment. This section describes the different levels on which programmers can access the RPC mechanism. You receive guidelines on which level to select for which application - either a higher level in the interests of transparency and portability or a lower level for greater control over communication.

The Chapter "Network services" (...) explains the transport service selection and name-to-address mapping functions, which ensure that TLI applications are independent of transport media and protocols. The transport service selection feature provides a standard interface to the networks that are currently available. Name-to-address mapping allows applications to map transport service-specific addresses.

The Chapter "XDR/RPC protocol specification" (...) contains the specification of XDR (External Data Representation) and a complete language description of RPC.

1.3 Changes since the previous version of the manual

Most of the changes made to this manual since the previous edition (October 1997) refer to the changes to the socket interface standard (see the Chapter "The sockets interface").

In Reliant UNIX versions up to Version 5.45, Streams implementation was the standard used. From an application point of view, both implementations are available in Reliant UNIX 5.45, although the standard is now Berkeley Sockets (going back to 4.4BSD).

1.4 Notational conventions

The following notational conventions have been used in this manual:

fixed-space font	System outputs, inputs, text on the screen, mask or file contents, program examples, names, descriptions, etc.
italics	File names, pathnames, commands, parameters, variables, etc.
CAPITALS	Variable types in programs and for the system as well as abbreviations etc.

Notes and warnings


This symbol draws your attention to particularly important information that you really must read.


This symbol indicates actions that may lead to loss of data or damage to a device.

References

References are specified in the following format:

"Manual title" and/or corresponding reference number in square brackets.

2 Transport Level Interface

The Transport Level Interface (TLI) was introduced in UNIX System V Release 3 as an interface for transport provider-independent programming. Transport provider selection and name-to-address mapping were added in Release 4 in order to guarantee medium and protocol independence. The transport service selection facility and name-to-address mapping allow programmers to process network-specific information in network programs regardless of the transport protocol used.

TLI and sockets are programming interfaces for the transport layer. Both were implemented in UNIX System V Release 4 by means of the STREAMS mechanism. In Reliant UNIX 5.45 the TCP/IP protocol stack has been enhanced and standardized on the basis of 4.4BSD (detailed information can be found in chapter 1.).

The difference between the TLI and sockets interfaces differ in that

- the sockets can be accessed via the mechanism and also using the direct BSD mechanism,
- TLI is medium and protocol independent.

Applications can use any transport protocol that has a TLI interface.

Applications that use the TLI interface must be linked with the *libnsl* library:

cc *prog*.c -lnsl

In order to write TLI applications that run independently of the transport protocol used, you require a full understanding of the transport service selection facility and name-to-address mapping. The transport service selection facility offers a standard interface to all networks that exist in an environment.

Name-to-address mapping allows an application to map names to network-specific addresses (see also the Chapter "Network services").

2.1 Background

To place the transport interface in perspective, a discussion of the OSI Reference Model is first presented. The Reference Model partitions networking functions into seven layers, as depicted in Figure 1.

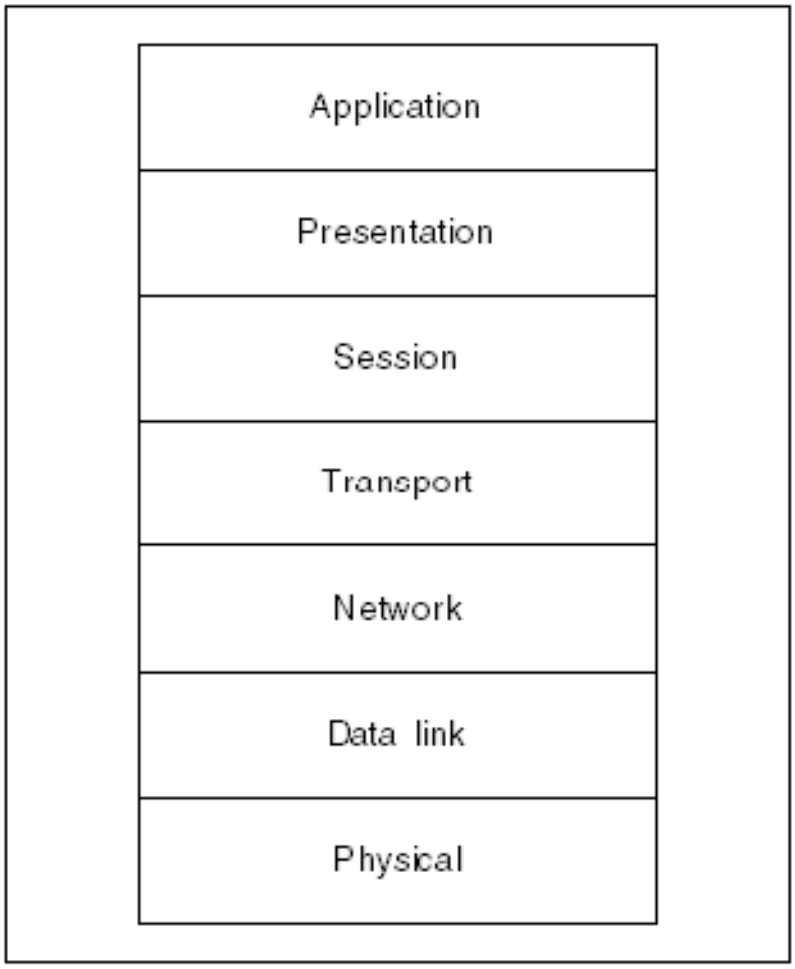


Figure 1: The OSI Reference Model

There is an interface between each layer and the next layer. The interface defines which functions and services a layer provides to the next higher layer. The Transport Level Interface (TLI) is an interface between the transport layer and the session layer.

The transport layer because it is the lowest layer in the reference model that provides an "end-to-end" connection. In the lower layers, the protocols only allow communication with the immediately adjacent machine.

Not until the transport layer is there a connection between the machines that are the actual communication partners, even when there are many other machines between them.

This service is required by applications and the higher layers. The topology and characteristics of the underlying network are hidden from its users. More importantly, the transport layer defines a large number of services that are common to many of today's protocols.

ISO therefore developed standards for the transport layer. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) standards used in the global Internet network have acquired great importance. Standards for the transport layer also define the Internet Transport Protocol (ITP) of Xerox Network Systems (XNS) and Transmission Control of Systems Network Architecture (SNA).

TLI allows applications and higher layer protocols to be implemented without knowledge of the underlying

protocol suite. TLI was modeled after the industry standard ISO Transport Service Definition (ISO 8072).

TLI is implemented as a user library using the STREAMS input/output mechanism. Therefore, many services available to STREAMS applications are also available to users of TLI.

In the ISO reference model, the communication between the layers is described in terms of entities in the layers involved exchanging messages. The entity in the upper layer is called the "user", and the entity in the lower layer is called the "provider".

The transport provider is the entity that provides the services of the transport interface, and the transport user is the entity that requires these services.

An example of a transport provider is the ISO transport protocol, while a transport "user" may be a networking application or session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

The TLI functions enable a user to make requests to the provider and process incoming events.

The transport interface provides two types of service: the connection-oriented and the connectionless service. The connection-oriented service is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. Connectionless mode, by contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units.

Connectionless service is attractive for applications that

- involve short-term request/response interactions,
- are dynamically reconfigurable,
- do not require guaranteed, in-sequence delivery of data.

2.2 Connection-oriented mode

The connection-oriented transport service is characterized by four phases:

- local management
- connection establishment
- data transfer
- connection release

Local management

The local management phase defines functions between the transport user, the transport provider and other instances that control connection establishment. For example, the user must establish a communication channel to the transport provider. Each channel between a transport user and transport provider is called the "transport endpoint". The *t_open* routine enables a user to choose a particular transport provider.

Another necessary local function is the identification of the transport endpoint to the transport provider. Each transport endpoint is identified by a transport address. More accurately, a transport address is bound to a transport endpoint, and one user process may manage several transport endpoints. In connection-oriented service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address may be a simple character string (for example "peter"), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses may be assigned to each transport endpoint by *t_bind*.

In addition to *t_open* and *t_bind*, several routines are available to support local operations. All local management routines of the transport interface are summarized in Table 1 below.

Command	Description
t_alloc	Allocates transport interface data structures
t_bind	Binds a transport address to a transport endpoint

t_close	Closes a transport endpoint
t_error	Prints a transport interface error message
t_free	Frees structures allocated using *t_alloc*
t_getinfo	Returns a set of parameters associated with the current transport provider
t_getstate	Returns the state of a transport endpoint
t_look	Returns the current event on a transport endpoint
t_open	Establishes a transport endpoint that is bound to a specific transport provider
t_optmgmt	Negotiates protocol-specific options with the transport provider
t_sync	Synchronizes a transport endpoint with the transport provider
t_unbind	Unbinds a transport address from a transport endpoint

Table 1: Procedures for the local administration of the transport interface

Connection establishment

During the connection establishment phase, a communication link or virtual connection is established between two users.

This phase is illustrated by a client-server relationship between two transport users. One user, the server, typically advertises some service to a group of users, and then listens for requests from those users. As each client requires the service, it attempts to connect itself to the server using the server's advertised transport address.

The *t_connect* routine initiates the connection request. One argument for *t_connect*, the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using *t_listen*, and must call *t_accept* to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Table 2 summarizes all routines available for establishing a transport connection.

Command	Description
t_accept	Accepts a request to establish a connection
t_connect	Establishes a connection with the transport user at a specified destination
t_listen	Receives a request from another transport user to establish a connection
t_rcvconnect	Completes connection establishment if *t_connect* was called in asynchronous mode

Table 2: Connection establishment procedures

Data transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, *t_snd* and *t_rcv*, send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent.

Table 3 summarizes the connection mode data transfer routines.

Command	Description
t_rcv	Receives data
t_snd	Sends data

Table 3: Data transfer procedures (connection-oriented)

Connection release

During the connection release phase, existing connections are cleared. When you decide that a connection should end, you can request that the provider release the transport connection. Two types of connection release are supported by the transport interface. The first is an abortive release, which directs the transport provider to release the connection immediately.

Any previously sent data that has not yet reached the other transport user may be discarded by the transport provider. The *t_snddis* routine initiates this abortive disconnect, and *t_rcvdis* processes the incoming indication for an abortive disconnect.

All transport providers must support the abortive release procedure.

In addition, some transport providers may also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions *t_sndrel* and *t_rcvrel* support this. Table 4 indicates the connection release routines.

Command	Description
t_rcvdis	Returns an indication of an aborted connection

t_rcvrel	Returns an indication that the remote user has requested an orderly release of a connection
t_snddis	Aborts a connection or rejects a connection request
t_sndrel	Requests the orderly release of a connection

Table 4: Connection release procedures

Connection-oriented client/server concept

As discussed in the previous section, the connection-oriented service can be illustrated using a client-server paradigm. The important concepts of connection-oriented service will be presented using two programming examples. The examples are related: the first example illustrates how a client establishes a connection to a server and then communicates with it; the second example shows the server's side of the interaction.

All examples discussed in this chapter are presented complete later in Section "Some examples".

In the examples, the client establishes a connection with a server process. The server then transfers a file to the client. The client, in turn, receives the data from the server and writes it to its standard output file.

Local management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using *t_open*, and establish its identity (or address) using *t_bind*.

The set of services supported by the transport interface may not be implemented by all transport protocols. Each transport provider has a set of characteristics associated with it that determines the services it offers and the limits associated with those services. This information is returned to the user by *t_open*, and consists of the following:

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

Three service types are defined:

T_COTS	The transport provider supports connection-oriented service but does not provide the optional orderly release facility.
T_COTS_ORD	The transport provider supports connection-oriented service with the optional orderly release facility.
T_CLTS	The transport provider supports connectionless service.

t_open returns the default provider characteristics associated with a transport endpoint. However, some characteristics may change after an endpoint has been opened. This will occur if the characteristics are associated with negotiated options (option negotiation is described later in this section). For example, if the support of expedited data transfer is a negotiated option, the value of this characteristic may change. *t_getinfo* may be called to retrieve the current characteristics of a transport endpoint.

Once a user establishes a transport endpoint with the chosen transport provider, it must give the transport provider the address at which it can be reached via this transport endpoint. As mentioned earlier, *t_bind* does this by binding a transport address to the transport endpoint. In addition, for servers, *t_bind* informs the transport provider that the endpoint will be used to listen for incoming connection requests, also called connect indications.

An optional facility, *t_optmgmt*, is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol is expected to define its own set of negotiable protocol options, which may include such information as Quality-of-Service parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The client

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

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#define SRV_ADDR 1    /* server's well known address */
main()
{
int fd;
int nbytes;
int flags = 0;
char buf[1024];
struct t_call *sndcall;
extern int t_errno;
if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
t_error("t_open failed");
exit(1);
}
if (t_bind(fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(2);
}
```

The first argument to *t_open* is the pathname of a file system node that identifies the transport protocol that will supply the transport service. In this example, */dev/ticotsord* is a special file that provides a generic, connection-oriented transport protocol. It is opened by the second argument for read/write accesses. The third argument may be used to return the service characteristics of the transport provider to the user. This information is useful when writing protocol-independent software (discussed in the Section "Guidelines for protocol independence"). For simplicity, the client and server in this example ignore this information and assume the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the T_COTS_ORD service type, and the example uses it for orderly connection release.
- User data may not be passed between users during either connection establishment or abortive release.
- The transport provider does not support protocol-specific options.

Because these characteristics are not needed by the user, NULL is specified in the third argument to *t_open*. If the user needs a service type other than T_COTS_ORD, a different special file must be opened. There is an example of T_CLTS in the Section "Connectionless mode".

The return value of *t_open* s an identifier for the transport endpoint that will be used by all subsequent transport interface function calls. This identifier is actually a file descriptor obtained by opening the transport protocol file. For more information, see the Section "Use of read/write interfaces".

After the transport endpoint is created, the client calls *t_bind* to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address to be bound to the transport endpoint, and the third argument contains on the return of the *t_bind* call the address actually bound.

The address associated with a server's transport endpoint is important, because that is the address used by all clients to access the server. However, the typical client does not care what its own address is, because no other process will try to access it. That is the case in this example, where the second and third arguments to *t_bind* are set to NULL. A NULL second argument directs the transport provider to choose an address for the user. A NULL third argument specifies that the client is not "interested" in the address found by the transport

provider.

If either *t_open* or *t_bind* fail, the program will call *t_error* to print an appropriate error message to stderr. If any transport interface routine fails, the global integer *t_errno* will be assigned a transport error value. A set of error values has been defined in *<tiuser.h>* for the transport interface, and *t_error* will output an error message corresponding to the value in *t_errno*. This routine is analogous to *perror*, which prints an error message based on the value of *errno*. If the error associated with a transport function is a system error, *t_errno* will be set to TSYSERR, and *errno* will be set to the appropriate value.

The server

The server in this example must take similar local management steps before communication can begin. The server must establish a transport endpoint through which it will listen for connection requests.

The necessary definitions and local management steps are shown below:

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#define DISCONNECT  -1
#define SRV_ADDR    1  /* server's well known address */
int conn_fd;        /* connection established here */
extern int t_errno;
main()
{
int listen_fd;    /* listening transport endpoint */
struct t_bind *bind;
struct t_call *call;
if ((listen_fd = t_open("/dev/ticotsord",
O_RDWR, NULL)) < 0) {
t_error("t_open failed for listen_fd.");
exit(1);
}
/*
* By assuming that the address is an integer value,
* this program may not run over another protocol.
*/
if ((bind = (struct t_bind *)t_alloc(listen_fd,
T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind structure failed.");
exit(2);
}
bind->qlen = 1;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
if (t_bind(listen_fd, bind, bind) < 0) {
t_error("t_bind failed for listen_fd.");
exit(3);
}
/*
* Was the correct address bound?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, "t_bind bound wrong address."\n);
exit(4);
}
```

As with the client, the first step is to call *t_open* to establish a transport endpoint with the desired transport provider. This transport endpoint, *listen_fd*, will be used to listen for connection requests.

Next, the server must bind its well-known address to the endpoint. This address is used by each client to access the server. The second argument to *t_bind* requests that a particular address be bound to the transport endpoint. This argument points to a structure with the following format:

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

addr describes the address to be bound, and *qlen* specifies the maximum outstanding connection requests that may arrive at this endpoint.

All transport interface structure and constant definitions are found in *<tiuser.h>*.

The address is specified using a *netbuf* structure that contains the following members:

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

where *buf* points to a buffer containing the data, *len* specifies the bytes of data in the buffer, and *maxlen* specifies the maximum bytes the buffer can hold (and need only be set when data is returned to the user by a transport interface routine). For the *t_bind* structure, the data pointed to by *buf* identifies a transport address. It is expected that the structure of addresses will vary among each protocol implementation under the transport interface. The *netbuf* structure is intended to support any address structure. You will find more information on name-to-address mapping in the Chapter "Network services".

If the value of *qlen* is greater than 0, the transport endpoint may be used to listen for connection requests.

In such cases, *t_bind* directs the transport provider to begin queueing connection requests destined for the bound address immediately. Furthermore, the value of *qlen* specifies the maximum outstanding connection requests the server wishes to process. The server must respond to each connection request, either accepting or rejecting the request for connection. An outstanding connection request is one to which the server has not yet responded. Often, a server will fully process a single connection request and respond to it before receiving the next indication. When this occurs, a value of 1 is appropriate for *qlen*. However, some servers may wish to retrieve several connection requests before responding to any of them. In such cases, *qlen* specifies the maximum number of outstanding indications the server will process. An example of a server that manages multiple outstanding connection requests is presented in the Section "Advanced topics".

t_alloc is called to allocate the *t_bind* structure needed by *t_bind*. *t_alloc* takes three arguments. The first is a file descriptor that references a transport endpoint. The second argument identifies the appropriate transport interface die structure to be allocated. The third argument specifies which, if any, *netbuf* buffers should be allocated for that structure. T_ALL specifies that all *netbuf* buffers associated with the structure should be allocated, and causes the *addr* buffer to be allocated in this example. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The *maxlen* field of this *netbuf* structure will be set to the size of the newly allocated buffer by *t_alloc*. The use of *t_alloc* helps ensure the compatibility of user programs with future releases of the transport interface.

The server in this example processes connection requests one at a time, so *qlen* is set to 1. The address information is then assigned to the newly allocated *t_bind* structure. This *t_bind* structure passes information to *t_bind* in the second argument and returns the actually bound address information to the user in the third argument.

If the provider could not bind the requested address (perhaps because it had been bound to another transport endpoint), it will choose another appropriate address.

Each transport provider manages its address space differently. Some transport providers may allow a single transport address to be bound to several transport endpoints, while others may require a unique address per endpoint. The transport interface supports either choice. Based on its address management rules, a provider will determine if it can bind the requested address. If not, it will choose another valid address from its address space and bind it to the transport endpoint.

The server must check the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients will be unable to reach the server.

If *t_bind* succeeds, the provider will begin queueing connection requests entering the next phase of communication, connection establishment.

Connection establishment

The connection establishment procedures highlight the distinction between clients and servers. The transport

interface imposes a different set of procedures in this phase for each type of transport user. The client starts the connection establishment procedure by requesting a connection to a particular server using *t_connect*. The server is then notified of the client's request by calling *t_listen*. The server may either accept or reject the client's request. It will call *t_accept* to establish the connection, or call *t_snddis* to reject the request. The client will be notified of the server's decision when *t_connect* completes.

The transport interface supports two facilities during connection establishment that may not be supported by all transport providers:

- The ability to transfer data between the client and server when establishing the connection:

  The client may send data to the server when it requests a connection. This data will be passed to the server by *t_listen*. Similarly, the server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by *t_open* determines how much data, if any, two users may transfer during connect establishment.
- The negotiation of protocol options during connection establishment:

  The client may specify protocol options that it would like the transport provider and/or the remote user to support. The transport interface supports both local and remote option negotiation. As discussed earlier, option negotiation is inherently a protocol-specific function. Use of this facility is discouraged if protocol-independent software is a goal (see the Section "Guidelines for protocol independence").

The client

Continuing with the client/server example, the steps needed by the client to establish a connection are shown next:

```
/* By assuming that the address is an integer value,
* this program may not run over another protocol.
*/
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR))
== NULL)
{
t_error("t_alloc failed");
exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;
if (t_connect(fd, sndcall, NULL) < 0) {
t_error("t_connect failed for fd");
exit(4);
}
```

The *t_connect* call establishes the connection with the server. The first argument to *t_connect* identifies the transport endpoint through which the connection is established, and the second argument, which has a *t_call* structure type, identifies the destination server. The third argument is also a pointer to the *t_call*structure, which has the following format:

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

addr identifies the address of the server (you will find more information on name-to-address mapping in the Chapter "Network services"). *opt* may be used to specify protocol-specific options that the client would like to associate with the connection, and *udata* identifies user data that may be sent with the connection request to the server. The *sequence* field has no meaning for *t_connect*.

t_alloc is called in the example to allocate the *t_call* structure dynamically. In this example, no options or user

data are associated with the *t_connect* call, but the server's address must be set. The third argument to *t_alloc* is set to T_ADDR to specify that an appropriate *netbuf* buffer should be allocated for the address. The server's address is then assigned to *buf*, and *len* is set accordingly.

The third argument to *t_connect* can be used to return information about the newly established connection to the user, and may retrieve any user data sent by the server in its response to the connection request. It is set to NULL by the client here to indicate that this information is not needed. The connection will be established on successful return of *t_connect*. If the server rejects the connection request, *t_connect* will fail and set *t_errno* to TLOOK.

Event handling

The TLOOK error has special significance in the transport interface. TLOOK notifies the user if a transport interface routine is interrupted by an unexpected asynchronous transport event on the given transport endpoint. As such, TLOOK does not report an error with a transport interface routine, but the normal processing of that routine will not be done because of the pending event. The events defined by the transport interface are listed here:

T_LISTEN	A connection request, has arrived at the transport endpoint.
T_CONNECT	The confirmation of a previously sent connection request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connection request.
T_DATA	User data has arrived at the transport endpoint.
T_EXDATA	Expedited user data has arrived at the transport endpoint. Expedited data will be discussed later in this section.
T_DISCONNECT	A notification that the connection was aborted or that the server rejected a connection request, called a disconnection request, has arrived at the transport endpoint.
T_ORDREL	A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.
T_UDERR	The notification of an error in a previously sent datagram, called a unit data error indication, has arrived at the transport endpoint.

The *t_look* routine enables a user to determine what event has occurred if a TLOOK error is returned. The user can then process that event accordingly. In the example, if a connection request is rejected, the event passed to the client will be a disconnection request. The client will exit if its request is rejected.

The server

Returning to the example, when the client calls *t_connect*, a connection request will be generated on the server's listening transport endpoint. The steps required by the server to process the event are discussed below. For each client, the server accepts the connection request and spawns a server process to manage the connection.

```
if ((call = □
(struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) == NULL) {
t_error("t_alloc of t_call structure failed");
exit(5);
}
while (1) {
if (t_listen(listen_fd, call) < 0) {
t_error("t_listen failed for listen_fd");
exit(6);
}
if ((conn_fd = accept_call(listen_fd, call)) !=
DISCONNECT)
run_service(listen_fd);
}
```

The server will loop forever, processing each connection request. First, the server calls *t_listen* to retrieve the next connection request. When one arrives, the server calls *accept_call* to accept the connection request. *accept_call* accepts the connection on an alternative transport endpoint (as discussed below) and returns the associated file descriptor. *conn_fd* is a global variable that identifies the transport endpoint where the connection is established. Because the connection is established on an alternate endpoint, the server may continue listening for connection requests on the endpoint that was bound for listening. If the request is accepted without error, *run_service* will spawn a process to manage the connection.

The server allocates a *t_call* structure to be used by *t_listen*. The third argument to *t_alloc*, T_ALL, specifies that all necessary buffers should be allocated for retrieving the caller's address, options, and user data. As mentioned earlier, the transport provider in this example does not support the transfer of user data during connection establishment, and also does not support any protocol options. Therefore, *t_alloc* will not allocate buffers for the user data and options. It must, however, allocate a buffer large enough to store the address of the caller. The *maxlen* field of each *netbuf* structure will be set to the size of the newly allocated buffer by *t_alloc* (*maxlen* is 0 for the user data and options buffers).

Using the *t_call* structure, the server calls *t_listen* to process the next connection request. If there is no connection request waiting, *t_listen* blocks the process until one arrives.

The transport interface supports an asynchronous mode for these routines, which prevents a process from blocking. This feature is discussed in the Section "Advanced topics".

When a connection request arrives, the server calls *accept_call* to accept the client's request, as follows:

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
int resfd;
if ((resfd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
t_error("t_open for accept failed");
exit(7);
}
if (t_bind(resfd, NULL, NULL) < 0) {
t_error("t_bind for accept failed");
exit(8);
}
```

```
if (t_accept(listen_fd, resfd, call) < 0) {
if (t_errno == TLOOK) {  /* must be a disconnect */
if (t_rcvdis(listen_fd, NULL) < 0) {
t_error("t_rcvdis failed for listen_fd");
exit(9);
}
if (t_close(resfd) < 0) {
t_error("t_close failed for responding fd");
exit(10);
}
/* go back up and listen for other calls */
return(DISCONNECT);
}
t_error("t_accept failed");
exit(11);
}
return(resfd);
}
```

accept_call takes two arguments:

- *listen_fd* identifies the transport endpoint where the connection request arrived.
- *call* is a pointer to a *t_call* structure that contains all information associated with the connection request.

The server first establishes another transport endpoint by opening the clone device node of the transport provider and binding an address. As with the client, a NULL value is passed to *t_bind* to specify that the user does not care what address is bound by the provider. The newly established transport endpoint, *resfd*, is used to accept the client's connection request.

The first two arguments of *t_accept* specify the listening transport endpoint and the endpoint where the connection will be accepted, respectively. A connection may be accepted on the listening endpoint, but this prevents other clients from accessing the server for the duration of the connection.

The third argument of *t_accept* points to the *t_call* structure associated with the connection request. This structure should contain the address of the calling user and the sequence number returned by *t_listen*. The value of *sequence* is significant if the server manages multiple outstanding connection requests. The Section "Advanced topics" contains an example of this.

Also, the *t_call* structure should identify protocol options the user would like to specify, and user data that may be passed to the client. Because the transport provider in this example does not support protocol options or the transfer of user data during connection establishment, the *t_call* structure returned by *t_listen* may be passed without change to *t_accept*.

For simplicity in the example, the server will exit if either the *t_open* or *t_bind* call fails. *exit(2)* will close the transport endpoint associated with *listen_fd*, causing the transport provider to pass a disconnection request to the client that requested the connection. This disconnection request notifies the client that the connection was not established; *t_connect* will fail, setting *t_errno* to TLOOK.

t_accept may fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted, and *t_errno* will be set to TLOOK. The state transition table in the Section "State transitions" shows that the only event that may occur in this state with only one outstanding connection request is a disconnection request. This event may occur if the client decides to undo the connection request it had previously sent. If a disconnection request arrives, the server must retrieve the disconnection request using *t_rcvdis*.

This routine takes a pointer to a *t_discon* structure as an argument, which is used to retrieve information associated with a disconnection request. In this example, however, the server does not care to retrieve this information, so it sets the argument to NULL. After receiving the disconnection request, *accept_call* closes the responding transport endpoint and returns DISCONNECT, which informs the server that the connection was disconnected by the client. The server then listens for further connection requests.

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

Data transfer

Once the connection has been established, both the client and server may begin transferring data over the connection using *t_snd* and *t_rcv*. The transport interface does not differentiate the client from the server from this point on. Either user may send and receive data, or release the connection. The transport interface guarantees reliable, correctly sequenced data transfer over an existing connection.

Two classes of data may be transferred over a transport connection:

- normal data
- expedited data

Expedited data is typically associated with urgent information. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of an expedited data class (see *t_open(3N)* in the "Networking Reference Manual (Reliant UNIX)").

All transport protocols support the transfer of data in byte stream mode, where "byte stream" implies no concept of message boundaries on data that are transferred over a connection.

However, some transport protocols support the preservation of message boundaries over a transport

connection. This service is supported by the transport interface, but protocol-independent software must not rely on its existence.

The message interface for data transfer is supported by a special flag of *t_snd* and *t_rcv* called T_MORE. The messages, called Transport Service Data Units (TSDU), may be transferred between two transport users as distinct units.

The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from *t_open* and/or *t_getinfo*. Because the maximum TSDU size can be large (possibly unlimited), the transport interface allows a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the T_MORE flag on every *t_snd* call except the last. This flag specifies that the user will send more data associated with the message in a subsequent call to *t_snd*. The last message unit should be transmitted with T_MORE turned off to specify that this is the end of the TSDU.

Similarly, a TSDU may be passed in multiple units to the receiving user. Again, if *t_rcv* returns with the T_MORE flag set, the user should continue calling *t_rcv* to retrieve the remainder of the message. The last unit in the message will be identified by a call to *t_rcv* that does not set T_MORE.

The T_MORE flag implies nothing about how the data may be packaged below the transport interface or how the data may be delivered to the remote user. Each transport protocol, and each implementation of that protocol, may package and deliver the data differently.

For example, if a user sends a complete message in a single call to *t_snd*, there is no guarantee that the data will arrive at the recipient as a single unit. Similarly, a TSDU transmitted in two message units may be delivered in a single unit to the remote transport user.

The message boundaries may only be preserved by noting the value of the T_MORE flag on *t_snd* and *t_rcv*. This will guarantee that the receiving user will see a message with the same contents and message boundaries as was sent by the remote user.

The client

Continuing with the client/server example, the server will transfer a log file to the client over the transport connection. The client receives this data and writes it to its standard output file. A byte stream interface is used by the client and server, where message boundaries (that is, the T_MORE flag) are ignored. The client receives data using the following instructions:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1)
if ((int)fwrite(buf, 1, nbytes, stdout) < 0) {
fprintf(stderr, "fwrite failed \n");
exit(5);
}
```

The client continuously calls *t_rcv* to process incoming data. If no data is currently available, *t_rcv* blocks until data arrives. *t_rcv* retrieves the available data up to 1024 bytes, which is the size of the client's input buffer, and returns the number of bytes received. The client then writes this data to standard output and continues. The data transfer phase will complete when *t_rcv* fails. *t_rcv* will fail if an orderly release or disconnection request arrives, as discussed later in this section.

If the *fwrite* call fails for any reason, the client will exit, closing the transport endpoint. If the transport endpoint is closed (either by *exit* or *t_close*) during the data transfer phase, the connection will be aborted and the remote user will receive a disconnection request.

The server

Looking now at the other side of the connection, the server manages its data transfer by spawning a child process to send the data to the client. The parent process then loops back to listen for further connection requests.

run_service is called by the server to spawn this child process as follows:

```
connrelease()
{
```

```
/* conn_fd is global because needed here */
if (t_look(conn_fd) == T_DISCONNECT) {
fprintf(stderr, "connection aborted\n");
exit(12);
}
/* else orderly release indication - normal exit
*/
exit(0);
}
run_service(listen_fd)
int listen_fd;
{
int nbytes;
FILE *logfp;        /* file pointer to log file */
char buf[1024];
switch (fork()) {
case -1:
perror("fork failed");
exit(20);
break;
```

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

After the *fork*, the parent process returns to the main processing loop and listens for further connection requests.

Meanwhile, the child process will manage the newly established transport connection. If the *fork* call fails, *exit* closes the transport endpoint associated with *listen_fd*, sending a disconnection request to the client, and the client's *t_connect* call will fail.

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

- T_EXPEDITED specifies that the data is expedited
- T_MORE defines the message boundaries

Neither flag is set by the server in this example.

If the user floods the transport provider with data, the provider may exert back pressure to provide flow control. In such cases, *t_snd* will block until the flow control is relieved, and will then resume its operation. *t_snd* will not complete until *nbyte* bytes have been passed to the transport provider.

The *t_snd* routine does not look for a disconnection request (showing that the connection was broken) before passing data to the provider. Also, because the data traffic flows in one direction, the user will never look for incoming events. If the connection is aborted, the user should be notified since data may be lost. The user can invoke *t_look*, which checks for incoming events before each *t_snd* call. A more efficient solution is presented in the example. The STREAMS I_SETSIG *ioctl* enables a user to request an immediate signal when a given event occurs. S_INPUT causes a signal to be sent to the user if any input arrives on the stream referenced by

conn_fd. If a disconnection request arrives, the signal catching routine (*connrelease*) prints an error message and then exits.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated *t_snd* and *t_rcv* calls, it could rely on *t_rcv* to recognize an incoming disconnection request.

Connection release

At any point during data transfer, either user may release the transport connection and end the conversation. As mentioned earlier, two forms of connection release are supported by the transport interface:

- Abortive release breaks a connection immediately and may result in the loss of any data that has not yet reached the destination user.

  Either user may call *t_snddis* to generate an abortive release. Also, the transport provider may abort a connection if a problem occurs below the transport interface.

  When the remote user is notified of the aborted connection, *t_rcvdis* must be called to retrieve the disconnection request. This call returns a reason code that identifies why the connection was aborted. This reason code is specific to the underlying transport protocol, and should not be interpreted by protocol-independent software.
- Orderly release gracefully terminates a connection and guarantees that no data will be lost.

All transport providers must support the abortive release procedure, but orderly release is an optional facility that is not supported by all transport protocols.

The server

The client-server example in this section assumes that the transport provider supports the orderly release of a connection.

When all the data has been transferred by the server, the connection may be released as follows:

```
if (t_sndrel(conn_fd) < 0) {
t_error("t_sndrel failed");
exit(27);
}
pause(); /* until orderly release indication arrives */
}
}
```

The orderly release procedure consists of two steps by each user. The first user who wishes to release the connection uses *t_sndrel* to send a release request (see example above). This routine informs the partner that no more data will be sent by the server. When the partner receives this indication, it may continue sending data back to the server if desired. When all data have been transferred, however, the client must also call *t_sndrel* to indicate that it is ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

In this example, data is transferred in one direction from the server to the client, so the server does not expect to receive data from the client after it has initiated the release procedure. Thus, the server simply calls *pause* after initiating the release. Eventually, the remote user responds with its orderly release request, which generates a signal that will be caught by *connrelease*. Remember that the server earlier issued an I_SETSIG *ioctl* call to generate a signal on any incoming event. Since the only possible transport interface events that can occur in this situation are a disconnection request or orderly release indication, *connrelease* terminates normally when the orderly release indication arrives.

The *exit* call in *connrelease* will close the transport endpoint, freeing the bound address for another user. If a user process wants to close a transport endpoint without exiting, it may call *t_close*.

The client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until *t_rcv* fails. If the server releases the connection (using either *t_snddis*

or *t_sndrel*), *t_rcv* will fail and set *t_errno* to TLOOK. The client then processes the connection release as follows:

```
if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
if (t_rcvrel(fd) < 0) {
t_error("t_rcvrel failed");
exit(6);
}
if (t_sndrel(fd) < 0) {
t_error("t_sndrel failed");
exit(7);
}
exit(0);
}
t_error("t_rcv failed");
exit(8);
}
```

When an event occurs on the client's transport endpoint, the client checks whether the expected orderly release indication has arrived. If so, it proceeds with the release procedures by calling *t_rcvrel* to process the request and *t_sndrel* to inform the server that it is also ready to release the connection. At this point the client exits, closing its transport endpoint.

Because not all transport providers support the orderly release facility just described, users may have to use the abortive release facility provided by *t_snddis* and *t_rcvdis*. However, steps must be taken by each user to prevent data loss. For example, a special byte pattern may be inserted in the data stream to indicate the end of a conversation. There are many ways to prevent data loss. Each application and high level protocol must choose an appropriate routine given the target protocol environment and requirements.

2.3 Connectionless mode

In a connectionless transport service, a distinction is drawn between local management and data transfer. In local management, the same functions are required as in a connection-oriented service.

Data transfer allows the user to transfer data units (datagrams) to another user. Each data unit must be accompanied by the full destination address. This message-based data exchange is supported by two procedures, *t_sndudata* and *t_rcvudata*. Table 5 shows all the procedures required for connectionless data transfer.

Command	Description
t_rcvudata	Receives a message sent by another user
t_rcvuderr	Provides error information on a previously sent message
t_sndudata	Sends a message to a specified user

Table 5: Data transfer procedures (connectionless)

Connectionless services are appropriate for short-term request/response dialogs (request systems, for example). Data are transferred in self-contained units with no logical relationship required among multiple units.

The connectionless mode will be described using a request system as an example. This server waits for incoming requests and processes and responds to each one.

Local management

Just as in connection-oriented mode, the transport users must carry out appropriate local management steps before transferring data. A user must choose the appropriate connectionless service provider using *t_open* and establish its identity using *t_bind*.

t_optmgmt may be used to negotiate protocol options associated with the transfer of each data unit. As with the connection-oriented service, each transport provider specifies the options, if any, that it supports. Option negotiation is therefore a protocol-specific activity.

In the example, the definitions and local management calls needed by the transaction server are as follows:

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>
#define SRV_ADDR    2       /* server's well known address */
main()
{
int fd;
int flags;
struct t_bind *bind;
struct t_unitdata *ud;
struct t_uderr *uderr;
extern int t_errno;
if ((fd = t_open("/dev/ticlts", O_RDWR, NULL)) < 0) {
t_error("unable to open /dev/provider");
exit(1);
}
if ((bind = (struct t_bind *)t_alloc(fd,
T_BIND, T_ADDR)) == NULL) {
t_error("t_alloc of t_bind structure failed");
exit(2);
}
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
bind->qlen = 0;
if (t_bind(fd, bind, bind) < 0) {
t_error("t_bind failed");
exit(3);
}
/*
* is the bound address correct?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, "t_bind bound wrong address\n");
exit(4);
}
```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using *t_open*.

Each provider has an associated service type, so the user may choose a particular service by opening the appropriate transport provider file. This connectionless server ignores the characteristics of the provider returned by *t_open* in the same way as the users in the connection-oriented example, by setting the third argument to NULL. For simplicity, the transaction server assumes the transport provider has the following characteristics:

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

The connectionless server also binds a transport address to the endpoint so that potential clients may identify and access the server. A *t_bind* structure is allocated using *t_alloc* and the *buf* and *len* fields of the address are set accordingly.

One important difference between the connection-oriented server and this connectionless server is that the *qlen* field of the *t_bind* structure has no meaning for connectionless service, since all users are capable of

receiving datagrams once they have bound an address. The transport interface defines an inherent client-server relationship between two users while establishing a transport connection in the connection-oriented service. However, no such relationship exists in the connectionless service. It is the context of this example, not the transport interface, that defines one user as a server and another as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by *t_bind* to ensure it is correct.

Data transfer

Once a user has bound an address to the transport endpoint, datagrams may be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, the transport interface enables a user to specify protocol options that should be associated with the transfer of the data unit. As discussed earlier, each transport provider defines the set of options, if any, that may accompany a datagram.

The following sequence of calls illustrates the data transfer phase of the connectionless server:

```
if ((ud = (struct t_unitdata *)t_alloc(fd,
T_UNITDATA, T_ALL)) == NULL) {
t_error("t_alloc of t_unitdata structure failed");
exit(5);
}
```

```
if ((uderr = (struct t_uderr *)t_alloc(fd,
T_UDERROR, T_ALL)) == NULL) {
t_error("t_alloc of t_uderr structure failed");
exit(6);
}
for(;;) {
if (t_rcvudata(fd, ud, &flags) < 0) {
if (t_errno == TLOOK) {
/*
*  Error on previously sent datagram
*/
if (t_rcvuderr(fd, uderr) < 0) {
t_error("t_rcverr failed");
exit(7);
}
fprintf(stderr,
"bad datagram, error = %d \n",
uderr->error);
continue;
}
t_error("t_rcvudata failed");
exit(8);
}
/*
* query() processes the request and places the
* response in ud->udata.buf and the length in
* ud->udata.len
*/
query(ud);
if (t_sndudata(fd, ud, 0) < 0) {
t_error("t_sndudata failed");
exit(9);
}
}
}
query()
{
/* Merely a stub for simplicity */
}
```

The server must first allocate a *t_unitdata* structure for storing datagrams, which has the following format:

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

addr holds the source address of incoming datagrams and the destination address of outgoing datagrams, *opt* identifies any protocol options associated with the transfer of the datagram, and *udata* holds the data itself. The *addr*, *buf* and *udata* fields must all be allocated with buffers large enough to hold any possible incoming values. As described in the previous section, the T_ALL argument to *t_alloc* will ensure this and will set the *maxlen* field of each *netbuf* structure accordingly. Because the transport provider does not support protocol options in this example, no options buffer will be allocated, and *maxlen* will be set to zero in the *netbuf* structure for options. The server also allocates a *t_uderr* structure for processing any datagram errors, as discussed later in this section.

The transaction server traverses a continuous loop, receiving requests, processing the requests, and responding to the clients. It first calls *t_rcvudata* to receive the next request. *t_rcvudata* will retrieve the next available incoming datagram. If none is currently available, *t_rcvudata* will block, waiting for a datagram to arrive. The second argument of *t_rcvudata* identifies the *t_unitdata* structure in which the datagram should be stored.

The third argument, *flags*, must point to an integer variable and may be set to T_MORE on return from *t_rcvudata* to specify that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to *t_rcvudata* will retrieve the remainder of the datagram. Because *t_alloc* allocates a *udata* buffer large enough to store the maximum datagram size, the transaction server does not have to check the value of *flags*.

If a datagram is received successfully, the transaction server calls the *query* routine to process the request.

Datagram errors

If the transport provider cannot process a datagram that was passed to it by *t_sndudata*, it will return a unit data error event, T_UDERR, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what may be wrong with the datagram. The reason a datagram could not be processed is protocol-specific. One reason may be that the could not interpret the destination address or options.

Each transport protocol is expected to specify all reasons why it is unable to process a datagram.

The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication will be used. Remember, the connectionless service does not guarantee reliable delivery of data.

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

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

where *addr* and *opt* identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that specifies why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

2.4 Use of read/write interfaces

A user may wish to establish a transport connection and then *exec* an existing user program such as *cat*. These programs use *read* and *write* for their input/output needs. The transport interface does not directly support a read/write interface to a transport provider, but one is available with UNIX System V. This interface enables a user to issue *read* and *write* calls over a transport connection that is in the data transfer phase. This section describes the read/write interface to the connection-oriented service of the transport interface. This interface is not available with the connectionless service.

The *read*/*write* interface is presented using the slightly modified client example of the Section "Connection-oriented mode". The clients are identical until the data transfer phase is reached. At that point, this client will use the *read*/*write* interface and *cat* to process incoming data. *cat* can be run without change over the transport connection. Only the differences between this client and that of the example in the section entitled "Connection-oriented mode" are shown below:

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

The client invokes the *read/write* interface by pushing the *tirdwr* module onto the STREAM associated with the transport endpoint where the connection was established. This module converts the transport interface above the transport provider into a pure *read/write* interface. With the module in place, the client calls *close* and *dup* to establish the transport endpoint as its standard input file, and uses */usr/bin/cat* to process the input. Because the transport endpoint identifier is a file descriptor, the *dup* facility for duping the endpoint is available to users.

Because the transport interface uses STREAMS, the facilities of this character input/output mechanism can be used to provide enhanced user services. By pushing the *tirdwr* module above the transport provider, the user's interface is effectively changed. The semantics of *read* and *write* must be followed, and message boundaries will not be preserved.

The *tirdwr* module may only be pushed onto a stream when the transport endpoint is in the data transfer phase. Once the module is pushed, the user may not call any transport interface routines. If a transport interface routine is invoked, *tirdwr* will generate a fatal protocol error, EPROTO, on that stream, rendering it unusable. Furthermore, if the user pops the *tirdwr* module off the stream (see I_POP in *streamio*), the transport connection will be aborted.

The exact semantics of *write*, *read*, and *close* using *tirdwr* are described below. To summarize, *tirdwr* enables a user to send and receive data over a transport connection using *read* and *write*. This module will translate all transport interface indications into the appropriate actions. The connection can be released with the *close* system call.

2.4.1 write

The user may transmit data over the transport connection using *write*. The *tirdwr* module will pass data through to the transport provider. However, if a user attempts to send a zero-length data packet, which the STREAMS mechanism allows, *tirdwr* will discard the message. If the transport connection is aborted (for example, because the remote user aborts the connection using *t_snddis*), further *write* calls will fail and set *errno* to ENXIO. The user can still retrieve any available data after a hangup.

2.4.2 read

read may be used to retrieve data that has arrived over the transport connection. The *tirdwr* module will pass data through to the user from the transport provider. However, any other event or indication passed to the user from the provider will be processed by *tirdwr* as follows:

- *read* cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, *tirdwr* will generate a fatal protocol error, EPROTO, on that stream. This error causes further system calls to fail. You should therefore not communicate with a process that is sending expedited data.

- If an abortive disconnection request is received, *tirdwr* will ignore it and generate a STREAMS hangup condition (disconnect) on that stream. Subsequent *read* calls will retrieve any remaining data, and then *read* will return zero for all further calls (indicating end-of-file).
- If an orderly disconnection request is received, *tirdwr* will ignore it and deliver a zero-length STREAMS message to the user. As described in *read*, this notifies the user of end-of-file.
- If any other transport interface indication is received, *tirdwr* generates a fatal protocol error, EPROTO, on that stream. This causes further system calls to fail.

2.4.3 close

With *tirdwr* on a STREAM, the user can send and receive data over a transport connection for the duration of that connection. Either user may terminate the connection by closing the STREAM associated with the transport endpoint or by popping the *tirdwr* module off the STREAM. In either case, *tirdwr* will take the following actions:

- If an orderly disconnection request was previously received by *tirdwr*, an orderly release request will also be passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure will receive the expected indication when data transfer completes.
- If an abortive disconnection request was previously received by *tirdwr*, no special action is taken.
- If neither an orderly disconnection request nor an abortive disconnection request was previously received by *tirdwr*, an abortive disconnection request will be passed to the transport provider to abort the connection.
- If an error previously occurred on the stream and an abortive disconnection request has not been received by *tirdwr*, an abortive disconnection request will be passed to the transport provider.

A process may not initiate an orderly release after *tirdwr* is pushed onto a stream, but *tirdwr* will handle a request properly if it is initiated by the user on the other side of a transport connection. If the client in this section is communicating with the server program in the Section "Connection-oriented mode", that server will terminate the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. As explained in the first list item above, when the file descriptor is closed, *tirdwr* will initiate the orderly release request from the client's side of the connection. This will generate the indication that the server is expecting, and the connection will be released properly.

2.5 Advanced topics

This section presents the following important concepts of the transport interface that have not been covered in the previous section:

- an optional non-blocking (asynchronous) mode for some library calls
- an advanced programming example that defines a server supporting multiple outstanding connection requests and operating in an event-driven manner

2.5.1 Asynchronous execution mode

Many transport interface library routines may block waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, an application may wish to do local processing while waiting for some asynchronous transport interface event.

Support for asynchronous processing of transport interface events is available to applications using a combination of the STREAMS asynchronous features and the non-blocking mode of the transport interface library routines. Earlier examples in this guide have illustrated the use of the *poll* system call and the I_SETSIG *ioctl* command for processing events asynchronously.

In addition, transport interface routines that may block waiting for some event can be run in a special non-blocking mode. For example, *t_listen* will normally block, waiting for a connection request. However, a server can periodically poll a transport endpoint for existing connection requests by calling *t_listen* in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting O_NDELAY or O_NONBLOCK on the file descriptor. These can be set as a flag on *t_open*, or by calling *fcntl* before calling the transport interface routine. *fcntl* can be used to enable or disable this mode at any time. All programming examples in this chapter use the default synchronous processing mode.

O_NDELAY or O_NONBLOCK affect each transport interface routine differently. To determine the exact semantics of O_NDELAY or O_NONBLOCK for a particular routine, see the relevant pages in the Chapter "XDR/RPC protocol specification" (...).

2.5.2 Programming example

The following example demonstrates two important concepts. The first is a server's ability to manage multiple outstanding connection requests. The second is an illustration of the ability to write event-driven software using the transport interface and the STREAMS system call interface.

The server example in the Section "Connection-oriented mode" is capable of supporting only one outstanding connection request, but the transport interface supports the ability to manage multiple outstanding connection requests. One reason a server might wish to receive several simultaneous connection requests is to impose a priority scheme on each client. A server may retrieve several connection requests, and then accept them in an order based on a priority associated with each client. A second reason for handling several outstanding connection requests is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connection request, other clients will find it busy. If, however, multiple connection requests can be processed simultaneously, the server will be found to be busy only if the maximum allowed number of clients attempt to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming transport interface events, and then takes the appropriate actions for the current event.

The definitions and local management functions needed by this example are similar to those of the server example in the Section "Connection-oriented mode".

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

```
#define NUM_FDS      1
#define MAX_CONN_IND 4
#define SRV_ADDR     1        /* server's well known address */
int conn_fd;          /* server connection here */
/* holds connection requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];
extern int t_errno;
```

```
main()
{
struct pollfd pollfds[NUM_FDS];
struct t_bind *bind;
int i;
/*
* Only opening and binding one transport endpoint,
* but more could be supported.
*/
if ((pollfds[0].fd = t_open("/dev/ticotsord",
O_RDWR, NULL)) < 0) {
t_error("t_open failed");
exit(1);
}
if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind structure failed");
exit(2);
}
bind->qlen = MAX_CONN_IND;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
if (t_bind(pollfds[0].fd, bind, bind) < 0) {
t_error("t_bind failed");
exit(3);
}
/*
* Was the correct address bound?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, "t_bind bound wrong address\n");
exit(4);
}
```

The file descriptor returned by *t_open* is stored in a *pollfd* structure (see *poll*). This is subsequently used with a *poll* call to process incoming events. Note that only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to the above code.

An important aspect of this server is that it sets *qlen* to a value greater than 1 for *t_bind*. This specifies that the server is willing to handle multiple outstanding connection requests. Remember that the earlier examples single-threaded the connection requests and responses. The server would accept the current connection request before retrieving additional connection requests. This example, however, can accept up to MAX_CONN_IND connection requests simultaneously. The transport provider may negotiate the value of *qlen* downward if it cannot support MAX_CONN_IND outstanding connection requests.

Once the server has bound its address and is ready to process incoming connection requests, it does the following:

```
pollfds[0].events = POLLIN;
for(;;) {
if (poll(pollfds, NUM_FDS, -1) < 0) {
perror("poll failed");
exit(5);
}
for (i = 0; i < NUM_FDS; i++) {
```

```
switch (pollfds[i].revents) {
default:
perror("poll returned error event");
exit(6);
case 0:
continue;
case POLLIN:
do_event(i, pollfds[i].fd);
service_conn_ind(i, pollfds[i].fd);
}
}
}
}
```

The *events* field of the *pollfd* structure is set to POLLIN, which will notify the server of any incoming transport interface events. The server then enters an infinite loop, in which it *poll*s the transport endpoint(s) for events, and then processes those events as they occur.

The *poll* call will block indefinitely, waiting for an incoming event. On return, each entry (corresponding to each transport endpoint) is checked for an event. If *revents* is set to 0, no event has occurred on that endpoint. In this case, the server continues to the next transport endpoint. If *revents* is set to POLLIN, an event does exist on the endpoint. In this case, *do_event* is called to process the event. If *revents* contains any other value, an error must have occurred on the transport endpoint, and the server will exit.

For each iteration of the loop, if any event is found on the transport endpoint, *service_conn_ind* is called to process any outstanding connection requests. However, if another connection request is pending, *service_conn_ind* will save the current connection request and respond to it later. This routine will be explained shortly.

If an incoming event is discovered, the following routine is called to process it:

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

```
if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
T_CALL, T_ALL)) == NULL) {
t_error("t_alloc of t_call structure failed");
exit(11);
}
if (t_listen(fd, calls[slot][i]) < 0) {
t_error("t_listen failed");
exit(12);
}
break;
case T_DISCONNECT:
discon = (struct t_discon *)t_alloc(fd,
T_DIS, T_ALL);
if (t_rcvdis(fd, discon) < 0) {
t_error("t_rcvdis failed");
exit(13);
}
```

```
/*
* find call ind in array and delete it
*/
for (i = 0; i < MAX_CONN_IND; i++) {
if (discon->sequence ==
calls[slot][i]->sequence) {
t_free(calls[slot][i], T_CALL);
calls[slot][i] = NULL;
}
}
t_free(discon, T_DIS);
break;
}
}
```

This routine takes a number, *slot*, and a file descriptor, *fd*, as arguments. *slot* is used as an index into the global array *calls*. This array contains an entry for each polled transport endpoint, where each entry consists of an array of *t_call* structures that hold incoming connection requests for that transport endpoint. The value of *slot* is used to identify the transport endpoint.

do_event calls *t_look* to determine the transport interface event that has occurred on the transport endpoint specified by *fd*. If a connection request (T_LISTEN event) or disconnection request (T_DISCONNECT) has arrived, the event is processed. Otherwise, the server prints an appropriate error message and exits.

For connection requests, *do_event* scans the array of outstanding connection requests looking for the first free entry. A *t_call* structure is then allocated for that entry, and the connection request is retrieved using *t_listen*. There must always be at least one free entry in the connection request array, because the array is large enough to hold the maximum number of outstanding connection requests as negotiated by *t_bind*. The processing of the connection request is deferred until later.

If a disconnection request arrives, it must belong to a previously received connection request. This occurs if a client attempts to undo a previous connection request. In this case, *do_event* allocates a *t_discon* structure to retrieve the relevant disconnect information. This structure has the following members:

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

where *udata* identifies any user data that might have been sent with the disconnection request, *reason* contains a protocol-specific disconnect reason code, and *sequence* identifies the outstanding connection request that matches this disconnection request.

Next, *t_rcvdis* is called to retrieve the disconnection request. The array of connection requests for *slot* is then scanned for one that contains a sequence number that matches the *sequence* number in the disconnection request. When the connection request is found, it is freed and the corresponding entry is set to NULL.

As mentioned earlier, if any event is found on a transport endpoint, *service_conn_ind* is called to process all currently outstanding connection requests associated with that endpoint as follows:

```
service_conn_ind(slot, fd)
{
int i;
for (i = 0; i < MAX_CONN_IND; i++) {
if (calls[slot][i] == NULL)
continue;
if ((conn_fd = t_open("/dev/ticotsord", O_RDWR, NULL))
< 0) {
t_error("open failed");
```

```
exit(14);
}
if (t_bind(conn_fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(15);
}
if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
if (t_errno == TLOOK) {
t_close(conn_fd);
return;
}
t_error("t_accept failed");
exit(16);
}
t_free(calls[slot][i], T_CALL);
calls[slot][i] = NULL;
run_service(fd);
}
}
```

For the given transport endpoint (*slot*), the array of outstanding connection requests is scanned. For each indication, the server will open a responding transport endpoint, bind an address to the endpoint, and then accept the connection on that endpoint. If another event (connection request or disconnection request) arrives before the current indication is accepted, *t_accept* will fail and set *t_errno* to TLOOK.

The user cannot accept an outstanding connection request if any pending connection request events or disconnection request events exist on that transport endpoint.

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

Eventually, all events will be processed, and *service_conn_ind* will be able to accept each connection request in turn. Once the connection has been established, the *run_service* routine used by the server in the Section "Connection-oriented mode" is called to manage the data transfer.

2.6 State transitions

These tables describe all state transitions associated with the transport interface. First, however, the states and events will be described.

2.6.1 Transport interface states

The table below defines the states used to describe the transport interface state transitions.

State	Meaning	Service type
T_UNINIT	uninitialized - initial and final state of interface	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	initialized but not bound	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	no connection established	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	outgoing connection pending for client	T_COTS, T_COTS_ORD
T_INCON	incoming connection pending for server	T_COTS, T_COTS_ORD
T_DATAXFER	data transfer	T_COTS, T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	

Table 6: States of the transport interface

2.6.2 Outgoing events

The outgoing events described in the next table correspond to the return of the specified transport routines, where these routines send a request or response to the transport provider.

In the table, some events (such as *acceptN*) are distinguished by the context in which they occur. The context is based on the values of the following variables:

ocnt Count of outstanding connection requests

fd File descriptor of the current transport endpoint

refsd File descriptor of the transport endpoint where a connection will be accepted

Event	Meaning	Service type
opened	successful return of *t_open*	T_COTS, T_COTS_ORD, T_CLTS
bind	successful return of *t_bind*	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	successful return of *t_optmgmt*	T_COTS_ORD, T_CLTS
unbind	successful return of *t_unbind*	T_COTS, T_COTS_ORD, T_CLTS
closed	successful return of *t_closed*	T_COTS, T_COTS_ORD, T_CLTS
connect1	successful return of *t_connect* in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on *t_connect* in asynchronous mode, or TLOOK error due to a disconnection request arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	successful return of *t_accept* with *ocnt* == 1, *fd* = resfd	T_COTS, T_COTS_ORD
accept2	successful return of *t_accept* with *ocnt* == 1, *fd!* = resfd	T_COTS, T_COTS_ORD
accept3	successful return of *t_accept* with *ocnt* >1	T_COTS, T_COTS_ORD
snd	successful return of *t_snd*	T_COTS, T_COTS_ORD
snddis1	successful return of *t_snddis* with *ocnt* <=1	T_COTS, T_COTS_ORD
snddis2	successful return of *t_snddis* with *ocnt* >1	T_COTS, T_COTS_ORD
sndrel	successful return of *t_sndrel*	T_COTS_ORD
sndudata	successful return of *t_sndudata*	T_CLTS

Table 7: Outgoing events

2.6.3 Incoming events

The incoming events correspond to the successful return of the specified routines, where these routines retrieve data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is *pass_conn*, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no transport interface routine is issued on that endpoint. *pass_conn* is included in the state tables to describe the behavior when a user accepts a connection on another transport endpoint.

In the next table, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connection requests on the transport endpoint.

Event	Meaning	Service type
listen	successful return of *t_listen*	T_COTS, T_COTS_ORD
rcvconnect	successful return of *t_rcvconnect*	T_COTS, T_COTS_ORD
rcv	successful return of *t_rcv*	T_COTS, T_COTS_ORD
rcvdis1	successful return of *t_rcvdis* with *ocnt* <= 0	T_COTS, T_COTS_ORD
rcvdis2	successful return of *t_rcvdis* with *ocnt* = = 1	T_COTS, T_COTS_ORD
rcvdis3	successful return of *t_rcvdis* with *ocnt* > 1	T_COTS, T_COTS_ORD
rcvrel	successful return of *t_rcvrel*	T_COTS_ORD
rcvudata	successful return of *t_rcvudata*	T_CLTS
rcvuderr	successful return of *t_rcvuderr*	T_CLTS
pass_conn	receive a passed connection	T_COTS, T_COTS_ORD

Table 8: Incoming events

2.6.4 Transport user actions

In the state tables that follow, some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [n], where n is the number of the specific action as described below.

1. Set the count of outstanding connection requests to zero.
2. Increment the count of outstanding connection requests.
3. Decrement the count of outstanding connection requests.
4. Pass a connection to another transport endpoint as indicated in *t_accept*.

2.6.5 State tables

The following tables describe the transport interface state transitions. Given a current state and an event, the transition to the next state is shown, as well as any actions that must be taken by the transport user (indicated by [n]). The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state, given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action list (as specified in the previous section). The transport user must take the specific actions in the order specified in the state table.

The following should be understood when studying the state tables:

- The *t_close* routine is referenced in the state tables (see the *closed* event in table 9), but may be called from any state to close a transport endpoint. If *t_close* is called when a transport address is bound to an endpoint, the address will be unbound. Also, if *t_close* is called when the transport connection is still active, the connection will be aborted.
- If a transport user issues a routine out of sequence, the transport provider will recognize this and the routine will fail, setting *t_errno* to TOUTSTATE. The state will not change.
- If any other transport error occurs, the state will not change unless explicitly stated on the manual page for that routine. The exception to this is a TLOOK or TNODATA error on *t_connect*, as described in Table 2. The state tables assume correct use of the transport interface.
- The support routines *t_getinfo*, *t_getstate*, *t_alloc*, *t_free*, *t_sync*, *t_look* and *t_error* are excluded from the state tables because they do not affect the state.

A separate table is shown for common local management steps, data transfer in connectionless mode, and connection establishment/connection release/data transfer in connection mode.

Event	**State**		
	T_UNINIT	**T_UNBND**	**T_IDLE**
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Table 9: General state table for local management

Event	**State**
	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table 10: State table for connectionless mode

Event	**State**		
	T_IDLE	**T_OUTCON**	**T_INCON**
connect1	T_DATAXFER		
connect2	T_OUTCON		
rcvconnect		T_DATAXFER	
listen	T_INCONN [2]		T_INCONN [2]

accept1			T_DATAXFER [3]
accept2			T_IDLE [3][4]
accept3			T_INCON [3][4]
snd			
rcv			
snddis1		T_IDLE	T_IDLE [3]
snddis2			T_IDLE [3]
rcvdis1		T_IDLE	
rcvdis2			T_IDLE [3]
rcvdis3			T_INCON [3]
sndrel			
rcvrel			
pass_conn	T_DATAXFER		

Table 11: State table for connection-oriented mode (part 1)

Event	**State**		
	T_DATAXFER	**T_OUTREL**	**T_INREL**
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd	T_DATAXFER		T_INREL
rcv	T_DATAXFER	T_OUTREL	
snddis1	T_IDLE	T_IDLE	T_IDLE
snddis2			
rcvdis1	T_IDLE	T_IDLE	T_IDLE
rcvdis2			
rcvdis3			
sndrel	T_OUTREL		T_IDLE
rcvrel	T_INREL	T_IDLE	
pass_conn			

Table 12: State table for connection-oriented mode (part 2)

2.7 Guidelines for protocol independence

By providing a set of services common to many transport protocols, the transport interface offers

programmers protocol independence. However, not all transport protocols support the services supported by the transport interface. If software must be run in a variety of protocol environments, only the common services should be accessed. The following guidelines highlight services that may not be common to all transport protocols.

- In connection-oriented mode, the concept of a transport service data unit (TDDE or TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection. If messages must be transferred over a connection, a protocol should be implemented above the transport interface to support message boundaries.
- Protocol- and implementation-specific service limits are returned by the *t_open* and *t_getinfo* routines. These limits are useful when allocating buffers to store protocol-specific transport addresses and options. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.
- User data should not be transmitted with connection requests or disconnection requests (see *t_connect* and *t_snddis* in the "Networking Reference Manual (Reliant UNIX)"). Not all transport protocols support this capability.
- The buffers in the *t_call*structure used for *t_listen* must be large enough to hold any information passed by the client during connection establishment. The server should use the T_ALL argument to *t_alloc*, which determines the maximum buffer sizes needed to store the address, options, and user data for the current transport provider
- The user program should not look at or change options that are associated with any transport interface routine. These options are specific to the underlying transport protocol. The user should not pass options with *t_connect* or *t_sndudata*. In such cases, the transport provider will use default values. Also, a server should use the options returned by *t_listen* when accepting a connection.
- Protocol-specific addressing issues should be hidden from the user program. A client should not specify any protocol address on *t_bind*, but instead should allow the transport provider to assign an appropriate address to the transport endpoint. Similarly, a server should retrieve its address for *t_bind* in such a way that it does not require knowledge of the transport provider's address space. Such addresses should not be hard-coded into a program. A name server procedure could be useful in this situation, but the details for providing this service are outside the scope of the transport interface. Detailed information about transport service selection and name-to-address mapping can be found in the Chapter "Network services".
- The reason codes associated with *t_rcvdis* are protocol-dependent. The user should not interpret this information if protocol independence is important.
- The error codes associated with *t_rcvuderr* are protocol-dependent. The user should not interpret this information if protocol independence is a concern.
- The names of devices should not be hard-coded into programs, because the device node identifies a particular transport provider, and is not protocol independent.
- The optional orderly release facility of the connection-oriented service (provided by *t_sndrel* and *t_rcvrel*) should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. In particular, its use will prevent programs from successfully communicating with ISO open systems.

2.8 Some examples

The examples presented throughout this guide are shown in their entirety in this section.

2.8.1 Client in connection-oriented mode

The following code represents the connection-oriented client program described in the Section "Connection-oriented mode". This client establishes a transport connection with a server, and then receives data from the server and writes it to its standard output. The connection is released using the orderly release facility of the transport interface. This client communicates with each of the connection-oriented servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#define SRV_ADDR    1    /* server's well known address */
main()
{
int fd;
int nbytes;
int flags = 0;
char buf[1024];
struct t_call *sndcall;
extern int t_errno;
if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
t_error("t_open failed");
exit(1);
}
if (t_bind(fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(2);
}
```

```
/*
* By assuming that the address is an integer value,
* this program may not run over another protocol.
*/
if ((sndcall = (struct t_call *)t_alloc(fd,
T_CALL, T_ADDR)) == NULL) {
t_error("t_alloc failed");
exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;
if (t_connect(fd, sndcall, NULL) < 0) {
t_error("t_connect failed for fd");
exit(4);
}
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1)
if (int fwrite(buf, 1, nbytes, stdout) < 0) {
fprintf(stderr, "fwrite failed\n");
exit(5);
}
}
if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
if (t_rcvrel(fd) < 0) {
t_error("t_rcvrel failed");
exit(6);
}
if (t_sndrel(fd) < 0) {
t_error("t_sndrel failed");
exit(7);
}
exit(0);
}
t_error("t_rcv failed");
exit(8);
}
```

2.8.2 Server in connection-oriented mode

The following code represents the connection-oriented server program described in the Section "Connection-oriented mode". This server establishes a transport connection with a client, and then transfers a log file to the client on the other side of the connection. The connection is released using the orderly release facility of the transport interface. The connection-oriented client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#define DISCONNECT    -1
#define SRV_ADDR    1     /* server's well known address */
int conn_fd;              /* connection established here */
extern int t_errno;
main()
{
int listen_fd;         /* listening transport endpoint */
struct t_bind *bind;
struct t_call *call;
if ((listen_fd = t_open("/dev/ticotsord", O_RDWR, NULL))
< 0) {
t_error("t_open failed for listen_fd");
exit(1);
}
/*
* By assuming that the address is an integer value,
* this program may not run over another protocol.
*/
if ((bind = (struct t_bind *)t_alloc(listen_fd,
T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind structure failed");
exit(2);
}
bind->qlen = 1;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
if (t_bind(listen_fd, bind, bind) < 0) {
t_error("t_bind failed for listen_fd");
exit(3);
}
```

```
/*
* Was the correct address bound?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, " t_bind bound wrong address\n");
exit(4);
}
if ((call = (struct t_call *)t_alloc(listen_fd,
T_CALL, T_ALL)) == NULL) {
t_error("t_alloc of t_call structure failed");
exit(5);
}
for(;;) {
if (t_listen(listen_fd, call) < 0) {
t_error("t_listen failed for listen_fd");
exit(6);
}
if ((conn_fd = accept_call(listen_fd, call))
!= DISCONNECT)
run_server(listen_fd);
}
}
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
int resfd;
if ((resfd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
t_error("t_open for responding fd failed");
exit(7);
}
if (t_bind(resfd, NULL, NULL) < 0) {
t_error("t_bind for responding fd failed");
exit(8);
}
if (t_accept(listen_fd, resfd, call) < 0) {
if (t_errno == TLOOK) {  /* must be a disconnect */
if (t_rcvdis(listen_fd, NULL) < 0) {
t_error("t_rcvdis failed for listen_fd");
exit(9);
}
if (t_close(resfd) < 0) {
t_error("t_close failed for responding fd");
exit(10);
}
/* go back up and listen for other calls */
return(DISCONNECT);
}
t_error("t_accept failed");
exit(11);
}
return(resfd);
}
connrelease()
```

```
{
/* conn_fd is global because needed here */
if (t_look(conn_fd) == T_DISCONNECT) {
fprintf(stderr, "connection aborted\n");
exit(12);
}
/* else orderly release indication - normal exit */
exit(0);
}
run_server(listen_fd)
int listen_fd;
{
int nbytes;
FILE *logfp;       /* file pointer to log file */
char buf[1024];
switch (fork()) {
case -1:
perror("fork failed");
exit(20);
break;
default:    /* parent process */
/* close conn_fd and then go up and listen again */
if (t_close(conn_fd) < 0) {
t_error("t_close failed for conn_fd");
exit(21);
}
return;
case 0:        /* child */
/* close listen_fd and do service */
if (t_close(listen_fd) < 0) {
t_error("t_close failed for listen_fd");
exit(22);
}
if ((logfp = fopen("logfile", "r")) == NULL) {
perror("cannot open logfile");
exit(23);
}
signal(SIGPOLL, (void(*)(int))connrelease);
if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
perror("ioctl I_SETSIG failed");
exit(24);
}
if (t_look(conn_fd) != 0) {
/* was there a disconnect? */
fprintf(stderr, "t_look: unexpected event\n");
exit(25);
}
while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
t_error("t_snd failed");
exit(26);
}
if (t_sndrel(conn_fd) < 0) {
t_error("t_sndrel failed");
```

```
exit(27);
}
pause();    /* until orderly release indication */
}
}
```

2.8.3 Datagram-oriented transaction server

The following code represents the connectionless transaction server program (request system) described in the Section "Connectionless mode". This server waits for incoming datagram requests, and then processes each request and sends a response.

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>
#define SRV_ADDR    2       /* server's well known address */
main()
{
int fd;
int flags;
struct t_bind *bind;
struct t_unitdata *ud;
struct t_uderr *uderr;
extern int t_errno;
if ((fd = t_open("/dev/ticlts", O_RDWR, NULL)) < 0) {
t_error("unable to open /dev/provider");
exit(1);
}
```

```
if ((bind = (struct t_bind *)t_alloc(fd,
T_BIND, T_ADDR)) == NULL) {
t_error("t_alloc of t_bind structure failed");
exit(2);
}
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
bind->qlen = 0;
if (t_bind(fd, bind, bind) < 0) {
t_error("t_bind failed");
exit(3);
}
/*
* Is the bound address correct?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, "t_bind bound wrong address\n");
exit(4);
}
if ((ud = (struct t_unitdata *)t_alloc(fd,
T_UNITDATA, T_ALL)) == NULL) {
t_error("t_alloc of t_unitdata structure failed");
exit(5);
}
if ((uderr = (struct t_uderr *)t_alloc(fd,
T_UDERROR, T_ALL)) == NULL) {
t_error("t_alloc of t_uderr structure failed");
exit(6);
}
for(;;) {
if (t_rcvudata(fd, ud, &flags) < 0) {
if (t_errno == TLOOK) {
/*
* Error on previously sent datagram
*/
if (t_rcvuderr(fd, uderr) < 0) {
t_error("t_rcvuderr failed");
exit(7);
}
fprintf(stderr,
"bad datagram, error = %d\n",
uderr->error);
continue;
}
t_error("t_rcvudata failed");
exit(8);
}
/*
* query() processes the request and places the
* response in ud->udata.buf, setting ud->udata.len
*/
query(ud);
if (t_sndudata(fd, ud, 0) < 0) {
t_error("t_sndudata failed");
```

```
exit(9);
}
}
}
query()
{
/* Merely a stub for simplicity */
}
```

2.8.4 Read/write client

The following code represents the connection-oriented read/write client program described in the Section "Use of read/write interfaces". This client establishes a transport connection with a server, and then uses *cat* to retrieve the data sent by the server and write it to its standard output. This client will communicate with each of the connection-oriented servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>
#define SRV_ADDR    1     /* server's well known address */
main()
{
int fd;
int nbytes;
int flags = 0;
char buf[1024];
struct t_call *sndcall;
extern int t_errno;
if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
t_error("t_open failed");
exit(1);
}
```

```
if (t_bind(fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(2);
}
/*
* By assuming that the address is an integer value,
* this program may not run over another protocol.
*/
if ((sndcall = (struct t_call *)t_alloc(fd,
T_CALL, T_ADDR)) == NULL) {
t_error("t_alloc failed");
exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;
if (t_connect(fd, sndcall, NULL) < 0) {
t_error("t_connect failed for fd");
exit(4);
}
if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
perror("I_PUSH of tirdwr failed");
exit(5);
}
close(0);
dup(fd);
execl("/usr/bin/cat", "/usr/bin/cat", 0);
perror("execl of /usr/bin/cat failed");
exit(6);
}
```

2.8.5 Event-driven server

The following code represents the connection-oriented server program described in the Section "Advanced topics". This server manages multiple connection requests in an event-driven manner. Either connection-oriented client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>
#define NUM_FDS         1
#define MAX_CONN_IND    4
#define SRV_ADDR        1   /* server's well known address */
int conn_fd;                /* server connection here */
extern int t_errno;
/* holds connection requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];
main()
{
struct pollfd pollfds[NUM_FDS];
struct t_bind *bind;
int i;
/*
* Only opening and binding one transport endpoint,
* but more could be supported
*/
if ((pollfds[0].fd = t_open("/dev/ticotsord", O_RDWR, NULL))
< 0) {
t_error("t_open failed");
exit(1);
}
if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind structure failed");
exit(2);
}
bind->qlen = MAX_CONN_IND;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
if (t_bind(pollfds[0].fd, bind, bind) < 0) {
t_error("t_bind failed");
exit(3);
}
```

```
/*
* Was the correct address bound?
*/
if (*(int *)bind->addr.buf != SRV_ADDR) {
fprintf(stderr, "t_bind bound wrong address\n");
exit(4);
}
pollfds[0].events = POLLIN;
for(;;) {
if (poll(pollfds, NUM_FDS, -1) < 0) {
perror("poll failed");
exit(5);
}
for (i = 0; i < NUM_FDS; i++) {
switch (pollfds[i].revents) {
default:
perror(
"poll returns error event");
exit(6);
break;
case 0:
continue;
case POLLIN:
do_event(i, pollfds[i].fd);
service_conn_ind(i, pollfds[i].fd);
}
}
}
}
do_event(slot, fd)
{
struct t_discon *discon;
int i;
switch (t_look(fd)) {
default:
fprintf(stderr,"t_look: unexpected event\n");
exit(7);
break;
case T_ERROR:
fprintf(stderr,"t_look returned T_ERROR-event\n");
exit(8);
break;
case -1:
t_error("t_look failed");
exit(9);
break;;
```

```
case 0:
/* since POLLIN returned, this should not happen */
fprintf(stderr,"t_look returned no event\n");
exit(10);
case T_LISTEN:
/*
* find free element in calls array
*/
for (i = 0; i < MAX_CONN_IND; i++) {
if (calls[slot][i] == NULL)
break;
}
if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
T_CALL, T_ALL)) == NULL)
{
t_error("t_alloc of t_call structure failed");
exit(11);
}
if (t_listen(fd, calls[slot][i]) < 0) {
t_error("t_listen failed");
exit(12);
}
break;
case T_DISCONNECT:
discon = (struct t_discon *)t_alloc(fd,
T_DIS, T_ALL);
if (t_rcvdis(fd, discon) < 0) {
t_error("t_rcvdis failed");
exit(13);
}
/*
* find call ind in array and delete it
*/
for (i = 0; i < MAX_CONN_IND; i++) {
if (discon->sequence ==
calls[slot][i]->sequence) {
t_free(calls[slot][i], T_CALL);
calls[slot][i] = NULL;
}
}
t_free(discon, T_DIS);
break;
}
}
```

```
service_conn_ind(slot, fd)
{
int i;
for (i = 0; i < MAX_CONN_IND; i++) {
if (calls[slot][i] == NULL)
continue;
if ((conn_fd = t_open("/dev/ticotsord",
O_RDWR, NULL)) < 0) {
t_error("open failed");
exit(14);
}
if (t_bind(conn_fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(15);
}
if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
if (t_errno == TLOOK) {
t_close(conn_fd);
return;
}
t_error("t_accept failed");
exit(16);
}
t_free(calls[slot][i], T_CALL);
calls[slot][i] = NULL;
run_server(fd);
}
}
connrelease()
{
/* conn_fd is global because needed here */
if (t_look(conn_fd) == T_DISCONNECT) {
fprintf(stderr, "connection aborted\n");
exit(12);
}
/* else orderly release indication - normal exit */
exit(0);
}
run_server(listen_fd)
int listen_fd;
{
int nbytes;
FILE *logfp;       /* file pointer to log file */
char buf[1024];
switch (fork()) {
```

```
case -1:
perror("fork failed");
exit(20);
break;
default:    /* parent process */
/* close conn_fd and then go up and listen again */
if (t_close(conn_fd) < 0) {
t_error("t_close failed for conn_fd");
exit(21);
}
return;
case 0:        /* child process */
/* close listen_fd and do service */
if (t_close(listen_fd) < 0) {
t_error("t_close failed for listen_fd");
exit(22);
}
if ((logfp = fopen("logfile", "r")) == NULL) {
perror("cannot open logfile");
exit(23);
}
signal(SIGPOLL, (void(*)(int))connrelease);
if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
perror("ioctl I_SETSIG failed");
exit(24);
}
if (t_look(conn_fd) != 0) {
/* disconnect already there? */
fprintf(stderr, "t_look: unexpected event\n");
exit(25);
}
while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
t_error("t_snd failed");
exit(26);
}
if (t_sndrel(conn_fd) < 0) {
t_error("t_sndrel failed");
exit(27);
}
pause();   /* until orderly release indication arrives  */
}
}
```

3 The sockets interface

The socket interface is another important interface, in addition to the TLI, for the development of communication applications. A large number of applications have been developed using this interface, including many implementations of the protocols belonging to the TCP/IP suite. The interface provides basic functions for developing connection-oriented (TCP) and connectionless (UDP, ICMP, RAWIP) applications.

For the socket interface, Reliant UNIX offers two conventional implementations. One implementation is based on 4.3BSD, the other on Streams. Streams implementation was originally a component of UNIX SVR4 (system V Release 4). In Reliant UNIX versions up to Version 5.45, Streams was the standard implementation. In Reliant UNIX 5.45, both implementations are available but the standard implementation is now Berkeley sockets (based on 4.4BSD). The peculiarities and limitations of this interface are described in the section entitled BSD and STREAMS sockets.

In addition to there being a difference between STREAMS and BSD sockets, there are two variants of the user interface for sockets - firstly the "traditional" interface, and secondly the "X sockets" interface standardized by X/Open. Applications that use the traditional interface must be linked with the *libsocket* library.

cc *prog.c* -lsocket -lnsl

Applications that use the Xsockets interface must be linked with the *libxnet* library:

cc *prog.c* -lxnet

This chapter discusses these approaches and illustrates them with a number of sample programs. The programs demonstrate communication with datagram sockets and stream sockets. The chapter is subdivided into the following sections:

The Section "Basics" introduces you to the socket routines and the underlying communication model. The Section "Supporting routines" describes some of the library functions that are helpful when developing distributed applications. The Section "The client/server model" discusses the model used to develop applications and contains examples of the two basic types of server. The Section "Advanced functions" contains information aimed at more experienced users. In the section entitled Section "Comparison of the sockets and TLI interfaces" you will learn about the differences between the socket and TLI interfaces. The section entitled Section "Differences between BSD and STREAMS sockets" goes into the differences and selection options.

3.1 Basics

A socket is a basic building block for developing communications applications. A socket represents a transport endpoint. It can be assigned a name or address by means of which the socket can be addressed.

Each socket is of a specific type and has at least one associated process. Several related processes can use the same socket, and a process can be linked to several sockets.

A socket belongs to a specific communication domain. A communication domain consists of address families and associated protocol families. An address family groups together addresses with the same address structure. A protocol family defines a set of protocols that can use the socket types of an address family. Communication domains were introduced to group together common properties of processes that communicate with each other via sockets.

One of these properties is the way in which a name is assigned to a socket. In the UNIX communication domain, for example, sockets receive UNIX pathnames; a socket can have the name */dev/abc*, for example. In the Internet communication domain, on the other hand, the name of a socket consists of the Internet address and a port number.

Sockets normally exchange data only with sockets belonging to the same domain. Communication between different communication domains is possible, but an additional conversion process is required. The socket interface in UNIX systems supports two different communication domains: the UNIX domain for local communication and the Internet domain for processes that communicate by means of the Internet communication protocols.

The underlying communication facilities provided by these domains have a significant influence on the internal

system implementation as well as the socket functions available to users. For example, a socket operating in the UNIX domain sees a subset of the error conditions that are possible when operating in the Internet domain.

3.1.1 Socket types

Sockets have types that reflect the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

There are several types of sockets currently available:

- stream sockets
- datagram sockets
- raw sockets

Stream sockets are simply connection-oriented sockets, i.e. sockets of the type SOCK_STREAM. They can be generated and edited using both of the available implementations, i.e. the 4.4BSD-based implementation (BSD sockets) or the System V Release 4 Streams-based implementation (STREAMS sockets).

The terms "Stream sockets" and "STREAMS sockets" refer to very different relationships. To make the distinction between these very similar terms as clear as possible, the following notational conventions have been observed:

- "Stream", "stream" and "SOCK_STREAM" refer to connection-oriented sockets,
- "STREAM", "STREAMS" and "Streams" refer to the System V Release 4-based STREAMS implementation.

3.1.1.1 Connection-oriented stream sockets

A "stream socket" provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. A pair of connected stream sockets provides an interface nearly identical to that of pipes.

Stream sockets are used to develop connection-oriented communication applications on the basis of the TCP protocol.

3.1.1.2 Connectionless datagram sockets

A datagram socket supports bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated and possibly in an order different from the order in which they were sent. It is the responsibility of the application to monitor and ensure correct reception of the data. An important characteristic of a datagram socket is that record boundaries in the data are preserved.

Datagram sockets are used to develop connectionless communication applications on the basis of the UDP protocol.

3.1.1.3 Raw sockets

A raw socket provides access to the underlying communication protocols. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for users interested in developing new communication protocols, or gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in the Section "Advanced functions".

3.1.2 Socket creation

The *socket()* system call is used to create a socket:

s = socket(family, type, protocol);

This call creates a socket in the specified address family with the specified type. If the protocol is left unspecified (i.e. a value of 0), the system will select an appropriate protocol from those that comprise the domain and that may be used to support the requested socket type. A descriptor (of the type *integer*) that may be used in later system calls that operate on sockets is returned. The family is specified through fixed constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX; for the Internet domain it is AF_INET.

The constants named AF_*whatever* show the address format to use in interpreting names.

The socket types are also defined in *<sys/socket.h>* and one of SOCK_STREAM, SOCK_DGRAM or SOCK_RAW must be specified.

To create a stream socket in the Internet domain the following call might be used:

s = socket(AF_INET, SOCK_STREAM, 0);

This call creates a stream socket, which means that the TCP protocol provides the underlying communication support.

The following call creates a datagram socket for local communication:

s = socket(AF_UNIX, SOCK_DGRAM, 0);

The default protocol, which is used when the *protocol* argument is 0, should be correct for most situations. However, it is possible to specify a protocol other than the default; this will be covered in the Section "Advanced functions".

A socket call may fail for several reasons. Aside from the rare occurrence of lack of memory (ENOBUF), a socket request may fail because the request is for an unknown protocol (EPROTONOSUPPORT), or because the request is for a type of socket for which there is no supporting protocol (EPROTOTYPE).

3.1.3 Binding local names and addresses

A socket is created without a name or address. This means it contains no information on how it is to be addressed. If a socket via which a service is provided is not assigned a name or address, it is not possible for processes to address it and no messages can be received via it. Binding a socket means assigning it a local name or address corresponding to its address family. This is done using the system call *bind()*:

bind(s, name, namelen);

name is a string of variable length specifying the name or address of the socket. It is interpreted by a protocol in accordance with the address family. The interpretation can differ from one communication domain to another (this depends on the properties of a domain). In the Internet domain, *name* contains an Internet address and a port number; in the UNIX domain, it contains a pathname. The address family in the UNIX domain is always AF_UNIX. The following program code assigns a socket in the UNIX domain the name */tmp/abc*:

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

Note that in determining the size of a UNIX domain address, null bytes are not counted, which is why, *strlen()* is used. The pathname referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and the file should be deleted by the caller when it is no longer needed.

In binding an Internet address things become more complicated. The call itself is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

However, what can be set for the address *sin* requires some explanation. We will return to the problem of creating Internet addresses in the section entitled Supporting routines, in which the library routines for name resolution are described.

Sockets that communicate with each other are linked by means of address assignment. In the Internet domain, this assignment consists of a local address and a local port number, and a remote address and a remote port number. In the UNIX domain, it consists of a local and a remote pathname.

A remote pathname is a pathname created by a remote process, not a pathname used by a remote system.

<local address, local port, remote address, remote port>

When a socket is set up, it is not necessary to specify both address pairs straight away. The *bind()* system call specifies one half of the assignment:

<local address, local port> (or local pathname)

The *connect()* and *accept()* calls complete the socket assignment when a connection is established.

3.1.4 Connection establishment

Connection establishment is usually asymmetrical, with one communication partner having the active role of the client and the other the passive role of the server. If the server is ready to provide its services, it binds a socket to the address and port number defined for the service and then waits passively for a connection request for its socket. It is then possible for any process to rendezvous with the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the *connect()* call is used to initiate connection establishment.

In the UNIX domain, this might appear as:

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

In the Internet domain, it might be:

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

addr.sun_path would contain either the UNIX pathname, or the Internet address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (however, any name automatically bound by the system remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection over a period of time, the system stopped attempting the connection. This may occur when the destination host is down or when problems in the network result in lost transmissions.

ECONNREFUSED
: The host refused service. This usually occurs when a server process is not present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate routers. The status returned is not always sufficient to distinguish between a network that is down and a host that is down.

To accept a client's connection request, once its socket has been assigned a name or address, a server must perform two steps:

1. First, it must ensure that the socket receives incoming connection requests. This is done by means of the *listen()* function. The *listen()* function causes incoming connection requests to be placed in a queue in order to prevent connection requests from being lost while the server is processing a connection request.

2. It must then accept the connection with *accept()*.

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

The first parameter to the *listen()* call is the socket via which the connection is to be accepted. The second parameter specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process.

For the UNIX domain, *from* would be set with *struct sockaddr_un*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples that follow, only Internet routines will be discussed.

When a connection request is received, a new descriptor is created for the new socket. If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*. It is then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

accept() normally blocks. That is, *accept()* will not return until a connection is available or the system call is interrupted by a signal to the process.

Further, there is no way for a process to indicate that it will accept connection requests only from one or more specific partners. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; these are described in the section entitled Advanced functions).

3.1.5 Data transfer

With a connection established, data may begin to flow. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer.

There are several calls for sending and receiving data. Here, the normal, *read()* and, *write()* system calls are usable:

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

In addition to *read()* and, *write()*, the calls, *send()* and, *recv()* may be used:

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

While, *send()* and, *recv()* are virtually identical to *read()* and *write()*, the extra *flags* argument is important. The flags, defined in <*sys/socket.h*>, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB
: send/receive out-of-band data

  Out-of-band data is specific to stream sockets. This mechanism allows you to send important outgoing data packets on a separate logical channel, "bypassing" other data. This option is described in more detail in the Section "Advanced functions".

MSG_PEEK
: look at data without reading

  When MSG_PEEK is specified with a *recv()* call, any data present is returned to the user but treated as still unread. That is, the next *read()* or *recv()* call applied to the socket will return the data previously previewed.

MSG_DONTROUTE
: send data without routing packets

  This option is currently used only by the process which administers the routing table, and is probably not of interest to the majority of users.

3.1.6 Closing sockets

Once a socket is no longer of interest, it may be discarded by applying a *close()* to the descriptor,

close(s);

If a socket that provides reliable data transfer (for example, a stream socket) still holds buffered data when a *close()* is to be executed, the system will continue to attempt to transfer the data. However, if the data is still undelivered after a time set by means of a timer, the socket is closed.

If data waiting for transfer is no longer required, you can specify what is to be done with it before the socket is closed by means of the *shutdown()* call:

shutdown(s, how);

how has:

- a value of 0 if the user is no longer interested in reading data
- a value of 1 if no more data is to be sent
- a value of 2 if no data is to be sent or received

3.1.7 Example of connection-oriented client/server communication

The following two code samples illustrate how to initiate and accept an Internet domain stream connection.

Example 1: Initiation of a stream connection by the client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Beam me up, Scotty . . ."
/*
* This program creates a socket and initiates a connection
* with the socket given in the command line. One message is
* sent over the connection and then the socket is closed,
* ending the connection. The form of the command line is:
* streamwrite hostname portnumber
*/
```

```
int main(int argc, char *argv[]){
int sock;
struct sockaddr_in server;
struct hostent *hp, *gethostbyname();
char buf[1024];
/* Create socket. */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
perror("opening stream socket");
exit(1);
}
/* Connect socket using name specified by command line. */
server.sin_family = AF_INET;
hp = gethostbyname(argv[1]);
if (hp == 0) {
fprintf(stderr, "%s: unknown host\n", argv[1]);
exit(2);
}
memcpy((char *)&server.sin_addr, (char *)hp->h_addr,
hp->h_length);
server.sin_port = htons(atoi(argv[2]));
if (connect(sock,
(struct sockaddr *)&server, sizeof server ) < 0) {
perror("connecting stream socket");
exit(1);
}
if (write(sock, DATA, sizeof DATA ) < 0)
perror("writing on stream socket");
close(sock);
exit(0);
}
```

Example 2: Acceptance of the stream connection by the server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
* This program creates a socket and then begins an infinite
* loop. Each time through the loop it accepts a connection and
* outputs messages from it. When the connection breaks, or a
* termination message comes through, the program accepts a new
* connection.
*/
```

```
int main(int argc, char *argv[]) {
int sock, length;
struct sockaddr_in server;
int msgsock;
char buf[1024];
int rval;
/* Create socket. */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
perror("opening stream socket");
exit(1);
}
/* Name socket using wildcards. */
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sock, (struct sockaddr *)&server, sizeof server )
< 0) {
perror("binding stream socket");
exit(1);
}
/* Find out assigned port number and print it out. */
length = sizeof server;
if (getsockname(sock, (struct sockaddr *)&server,
&length) < 0) {
perror("getting socket name");
exit(1);
}
printf("Socket port #%d\n", ntohs(server.sin_port));
/* Start accepting connections. */
listen(sock, 5);
do {
msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
if (msgsock == -1)
perror("accept");
else do {
memset(buf, 0, sizeof buf );
if ((rval = read(msgsock, buf, 1024)) < 0)
perror("reading stream message");
if (rval == 0)
printf("Ending connection\n");
else
printf("-->%s\n", buf);
} while (rval != 0);
close(msgsock);
} while (TRUE);
```

```
/*
* Since this program has an infinite loop, the socket
* "sock" is never explicitly closed.  However, all sockets
* will be closed automatically when a process is killed or
* terminates normally.
*/
exit(0);
}
```

3.1.8 Connectionless sockets

Up to this point we have been concerned primarily with connection-oriented sockets. However, connectionless interactions using the UDP protocol are also supported. A datagram socket provides a symmetrical interface for data exchange via datagrams. In contrast to what happens in the case of connection-oriented processes, a connection is not established between the client and server sockets when datagrams are transferred. Instead, each message includes the destination address.

Datagram sockets are created as described above in the Section "Socket creation". If a particular local address is needed, a *bind()* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the, *sendto()* call is used:

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

The, *s*, *buf*, *buflen*, and *flags* parameters are used the same as with connection-oriented sockets. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message and the message length respectively. When using an unreliable datagram interface, if the recipient cannot receive a data packet, this is not reported to the sender. When information is present locally that allows the system to recognize a message that cannot be delivered (for instance when a network is unreachable), the call will return -1 and the global value *errno* will contain the error number.

To receive messages on an unconnected datagram socket, the *recvfrom()* call is used:

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

The *fromlen* parameter initially contains the size of the *from* buffer; it is modified on return to show the size of the address from which the datagram was received.

In addition to the two calls mentioned above, you can also use the *connect()* call to bind a datagram socket to a specific destination address. Here, any data sent on the socket without explicitly specifying the destination address will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time. A second connect will change the destination address. Connection requests on datagram sockets return immediately; the system simply records the peer's address.

accept() and *listen()* are not used with datagram sockets.

While a datagram socket is connected, errors from recent *send()* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status (see also Section "Socket options").

3.1.9 Example of connectionless communication

Example 3: **Reading datagrams**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
* This program creates a datagram socket, binds a name to it,
* then reads from the socket.
*/
int main(int argc, char *argv[]) {
```

```
int sock, length;
struct sockaddr_in name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
perror("opening datagram socket");
exit(1);
}
```

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

Example 4: Sending datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."
/*
* Here I send a datagram to a receiver whose name I get
* from the command line arguments.
* The form of the command line is:
* dgramsend hostname portnumber
*/
```

```
int main(int argc, char *argv[]) {
int sock;
struct sockaddr_in name;
struct hostent *hp, *gethostbyname();
/* Create socket on which to send. */
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
perror("opening datagram socket");
exit(1);
}
/*
* Construct name, with no wildcards, of the socket to send
* to. gethostbyname returns a structure including the
* network address of the specified host.  The port number
* is taken from the command line.
*/
hp = gethostbyname(argv[1]);
if (hp == 0) {
fprintf(stderr, "%s: unknown host\n", argv[1]);
exit(2);
}
memcpy( (char *)&name.sin_addr, (char *)hp->h_addr,
hp->h_length);
name.sin_family = AF_INET;
name.sin_port = htons(atoi(argv[2]));
/* Send message. */
if (sendto(sock, DATA, sizeof DATA, 0,
(struct sockaddr *)&name, sizeof name) < 0)
perror("sending datagram message");
close(sock);
exit(0);
}
```

3.1.10 Input/output multiplexing

The ability to multiplex I/O requests among multiple sockets or files is a facility that is often used in developing applications. The *select()* call is used for this type of input/output multiplexing.

This function allows a program to monitor several connections simultaneously.

The following example demonstrates the use of *select()*.

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

select() has as its arguments three pointers to bitmasks that each represent a set of descriptors. Each bit of a bitmask stands for a descriptor. *readmask* stands for a set of descriptors from which the calling process wants to read data, *writemask* stands for a set of descriptors to which data is to be written, and *exceptmask* stands for a set of descriptors that indicate the occurrence of exceptional events (such as the existence of out-of-band data). If you are not interested in one of these types of information (e.g. read, write or exceptional events), set the corresponding parameter in the *select()* call as a null pointer.

Each set is a structure containing an array of long integer bit masks. The size of the array is set by FD_SETSIZE. The array is long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set mask.

The *nfds* parameter specifies the number of descriptors to be examined in a set. Since the count begins at 0, the position of the largest descriptor to be examined plus one must be specified here.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a poll, returning immediately. If the last parameter is a NULL pointer, the *select()* call blocks until a descriptor is found.

select() normally returns the number of file descriptors selected. If the *select()* call returns because the timeout has expired, the value 0 is returned. If the *select()* terminates because of an error or interrupt, a -1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

The status of a file descriptor in a select mask may be tested with the FD_ISSET(*fd, &mask*) macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

You can use *select()* to determine whether a connection request for a socket descriptor is shown in *readmask*. Then you use an FD_ISSET(*fd, &mask*) macro to determine the relevant socket descriptor(s). If FD_ISSET returns a non-zero value for a descriptor, a connection request is waiting at this socket.

As an example, to read data from two sockets *s1* and *s2*, as it is available from each and with a five-second timeout, the following code might be used:

Example 5: Using select to check for pending connections

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <unistd.h>
```

```
□
#define NR_OF_SOCKETS 2□
#define TIMEOUT 5□
□
int handle_request(int sock);□
/*□
* This program uses select() to check that □
* someone is trying to connect□
* before calling accept().□
*/□
```

```
int main(int argc, char *argv[]) {□
int s[NR_OF_SOCKETS],□
serv_len[NR_OF_SOCKETS],□
i, rc,□
nfds;□
struct sockaddr_in serv_addr[NR_OF_SOCKETS];□
fd_set read_mask;□
struct timeval timeout;□
□
nfds = 0;□
for (i = 0; i < NR_OF_SOCKETS; i++) {□
/* Create socket */□
if ((s[i] = socket(AF_INET, SOCK_STREAM, 0)) < 0) {□
perror("socket()");□
exit(1);□
}□
□
/* Name socket using wildcards */□
serv_addr[i].sin_family = AF_INET;□
serv_addr[i].sin_addr.s_addr = INADDR_ANY;□
serv_addr[i].sin_port = htons(0);□
serv_len[i] = sizeof(serv_addr[i]);□
if (bind(s[i], (struct sockaddr *) &serv_addr[i],□
serv_len[i]) < 0) {□
perror("bind()");□
exit(1);□
}□
□
/* Find out assigned port number and print it out */□
if (getsockname(s[i],(struct sockaddr *) &serv_addr[i],□
&serv_len[i]) < 0) {□
perror("getsockname()");□
exit(1);□
}□
printf("s[%d]: Port-# %d\n",□
i, ntohs(serv_addr[i].sin_port));□
□
/* Accept connection requests on socket s[i]  */□
if (listen(s[i], 5) < 0) {□
perror("listen()");□
exit(1);□
}□
□
nfds = s[i] > nfds ? s[i]:nfds;□
}□
```

```
FD_ZERO(&read_mask);
 
for(;;) {
for (i = 0; i < NR_OF_SOCKETS; i++)
FD_SET(s[i],&read_mask);
timeout.tv_sec = TIMEOUT;
timeout.tv_usec = 0;
 
rc = select(nfds + 1, &read_mask, (fd_set *) 0,
(fd_set *) 0, &timeout);
if (rc < 0) {
perror("select()");
exit(1);
}
else if (rc == 0) {
printf("select(): no connection request in "
"%d seconds\n", TIMEOUT);
}
else {
for (i = 0; i < NR_OF_SOCKETS; i++)
if (FD_ISSET(s[i], &read_mask))
handle_request(s[i]);
}
}
}
int handle_request(int sock) {
int new_sock,
peer_len;
struct sockaddr_in peer_addr;
/*
* Accept connection, output peer address
* and close connection
*/
peer_len = sizeof(peer_addr);
if ((new_sock=accept(sock, (struct sockaddr *) &peer_addr,
&peer_len)) < 0) {
perror("accept()");
return(-1);
}
printf("connection request on socket %d of %s:%d "
"accepted\n",
sock, inet_ntoa(peer_addr.sin_addr),
ntohs(peer_addr.sin_port));
if (close(new_sock) < 0) {
perror("close()");
return(-1);
}
return(0);
}
```

select() provides a synchronous multiplexing mechanism. The SIGIO and SIGURG signals described in the Section "Advanced functions" may be used to provide asynchronous notification of output completion, input availability or the existence of exceptional conditions.

3.2 Supporting routines

It was mentioned in the previous section that you may have to determine and create network addresses when using the communication facilities in a distributed environment. The socket library contains various routines that help you do this. In this section we will consider the routines provided for handling network addresses.

Locating a service on a remote host requires many levels of address mapping before client and server may communicate with each other. To make things clearer for the user, a name is assigned to each service and each host (e.g. the service *login* on the host *monet*). The service name and host name must then be translated into a number and a network address, respectively. This service number and the network address are then used to determine the physical location of the service and the route to it.

The physical address of a host should not be recognizable from the host name. Instead, it is the task of underlying network services to discover the location of the host when another machine wants to communicate with it. As a result, the physical location of a host can be changed without affecting how it is addressed by its peers.

The following routines are available for mapping:

- host names to Internet addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers and the appropriate protocol for communication with the server process

You must include the <*netdb.h*> file if you want to use one of these routines.

3.2.1 Host names

The mapping of a host name to an Internet address is represented by the *hostent* structure:

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

The routine *gethostbyname()* takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr()* maps Internet host addresses into a *hostent* structure. The routine *inet_ntoa()* maps an Internet address to a string for output.

The official name of the host and its public aliases (as contained in */etc/inet/hosts*) are returned by these routines, along with the address type (domain) and a null terminated list of variable length addresses. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

3.2.2 Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

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

The routines *getnetbyname*, *getnetbyaddr*, and *getnetent* are the network counterparts to the host routines described above.

3.2.3 Protocol names

For protocols, the *protoent* structure defines the protocol name to number mapping used with the routines *getprotobyname*, *getprotobynumber*, and *getprotoent*:

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

3.2.4 Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific port and use a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended.

A service mapping is described by the *servent* structure:

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

The routine *getservbyname()* maps service names to a *servent* structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

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

returns the port number for a Telnet server using any protocol, while the call

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

returns the port number of a Telnet server that uses the TCP protocol.

The *getservbyport()* routine is also provided. It works in a similar way to *getservbyname()*, except that in this case the port number is passed and the name supplied.

3.2.5 Miscellaneous

You can use the routines described above to develop socket application programs that use names rather than addresses. This allows you to develop services that are, as far as is possible, network-independent. However, it is not possible to keep them free of all network dependencies. The Internet address must be specified in an application program, so when a name is assigned to a service or socket, the application always has a certain amount of network dependency.

For an example of the use of mapping routines, refer to example 6 below, in which you will find code that is integrated in client programs such as *remote login*. (This example is considered in more detail in the Section "The client/server model".)

In addition to the database routines for the mapping of names to addresses and vice versa, there are various other routines in the run-time library that may be of use to you. These simplify the handling of names and addresses.

In some architectures, host byte ordering and network byte ordering are different. Consequently, programs sometimes have to change these byte orders. The following table outlines the routines for changing byte strings of variable length and for manipulating the byte order of network addresses and values. These library routines should be used for creating portable sockets applications.

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

	order
ntohs(*val*)	16-bit quantity from network into host byte order

Table 13: Library routines for converting byte orders

The library routines that return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. Users should therefore encounter byte swapping problems only when interpreting network addresses.

For example, the following code outputs an Internet port number:

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

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

Example 6 - Client code of the remote login

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
int main(int argc, char *argv[]) {
struct sockaddr_in server;
struct servent *sp;
struct hostent *hp;
int s;
/*
...
*/
sp = getservbyname("login", "tcp");
if (sp == NULL) {
fprintf(stderr, "rlogin: tcp/login: unknown service\n");
exit(1);
}
hp = gethostbyname(argv[1]);
if (hp == NULL) {
fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
exit(2);
}
memset((char *)&server, 0, sizeof server);
memcpy((char *)&server.sin_addr, hp->h_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
s = socket(AF_INET, SOCK_STREAM, 0);
if (s < 0) {
perror("rlogin: socket");
exit(3);
}
/*    * Binding a privileged ports below 1024    * using rresvport() (cf.chapter 1. )    */   /*    s = rresvport(...);
*/
```

```
/* Connect does the bind for us */
if (connect(s, (struct sockaddr *)&server, sizeof server) <0)
{
perror("rlogin: connect");
exit(5);
}
...
exit(0);
}
```

3.3 The client/server model

The most commonly used paradigm in building distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server that has been examined in the Section "Basics". In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol that must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol, a terminal emulation in the Internet. An example of an asymmetric protocol is the Internet file transfer protocol, FTP.

Regardless of whether the protocol used for a service is symmetric or asymmetric, when accessing a service there is a client process and a server process. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process wakes up and services the client, performing whatever appropriate actions the client requests of it.

3.3.1 Connection-oriented servers

Servers are accessed by means of their well known Internet addresses. The form of their main loop is illustrated in the following example for an *rlogin* server:

Example 7 - Connection-oriented server

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <sys/syslog.h>
int main(int argc, char *argv[]) {
int f;
struct sockaddr_in from;
struct sockaddr_in sin;
struct servent *sp;
sp = getservbyname("login", "tcp");
if (sp == NULL) {
fprintf(stderr,
"rlogind: tcp/login: unknown service\n");
exit(1);
}
/*
```

```
...
*/
f = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_port = sp->s_port;     /* Restricted Port */
sin.sin_addr.s_addr = INADDR_ANY;
if (bind(f, (struct sockaddr *)&sin, sizeof sin) < 0) {
/*
...
*/
}
```

```
/*
...
*/
listen(f, 5);
for (;;) {
int g, len = sizeof from;
g = accept(f, (struct sockaddr *) &from, &len);
if (g < 0) {
if (errno != EINTR)
syslog(LOG_ERR, "rlogind: accept: %m");
continue;
}
if (fork() == 0) {
close(f);
doit(g, &from);
}
close(g);
}
exit(0);
}
```

The result of the *getservbyname()* call (namely the port number) is used in later portions of the code to bind the socket to the local port defined for the *login* service and the TCP protocol.

The server creates a socket and waits for a connection request. The *bind()* call is required to ensure the server listens for connection requests at the correct address. Note that the remote login server accepts connection requests on a privileged port number and must therefore be called with the *root* login name. This concept of a privileged port number is covered in the section entitled Section "Advanced functions" below.

The main body of the loop is simple:

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

An *accept()* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in the Section "Advanced functions"). Therefore, the return value from *accept()* is checked to insure a connection has been established, and an error report is logged via *syslog()* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept()* is closed in the parent. The address of the client is also handed the *doit()* routine because it requires it in authenticating clients.

3.3.2 Connection-oriented clients

The client side of the remote login service was shown in example 6. One can see the separate, asymmetric

roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client *remote login* process. As in the server process, the first step is to locate the service definition for a *remote login*:

Example 8 - Connection-oriented client

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

Next, the destination host (as passed in *argv[]*) is looked up with a *gethostbyname()* call:

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

With this done, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

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

A socket is created, and a connection initiated. Note that *connect()* implicitly performs a *bind()* call, since s is unbound.

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

The details of the remote login protocol will not be considered here.

3.3.3 Connectionless data transfer

While connection-oriented services are the norm, some services are based on the use of datagram sockets and are thus connectionless.

Example 9 - Connectionless server

```
/* Copyright (C) Siemens Nixdorf Informationssysteme AG 1992
* All rights reserved
*
* AF_INET SOCK_DGRAM Server for testing system calls
* sendto()/recvfrom()
*/
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/uio.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <string.h>
#include <errno.h>
#define SERV_PORT 5000
#define CLNT_PORT 5001
#define MAXMESG 2048
int main(int argc, char *argv[]) {
void dg_echo(),
error_exit();
int soc, servlen, clilen;
struct sockaddr_in cli_addr,
serv_addr;
if((soc = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
error_exit("Error creating the socket",errno);
memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_PORT);
servlen=sizeof(serv_addr);
```

```
if(bind(soc, (struct sockaddr *) &serv_addr, (size_t)
servlen) < 0)
error_exit("Error binding the socket",errno);
memset((char *) &cli_addr, 0, sizeof(cli_addr));
clilen=sizeof(cli_addr);
dg_echo(soc, (struct sockaddr *) &cli_addr, clilen);
close(soc);
exit(0);
}
void error_exit(char *estring, int erno) {
/*  Output error message and terminate */
fprintf(stderr,"Udp_svr: %s ",estring);
fprintf(stderr,"[%d: %s]\n",erno, strerror(erno));
fflush(stderr);
exit(erno);
}
void dg_echo(int soc, struct sockaddr *addr, int addrlen) {
/*
* Datagram echo; read datagram from a UDP socket and
* write it back.
*/
int n;
char mesg[MAXMESG];
for (;;) {
#ifdef SVR_DEBUG
fprintf(stderr,"\tUdp_svr: Receiving... ");fflush(stderr);
#endif
n=recvfrom(soc, mesg, MAXMESG, 0, addr, (size_t *)&addrlen);
#ifdef SVR_DEBUG
fprintf(stderr,"done[%d]\n", n);
fprintf(stderr,"Received from: [%d]\n", addr->sin_family );
fprintf(stderr,"   : [%d]\n", ntohl(addr->sin_addr.s_addr));
fprintf(stderr,"   : [%d]\n", ntohs(addr->sin_port));
fflush(stderr);
#endif
if(n<0) error_exit("Recvfrom failed ",errno);
if (n == 0) return;
fwrite(mesg, 1, n, stdout);
fflush(stdout);
#ifdef SVR_DEBUG
fprintf(stderr, "\tUdp_svr: Sending ... "); fflush(stderr);
#endif
if(sendto(soc,mesg, n, 0, addr, (size_t) addrlen) != n)
error_exit("Sendto failed",errno);
```

```
#ifdef SVR_DEBUG
fprintf(stderr, "done[%d]\n",n); fflush(stderr);
#endif
} /* End for(;;) - Loop */
}
```

Example 10 - Connectionless client

```
/* Copyright (C) Siemens Nixdorf Informationssysteme AG 1992 * All rights reserved * AF_INET
SOCK_DGRAM Client for testing system calls * sendto()/recvfrom()*/#include <stdio.h>#include
<sys/types.h>#include <sys/socket.h>#include <netinet/in.h>#include <netdb.h>#include
<arpa/inet.h>#include <stdlib.h>#include <string.h>#include <errno.h>#define SERV_PORT  5000#define
MAXLINE 512int main(int argc, char *argv[]) {  void dg_cli(),       error_exit();  int soc, servlen;  FILE *fileptr;
struct sockaddr_in serv_addr;  struct hostent *hp;  char svrname[33],       *str, *inet_dot;  if (argc < 2)
{      fprintf(stderr,"Usage %s filename\n",argv[0]);      exit(1);  }  if ((fileptr = fopen(argv[1],"r")) == NULL)
{      fprintf(stderr,"Can't open file %s \n",argv[1]);      exit(1);  }  if((soc = socket(AF_INET, SOCK_DGRAM, 0))
< 0)      error_exit("Socket() failed",errno);
```

```
/* Prepare structure for addressing the server*/  str = (char *)getenv("REMHOST");  if ( str == NULL ||
strcmp(str,"") == 0 )      gethostname(svrname,33);  else strcpy(svrname, str);
if ((hp = gethostbyname(svrname))  == NULL)      error_exit("Gethostbyname failed", errno);  if
((*(hp->h_addr_list)) != NULL)     if((inet_dot = (char *)inet_ntoa(*((struct in_addr *)     hp->h_addr))) == NULL)
error_exit("inet_ntoa failed", errno);  memset((char *) &serv_addr, 0, sizeof(serv_addr));  serv_addr.sin_family
= AF_INET;  serv_addr.sin_addr.s_addr = inet_addr(inet_dot);  serv_addr.sin_port        =
htons(SERV_PORT);  servlen=sizeof(serv_addr);  dg_cli(fileptr,soc, (struct sockaddr *) &serv_addr, servlen);
close(soc);  exit(0);}void error_exit(char *estring, int erno) {  fprintf(stderr,"Udp_clnt: %s ",estring);
fprintf(stderr,"[%d: %s]\n",erno, strerror(erno));  fflush(stderr);  exit(erno);}voiddg_cli(FILE *fp, int soc, struct
sockaddr *addr, int addrlen)/* * Read lines from a file and write to the socket. * Then read the line from the
socket and output to * standard output.*/{  int n;  char sendline[MAXLINE],       recvline[MAXLINE];  while
(ferror(fp) == 0 && feof(fp) == 0 ) {     if ((n = fread(sendline, 1, MAXLINE, fp)) <= 0 ) return;         /*
error_exit("Reading from file failed", errno); */#ifdef CLNT_DEBUG     fprintf(stderr, "\tUdp_clnt: Sending... ");
fflush(stderr);#endif     if(sendto(soc, sendline, n, 0, addr, (size_t)addrlen) != n)            error_exit("Sending
failed",errno);
```

```
#ifdef CLNT_DEBUG     fprintf(stderr, "finished[%d]\n",n); fflush(stderr);#endif#ifdef CLNT_DEBUG
fprintf(stderr, "\tUdp_clnt: Receiving... "); fflush(stderr);#endif  n = recvfrom(soc, recvline, MAXLINE, 0, (struct
sockaddr *) 0, 0);#ifdef CLNT_DEBUG    fprintf(stderr, "finished[%d]\n",n); fflush(stderr);#endif      if (n == 0)
return;      if(n< 0) error_exit("Receiving failed",errno);      fwrite(recvline, 1, n, stdout);      fflush(stdout);  }
if(ferror(fp)) error_exit("File pointer error",errno);}
```

3.4 Advanced functions

Several facilities have yet to be discussed. For most programmers, the mechanisms already described will suffice for developing distributed applications. However, others will need to use some of the features that we consider in this section. This section describes the following:

- out-of-band data
- non-blocking sockets
- interrupt-driven socket I/O
- signals and process groups
- protocol selection
- address binding
- address assignment with wildcards
- broadcasting network configuration
- socket options

3.4.1 Out-of-band data

The stream socket abstraction includes the notion of out-of-band data. out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. This makes it possible to give special treatment to extremely important messages and allow them to "bypass" other data.

The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and only one message may be pending delivery to the user at any one time. For communications protocols (such as TCP) that support only in-band signaling (i.e., the urgent data is delivered in sequence with the normal data), the system normally extracts the urgent data from the normal data stream and stores it separately.

This allows users to choose between receiving urgent data in the normal data stream or receiving it immediately without having to read the data that comes before it in the sequence.

It is possible to view out-of-band data by means of MSG_PEEK. If the system detects the existence of urgent data, the process bound to the socket is notified of this by a SIGURG signal. In order to be able to receive the SIGURG signal, the process must set a process group or process ID with *fcntl()*, as described below for SIGIO.

If multiple sockets can have out-of-band data awaiting transfer, you can use a *select()* call to evaluate the *exceptmask* bitmask and determine at which socket important data is awaiting transfer. Neither the signal nor the *select* call indicate the arrival of out-of-band data, merely that it exists.

In addition to the information passed, a logical mark is placed in the data stream to specify the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility, for example, to propagate signals between client and server processes. When a signal flushes any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

Normally, only one byte is sent as an out-of-band message. This byte is not stored in the normal data stream. You can send an out-of-band message like this by setting the MSG_OOB flag in a *send()* or *sendto()* call. You can receive an out-of-band message by setting MSG_OOB in a *recvfrom()* or *recv()* call.

If you set the SO_OOBLINE option for a socket with *setsockopt()*, an out-of-band message remains in the normal data stream and can be read without the MSG_OOB flag. In this case, you can use SIOCATMARK in an *ioctl* call to determine where in a data stream out-of-band data occur:

ioctl(s, SIOCATMARK, &yes);

If *yes* is 1 on return, the next read will return data after the mark. Otherwise (assuming out-of-band data has

arrived), the next read will provide data sent by the client before transmission of the out-of-band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in the following example. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
int out = FWRITE;
char waste[BUFSIZ];
int mark;
/* flush local terminal output */
ioctl(1, TIOCFLUSH, (char *)&out);
for (;;) {
if (ioctl(rem, SIOCATMARK, &mark) < 0) {
perror("ioctl");
break;
}
if (mark)
break;
(void) read(rem, waste, sizeof waste);
}
if (recv(rem, &mark, 1, MSG_OOB) < 0) {
perror("recv");
...
}
...
}
```

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time. With such protocols, the out-of-band byte may not yet have arrived when a *recv()* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be so much in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data before the urgent data may be delivered.

Some programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g. *telnet*) need to retain the position of urgent data within the socket stream. This can be achieved by means of the socket option SO_OOBINLINE (see the *setsockopt()* function). With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent messages causes the mark to move, but no out-of-band data is lost.

3.4.2 Non-blocking sockets

It is occasionally convenient to make use of sockets that do not block; that is, I/O requests that cannot be completed immediately and would therefore block the process are not executed, and an error code is returned. Once a socket has been created via a *socket()* call, it may be marked as non-blocking by *fcntl()* as follows:

```
#include <fcntl.h>
#include <sys/file.h>
...
int     s;
...
```

```
s = socket(AF_INET, SOCK_STREAM, 0);
...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
perror("fcntl (s, F_SETFL,FNDELAY) <0) F_SETFL, FNDELAY");
exit(1);
}
...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking.

The *accept()* and *connect()* functions and all read and write operations can return the EWOULDBLOCK error code, and processes should be prepared to deal with such return codes. If an operation such as a *send()* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will show the amount actually sent.

3.4.3 Interrupt-driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by means of the *sigaction()* routine. Second, it must set the process id or process group id that is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This can be done by using a *fcntl()* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl()* call. Sample code to allow a given process to receive SIGIO signals is given in the next example. Calling the signal handling routine for SIGIO ensures that the process is informed asynchronously whenever data can be read or written. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
#include <sys/file.h>
...
int     io_handler();
...
signal(SIGIO, io_handler);
/* Set the process to receive SIGIO/SIGURG signals. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
perror("fcntl F_SETOWN");
exit(1);
}
/* Allow receipt of asynchronous I/O signals. */
if (fcntl(s, F_SETFL, FASYNC) < 0) {
perror("fcntl F_SETFL, FASYNC");
exit(1);
}
```

3.4.4 Signals and process groups

For the use of the SIGURG and SIGIO signals, each socket is assigned a process id, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl()*, such as was done in the code above for SIGIO.

The process number of a socket is set for signals by means of a positive argument with the *fcntl()* call. To set the socket's process group for signals, a negative argument is passed with *fcntl()*.

The only acceptable arguments for these system calls are the process id or negative process group id of the calling process. The process group ids must have the same absolute value as the process id of the calling process (the process must be the process group leader of its own process group). Therefore, the only allowed

recipient of SIGURG and SIGIO signals is the calling process or its process group.

Note that the process id shows either the associated process id or the associated process group id; it is impossible to specify both at the same time. A similar *fcntl()*, F_GETOWN is available for determining the current process number of a socket.

Note that the receipt of SIGURG and SIGIO can also be enabled by using the *ioctl()* call. SIOCSPGRP in the *ioctl()* call is used to make the process group of the user process known to the socket.

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

Another signal that is useful when building server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to reap child processes that have terminated. The program loop in the program segment of the remote login server in the following example may be augmented as follows:

```
int reaper();
...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
int g, len = sizeof from;
g = accept(f, (struct sockaddr *)&from, &len,);
if (g < 0) {
if (errno != EINTR)
syslog(LOG_ERR, "rlogind: accept: %m");
continue;
}
...
}
...
#include <wait.h>
reaper()
{
int status;
while (wait(&status) > 0)
continue;
}
```

If a parent process (the server) does not query the *exit* status of its child processes with *wait()*, zombie processes may be created.

3.4.5 Selecting specific protocols

If the third argument to the *socket()* call is 0 (see Section "Socket creation"), *socket()* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually appropriate for the address family of the socket, and alternative choices are not usually available. For example, the default protocol for a stream socket (SOCK_STREAM) in the Internet domain (AF_INET) is IP_PROTO, i.e. a protocol from the Internet family. For a connection-oriented socket it will be TCP. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the *protocol* argument may be important for setting up demultiplexing. For example, raw sockets in the Internet domain may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the protocol domain (in */etc/inet/protocols*). For the Internet domain one may use one of the library routines discussed in the Section "Supporting routines". Here is an example of how to use *getprotobyname()*:

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

This would result in a socket *s* using a stream based connection, but with protocol type of *newtcp* instead of the default *tcp*.

3.4.6 Address binding

As was mentioned in the Section "Basics", binding addresses to sockets in the Internet domain can be complex. As a brief reminder, sockets which communicate with each other are associates by means of addresses. In the Internet domain, these associations are composed of local and foreign addresses, and local and foreign ports. The *bind()* system call defines the local end of the assignment, namely *<local address, local port>*. The *connect()* and *accept()* system calls are used to complete the association by specifying the *<remote address, remote port>* part. Since the association is created in two steps the association uniqueness requirement mentioned previously could be violated unless care is taken. Further, it is unrealistic to expect user programs always to know proper values to use for the local address and local port since a host may reside on multiple networks and the assigned port numbers may not be directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a wildcard address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <*netinet/in.h*>), the system interprets the address as any valid address. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

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

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests that are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network to connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
...
}
else
memcpy((char *) sin.sin_addr, hp->h_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

If an unbound socket is addressed by a *connect()* call, the system looks for the local port number on the basis of two criteria:

1. Internet port numbers lower than the value of IPPORT_RESERVED (1024) are reserved for privileged users. Internet port numbers higher than the value of IPPORT_USERRESERVED (5000) are reserved for non-privileged servers. There are two areas in the system: one for privileged port numbers and one for non-privileged port numbers.
2. The port number must not currently be assigned to another socket.

You can use the *rresvport()* library routine to find a free Internet port number in the privileged area. The following sample code returns a stream socket with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
...
s = rresvport(&lport);
if (s < 0) {
if (errno == EAGAIN)
fprintf(stderr, "socket: all ports in use\n");
else
perror("rresvport: socket");
...
}
```

This restriction was placed on port allocation to allow processes executing in a secure environment to do authentication based on the originating address and port number. For example, the *rlogin* command allows users to log in across a network without being asked for a password, provided that two conditions are met: First, the name of the system the user is logging in from must be in the file */etc/hosts.equiv* on the system being logged in to (or the system name and the user name must be in the user's *.rhosts* file in the user's home directory). Second, the user's *rlogin* process must come from a privileged port on the machine from which the user is logging in. The port number and network address of the machine from which the user is logging in can be determined either from the *accept()* call (the *from* result), or the *getpeername()* call.

In certain cases the algorithm for the automatic selection of port numbers is unsuitable for an application program. This is because associations are created in a two step process. For example, the Internet file transfer protocol FTP specifies that data connections must always originate from the same local port. Duplicate associations are avoided by creating associations between the same local port and different remote ports. The prerequisite for this is that a bound local socket can be bound again immediately. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed before address binding:

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

With the above call, local addresses may be bound that are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

3.4.7 Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks connected to the system. The network itself must support broadcast; the system provides no simulation of broadcast in the software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets that are explicitly marked as allowing broadcasting.

Broadcasting is normally used for one of the following reasons:

- It is desired to find a resource on a local network without prior knowledge of its address.
- Important functions such as routing require that information be sent to all accessible hosts.

To send a broadcast message, a datagram socket must be created:

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

The socket is marked as allowing broadcasting,

```
int      on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

Finally, a port number is bound to the socket:

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

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, namely the address INADDR_BROADCAST (defined in <*netinet/in.h*>.

To determine the list of addresses for all hosts that can be reached by broadcast requires knowledge of the networks to which the host is connected.

Under UNIX, this information can be obtained in a host-independent fashion. The *ioctl* call SIOCGIFCONF returns the interface configuration of a host as an *ifconf* structure. This structure contains a data area that is made up of an array of *ifreq* structures, one for each address family supported by a network interface to which the host is connected. These structures are defined in <*net/if.h*> as follows:

```
struct ifreq {
#define IFNAMSIZ    16
char ifr_name[IFNAMSIZ];          /* if name, e.g., "en0" */
union {
struct sockaddr ifru_addr;
struct sockaddr ifru_dstaddr;
char ifru_oname[IFNAMSIZ];    /* other if name */
struct sockaddr ifru_broadaddr;
short ifru_flags;
int ifru_metric;
char ifru_data[1];     /* interface dependent data */
char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr      ifr_ifru.ifru_addr     /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr  /* other end of */
/* p-to-p link */
#define ifr_oname     ifr_ifru.ifru_oname    /* other if name */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast */
/* address   */
#define ifr_flags     ifr_ifru.ifru_flags    /* flags */
#define ifr_metric    ifr_ifru.ifru_metric   /* metric */
```

```
#define ifr_data      ifr_ifru.ifru_data     /* for use by */
/*interface   */
#define ifr_enaddr    ifr_ifru.ifru_enaddr   /* ethernet */
/* address  */
};
```

The call that obtains the interface configuration is:

```
struct ifconf ifc;
char buf[BUFSIZ];
ifc.ifc_len = sizeof buf;
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
...
}
```

After this call *buf* will contain a list of *ifreq* structures, one for each network to which the host is connected. These structures will be ordered first by interface name and then by supported address families. *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of "interface flags" that tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

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

Once the flags have been obtained, the broadcast address must be obtained. With broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;
if (ifr->ifr_flags & IFF_POINTOPOINT) {
if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
...
}
memcpy((char *) &dst, (char *) &ifr->ifr_dstaddr,
sizeof ifr->ifr_dstaddr);
} else if (ifr->ifr_flags & IFF_BROADCAST) {
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
sizeof ifr->ifr_broadaddr);
}
```

After the appropriate *ioctl* has obtained the broadcast or destination address (now in *dst*), the *sendto()* call may be used:

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

In the above loop one *sendto()* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the sender's address and port number; datagram sockets must be assigned names before they can send a message.

Sending broadcasts on a local network (limited broadcasts)

A special broadcast is defined in the Internet domain: INADDR_BROADCAST (defined in *<netinet/in.h>*). In

conventional Internet notation, this is the address 255.255.255.255. Packets sent with this as the recipient address must be received by all hosts in a local network. They are not forwarded by routers.

In the conventional definition for sending such broadcasts, it was not specified via which network interfaces the packets be sent. In the case of hosts with only one interface this is not a problem, but it is when a host has interfaces to a number of networks.

The following rules apply:

- The user can specify via which network interface (and thus to which local network) a broadcast is sent. To do this, the user must bind the socket via which the broadcast packet is to be sent to a local address. The interface with this address must, of course, support broadcasts.

  Example: A host has two network interfaces and thus two names:

  ```
  luna     129.22.0.1
  luna-gw  129.25.1.1
  ```

  To send a broadcast packet to network 129.25.0.0, you proceed as follows:

  ```
  ...
  struct sockaddr_in local;
  struct sockaddr_in dst;
  /* Create datagram socket */
  s = socket(AF_INET,SOCK_DGRAM,0);
  /* Error check ...*/
  /* Set the broadcast option for the socket */
  {
  int bc = 1;
  error = setsockopt(s,SOL_SOCKET,SO_BROADCAST,&bc,sizeof(bc);
  /* Error check ... */
  }
  /* Bind socket to a local address */
  {
  struct hostent *hp;
  hp = gethostbyname("luna-gw");
  if (hp == NULL)
  {
  /* Error handling ...*/
  }
  local.sin_family = AF_INET;
  memcpy(&local.sin_addr,hp->h_length);
  local.sin_port = htons(local_port);
  if (bind(s,struct sockaddr *)&local,sizeof(local)) != 0)
  {
  /* Error handling ...*/
  }
  }
  /* Send broadcast packet */
  dst.sin_family = AF_INET;
  dst.sin_addr.s_addr = INADDR_BROADCAST;
  dst.sin_port = htons(remote_port);
  if (sendto(s,data,dat_len,0,(struct sockaddr *)&dst,sizeof(dst)) < 0
  {
  /* Error handling ... */
  }
  ```

- If the socket via which a broadcast packet is sent is not bound to a local address, the system selects an interface (and thus a local network) via which the packet is sent. A network interface whose broadcast address is 255.255.255.255 is preferred. If there is no such interface, any interface that supports

broadcasts is supported.

A broadcast packet for a local network (i.e. one with the recipient address 255.255.255.255) is always sent to only one local network, even if the host has ports to more than one.

- A packet sent to the network address 255.255.255.255 by means of *sendto()* is always sent to a local network as a broadcast. In the protocol header of the associated packet, the value of INADDR_BROADCAST is also entered as the recipient address.

3.4.8 Socket options

It is possible to set and get several options on sockets via the *setsockopt()* and *getsockopt()* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc.

The general forms of the calls are:

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

and

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

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *level* specifies the protocol layer on which the option is to be applied; usually this is the socket level, indicated by the symbolic constant SOL_SOCKET, defined in <*sys/socket.h*>. The option is specified in *optname*, and is a symbolic constant also defined in <*sys/socket.h*>. *optval* and *optlen* point to the value of the option (usually, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt()*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified on return to show the amount of storage used.

An example should help clarify things. The call queries the socket type:

```
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type,
&size) < 0) {
...
}
```

After the *getsockopt()* call, *type* will be set to the value of the socket type, as defined in <*sys/socket.h*>. If, for example, the socket were a datagram socket *type* would have the value corresponding to SOCK_DGRAM.

The socket options SO_SNDBUF and SO_RCVBUF

SO_SNDBUF and SO_RCVBUF are options for the *getsockopt()* and *setsockopt()* system calls at SOL_SOCKET level. These options enable the send and receive buffer sizes for a socket to be adapted with a calling application. The send and receive buffer sizes can have any positive value smaller than *sb_max*. The value of *sb_max* can be checked and modified with sysctl kern.maxsockbuf.

In the case of a TCP socket which is bound with the server or which accepts connections, the system performs calculations to find a suitable default value for the buffer size and modifies the previous value accordingly. The system overwrites all previously lower values. From this point on, an application may increase, but not decrease, the buffer size.

When sending data via a TCP socket, the SO_SNDBUF option for increasing the send buffer size is useful for ensuring that the sending socket is able to buffer sufficient data for the proper flow of data via the TCP connection.

In the case of a socket receiving data, increasing the receive buffer size with the SO_RCVBUF option can have a negative result, which is noticeable when the receiving socket is bound with the sender of the data. The receiving socket announces a large receive window and enables the sending side to send if the acknowledgments from the receive side are delayed: the sender can send data as long as it is assumed that the receive window on the receive side is not full.

Please note that the send buffer size can be increased for a socket at any time during its "lifetime". The same applies to the socket's receive buffer size. However, this only has an affect on the announced receive window if the modification took place before the socket established a connection to a server using *connect()* or, in the case of a server socket, using *accept()*. In the case of a server socket modified with *accept()*, the settings configured for the original server socket are retained.

Remember that a connection from or to this socket can overwrite all settings made with *setsockopt()* if the system decides that the buffer size was initialized with a smaller value than the defaults calculated by the

system.

For more information on send and receive window sizes (buffers), see the "Network Administration" manual.

3.4.9 Traditional sockets and T/TCP

The experimental protocol T/TCP described in RFC1664 has been implemented in Reliant UNIX systems with the "enhanced" protocol stack. T/TCP is an enhancement of the standard TCP protocol which enables short-term TCP connections to operate more efficiently than is possible with the standard TCP protocol. Transaction-oriented applications can really benefit from this protocol. The corresponding programming interface is not described here; for information see Richard W. Stevens, TCP/IP Illustrated, Volume3. There are a number of points worth noting in addition to the information contained in this publication:

1. A system with the "enhanced" protocol stack must be active:

   $ test_enhanced
   enhanced
   $
2. T/TCP support must be activated:

   $ sysctl net.inet.tcp.rfc1644
   net.inet.tcp.rfc1644 = 1
   $
3. T/TCP can only be used with traditional BSD sockets; T/TCP is not supported by Xsockets.
4. Because of security problems, the description by Richard W. Stevens has been expanded to include the following change for the implementation of T/TCP servers:

   In the case of a server that accepts T/TCP connections and is using the T/TCP protocol, the TCP_NOPUSH socket option must be enabled for the socket via which these connections are to be accepted. This is done as follows with a *setsockopt()* call:
   int opt = 1;
   setsockopt(s, IPPROTO_TCP, TCP_NOPUSH, (char *)&opt,
   sizeof(int));

   If this is not done, the connection requests are accepted anyway and the connections are set up correctly. However, the standard TCP protocol is used which means that the same level of efficiency cannot be reached as with the T/TCP protocol.

   
   The efficiency gains with T/TCP are only achieved if both partner systems communicating via a T/TCP client/server application support the T/TCP protocol.

3.4.10 Starting servers with inetd

One of the daemons provided with the UNIX system is *inetd*, the so called Internet super-server. *inetd* is invoked at boot time by the Service Access Controller, and determines the services for which it is to listen from the file */etc/inetd.conf*. Once this information has been read and a pristine environment created *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

The characteristics of BSD and STREAMS sockets are described in the Section "The behavior of daemons".

inetd then performs a *select()* on these sockets so that connection requests from clients can be received. When a connection request is made, *inetd* executes an *accept()* call for the socket in question and, by means of *fork()* and *dup()*, creates two file descriptors (0 (stdin) and 1 (stdout)) for the socket. The corresponding service is then started by means of *exec()*.

The use of *inetd* considerably simplifies the tasks of a server, as *inetd* takes care of most of the communication work required in establishing a connection. A server invoked by *inetd* expects the file descriptors 0 and 1 to refer to the socket connected to the client. The server can then immediately execute operations such as *read()*, *write()*, *send()*, or *recv()*. Indeed, servers may use buffered I/Os as provided by the stdio conventions, provided they use *fflush()*.

The *getpeername()* call can be important for application programmers developing servers started by means of *inetd*. This returns the address of the peer connected via the socket. For example, to log the Internet address in "dot notation" (e.g., 128.32.0.4) of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof name;
...
if (getpeername(0,
(struct sockaddr *)&name, &namelen) < 0) {
syslog(LOG_ERR, "getpeername: %m");
```

```
exit(1);
} else
syslog(LOG_INFO, "Connection from %s",
inet_ntoa(name.sin_addr));
...
```

3.4.11 Behavior of SO_LINGER

When a linger timeout was set in Reliant UNIX versions earlier than Version 5.44 B00, the following behavior resulted:

Streams	When the timeout expires any data that is still buffered is discarded and the connection with FIN is terminated.
BSD	When the timeout expires and data is still buffered, the FIN is delayed until the data can be sent. If the data cannot be sent within the retransmission timeout, the connection is terminated and an RST is sent.

Systems with a "fallback" protocol stack solution

With Reliant UNIX 5.45 and later, both protocol stacks behave in the same way, i.e. the data is discarded after the linger timeout expires and the connection with the FIN is terminated. *close()* does not return an error in any of the possible situations.

Systems with an "enhanced" protocol stack solution

After the linger timeout expires, *close()* is terminated but data that has not yet been transferred is not discarded and the connection is not aborted. An abortive release of the connection only takes place if:

- A retransmission timeout indicates that the partner can no longer be reached.
- The connection is in persist mode, i.e. the partner can still be reached, but is blocked (which means that data already received cannot be processed) and a persist timeout begins.

  The value for the persist timeout can be set using the *sysctl* variable *net.inet.tcp.persistdrop*.
- The abortive release of all TCP connections in *close()* is requested with the *sysctl* variable *net.inet.tcp.abort_linger*.

In these cases the data is discarded and an RST is generated. A counterpart that is still available may receive information regarding the incomplete data transfer in this case.

In the case of BSD sockets, *close()* returns an error EAGAIN when the linger timeout expires. This makes it possible for the local application to respond to the event "linger timeout". Despite this error, *close()* is successfully executed, i.e. access to the corresponding file descriptor and the socket linked to it is no longer possible.

In the case of STREAMS sockets, *close()* does not return an error.

3.5 Comparison of the sockets and TLI interfaces

This section compares developing communication applications using the functions of the socket interface with developing them using the functions of the TLI interface. It also indicates the differences between BSD sockets and STREAMS sockets.

Both TLI and sockets routines are defined in terms of communications paths identified by file descriptors. These file descriptors are known as transport endpoints for TLI and as sockets for the socket interface. In most cases, there are parallel routines for each transport function. For example, the TLI routine *t_open()* returns a file descriptor that identifies a transport endpoint; the routine *socket()* returns a file descriptor that identifies a socket. The table at the end of this section shows the parallels among TLI and sockets interface routines.

The section focuses on the areas in which there is no direct correspondence between TLI and sockets routines. The examples first show code for the socket interface and then show how to write the programs for the TLI.

At the end of the section, the differences between STREAMS and BSD sockets are presented in tabular form.

Application programs that use sockets must be compiled and linked with the socket library:

```
cc prog.c -dy -lsocket -lnsl
```

3.5.1 Connection mode

Both TLI and sockets support two distinct types of service: connection-oriented and connectionless.

Establishing connections with sockets: the code for the client

When creating a socket, the type of service must be specified (for example, SOCK_STREAM, SOCK_DGRAM, SOCK_RAW). The service type determines whether connection-oriented or connectionless semantics are used. For a simple example of connection establishment, consider the client side of a connection-oriented application, as in the next example. It must initiate a connection by first creating a stream socket and then using the *connect()* call to establish communication with a preexisting socket on a server machine.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
int sock;
struct sockaddr_in server;
struct hostent *hp, *gethostbyname();
struct servent *sp, *getservbyname();
/* create socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
perror("opening stream socket");
exit(1);
}
/* bind socket using name specified by command line */
server.sin_family = AF_INET;
hp = gethostbyname(argv[1]);
if (hp == 0) {
fprintf(stderr, "%s: unknown host\n", argv[1]);
exit(2);
}
```

```
memcpy((char *)&server.sin_addr, (char *)hp->h_addr,
hp->h_length);
sp = getservbyname(argv[2], "tcp");
if (sp == 0) {
fprintf(stderr, "%s: unknown service\n", argv[2]);
exit(3);
}
server.sin_port = sp -> s_port;
if (connect(sock, (struct sockaddr *)&server,
sizeof server ) < 0) {
perror("stream socket bound");
exit(1);
}
}
```

Notice the calls to *gethostbyname()* and *getservbyname()*. These are routines which access name and address databases. They are described in more detail in the Section "Connectionless data transfer". They take a host and service name, respectively, and return the host Internet address and the service port number. The service port number can be thought of as a machine-specific service address. Certain well-known services are assumed to have "privileged" TCP port numbers in the 1-to-1023 range. In some applications these port numbers are precoded; these do not call *getservbyname()*.

In the case of TLI applications, the *netdir_getbyname()* call is used rather than the *gethostbyname()* and *getservbyname()* calls and precoded TCP port numbers.

If the target socket exists and is prepared to handle a connection, the connection will complete successfully and the program can begin to send messages. Messages will be delivered in order without message boundaries.

The connection is destroyed when both sockets are closed.

Some transport services hold the connection open briefly in case more data are sent. The user may also have directed the system to wait. For more information, see the discussion of the SO_LINGER option for *getsockopt* in the "Network Reference Manual".

Establishing TLI connections: client code

Connection-oriented TLI transport services work in a similar way to stream sockets. The data is transferred over an established connection in a reliable, sequenced manner. Typical TLI client code is shown in the next example:

```
#include <stdio.h>
#include <netdir.h>
#include <netconfig.h>
extern int t_errno;
main()
{
int fd;
struct netconfig *nconf;
void *handlep;
/*
* select an appropriate transport service
*/
if ((handlep = setnetpath()) == NULL) {
nc_perror("Error in initializing networks");
exit(1);
}
/*
* try all transport services until finding one that matches
* the users stated preferences
*/
while ((nconf = getnetpath(handlep)) != NULL) {
if (nconf->nc_semantics == NC_TPI_COTS)
break;
}
if (nconf == NULL) {
fprintf(stderr, "no transport services available\n");
exit(1);
}
if ((fd = t_open(nconf->nc_device, O_RDWR, NULL))<0){
t_error("t_open failed");
exit(2);
}
if (t_bind(fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(3);
}
endnetpath(handlep);
}
```

The selection of the transport service is used by TLI applications to find the special file name associated with the requested transport protocol. The special file name that matches the protocol is passed to *t_open()*. *t_open()* then returns a file descriptor that identifies a new transport endpoint, and optionally (by way of its third argument), the default characteristics of the transport provider associated with that endpoint (and indirectly specified by the first argument). *t_bind()* then binds the new transport endpoint to the transport address contained in its second argument. The typical client doesn't care what its own address is because no other process will try to access it. The second and third arguments in the example are therefore NULL.

Establishing socket connections: server code

Connection establishment for a server process is slightly different. The process must bind itself to an address and wait for clients to connect to it. The next example shows how a sockets server is bound to its known

address:

```
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>
#include        <stdio.h>
#define         SRV_PORT    2
main(argc,argv)
int     argc;
char    *argv[];
{
int             sock;
struct sockaddr_in  server;
int             msgsock;
char              buf[BUFSIZ];
/* create socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
perror("opening stream socket");
exit(1);
}
```

```
/* name socket */
server.sin_family = AF_INET
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = SRV_PORT;
if (bind(sock, (struct sockaddr *)&server), sizeof server)<0)
{
perror("binding stream socket");
exit(2);
}
/* the server now does a listen and accept */
}
```

In the example, the server explicitly asks to be bound to port SRV_PORT.

Establishing TLI connections: server code

The equivalent code for a TLI server process is shown in the next example:

```
#include <stdio.h>
#include <fcntl.h>
#include <netconfig.h>
#include <netdir.h>
#include <tiuser.h>
#define  SRV_ADDR 2
main(argc,argv)
int      argc;
char     *argv[];
{
struct nd_hostserve     hostserv;
int                  fd;
char                 buf[BUFSIZ];
struct netconfig       *nconf;
struct t_bind          *bind;
void                 *handlep;
extern int                 t_errno;
if ( argc != 3 ) {
fprintf(stderr, "USAGE: %s host service\n", argv[0]);
exit(1);
}
hostserv.h_host = argv[1];
hostserv.h_serv = argv[2];
if ((handlep = setnetpath()) == NULL) {
nc_perror("setnetpath");
exit(2);
}
/*
* select an appropriate transport and
* get address for remote host/service
*/
while ((nconf = getnetpath(handlep)) != NULL) {
if (nconf->nc_semantics == NC_TPI_COTS)
break;
}
if (nconf == NULL) {
fprintf(stderr, "no connection mode transport\n");
exit(3);
```

```
}
if ((fd = t_open(nconf->nc_device, O_RDWR, NULL)) < 0) {
t_error("t_open failed");
fprintf(stderr, "unable to open %s\n", nconf->nc_device);
exit(4);
}
endnetpath(handlep);
if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL)) ==□
NULL) {
t_error("t_alloc of t_bind structure failed");
exit(5);
}
/*
* for simplicity in this example, assume the
* address is an integer
*/
bind->qlen = 1;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;
/* 2nd arg NULL -> bind to any address */
if (t_bind(fd, bind, bind) < 0) {
t_error("t_bind failed");
exit(6);
}
/* was the correct address bound? */
if (*(int *)bind->addr.buf != SRV_ADDR) {
t_error("t_bind bound wrong address");
exit(7);
}
/* the server now does a listen and accept */
}
```

This example shows two significant differences between sockets and TLI. The first is that TLI server applications work with any transport service. They use names for transport service selection and address assignment in order to be protocol independent; socket applications, on the other hand, use fixed addresses.

A second difference is in the behavior of the TLI and sockets bind routines when an address is invalid or unavailable. The sockets *bind()* routine fails. The TLI *t_bind()* routine may bind to another address instead. For this reason, TLI servers should check that the address returned by *t_bind()* as its third argument is correct.

Connectionless mode

Connectionless transport services, in contrast to connection-oriented services, are message-oriented and support transfer in self-contained units (datagrams) with no necessary logical relationship to each other. Sockets and TLI both provide connectionless service.

To transfer a datagram, the information required is made available to the transport provider together with the data to be transferred. This happens in a single function call. Each data packet transmitted is entirely self-contained, and can be independently routed by the transport provider.

Datagrams under the socket interface

The differences between socket library datagrams and the connectionless service provided by TLI parallel the differences between sockets and TLI connection-oriented service described above. The next example gives the code necessary to send an Internet domain datagram to a receiver whose host and service names are given as command line arguments.

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

```
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "It was the best of times. It was the worst of times."
/*
* The form of the command line is:
* dgramsend hostname servicename
*/
main(argc, argv)
int argc;
char *argv[];
{
```

```
struct servent *sp, *getservbyname();
int sock;
struct sockaddr_in name;
struct hostent *hp, *gethostbyname();
/* create socket on which to send */
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
perror("opening datagram socket");
exit(1);
}
/*
* Find the socket to send to.  gethostbyname() returns a
* structure including the network address of the
* specified host.  The port
* number is taken from the command line.
*/
hp = gethostbyname(argv[1]);
if (hp == 0) {
fprintf(stderr, "%s: unknown host\n", argv[1]);
exit(2);
}
memcpy((char *)&name.sin_addr, (char *)hp->h_addr,
hp->h_length);
name.sin_family = AF_INET;
sp = getservbyname(argv[2], "tcp");
if (sp == 0) {
fprintf(stderr, "%s: unknown service\n", argv[2]);
exit(3);
}
server.sin_port = sp->s_port;
/* send message */
if (sendto(sock, DATA, sizeof DATA, 0,
(struct sockaddr *)&name, sizeof name) < 0)
perror("sending datagram message");
close(sock);
exit(0);
}
```

The program looks up the host Internet address and the service port (both given on the command line) by calling *gethostbyname()* and *getservbyname()*. The host Internet address and service port number are in the structures returned by these two library routines. They are copied into the structure that specifies the destination of the message.

TLI datagrams

TLI connectionless service is functionally similar to sockets datagram service. The sockets address management routines *gethostbyname()* and *getservbyname()* are replaced by *netdir_getbyname()* for both connection-oriented and connectionless service.

The TLI code in the next example sends a datagram to a receiver whose host and service names are given on the command line:

```
#include <stdio.h>
#include <fcntl.h>
#include <netconfig.h>
#include <netdir.h>
#include <tiuser.h>
```

```
/* the form of the command line is:
* dgramsend hostname servicename
*/
main(argc,argv)
int      argc;
char     *argv[];
{
int        fd;
struct nd_hostserve   hostserv;
struct nd_addrlist    *addrs;
struct netconfig      *nconf;
struct t_unitdata     *ud;
void                  *handlep;
extern int      t_errno;
if ( argc != 3 ) {
fprintf(stderr, "USAGE: %s host service\n", argv[0]);
exit(1);
}
hostserv.h_host = argv[1];
hostserv.h_serv = argv[2];
if ((handlep = setnetconfig()) == NULL) {
nc_perror ("setnetconfig failed");
exit(1);
}
/*
* select an appropriate transport and
* get address for remote host/service
*/
while ((nconf = getnetconfig(handlep)) != NULL) {
if (nconf->nc_semantics == NC_TPI_CLTS &&
netdir_getbyname(nconf, &hostserv, &addrs) == 0) {
break;
}
}
if (nconf == NULL) {
fprintf(stderr,
"no address for host %s service %s\n", argv[1], argv[2]);
exit(2);
}
if ((fd = t_open(nconf->nc_device, O_RDWR, NULL)) < 0) {
t_error("t_open failed");
fprintf(stderr, "unable to open %s\n", nconf->nc_device);
exit(3);
}
if (t_bind(fd, NULL, NULL) < 0) {
t_error("t_bind failed");
exit(4);
}
if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA, □
T_ALL)) == NULL ) {
t_error("t_alloc of t_unitdata structure failed");
exit(5);
}
/*
```

```
* use first address returned by netdir_getbyname()
* filldata() will fill in the data to be sent
*/
ud->addr = addrs->n_addrs;
filldata(ud->udata);
endnetconfig(handlep);
/* send the datagram */
if (t_sndudata(fd, ud) <0 ) {
t_error("t_sndudata failed");
exit(6);
}
exit(0);
}
```

For more information about the functions *t_open*, *t_bind* and *t_sndudata* see the description of *t_open*, *t_bind*, and *t_sndudata* in the "Network Reference Manual".

3.5.2 Synchronous and asynchronous modes

Transport services are inherently asynchronous, with events occurring independently of the actions of the transport user. For example, a user may be sending data over a transport connection when an asynchronous disconnection request arrives. The user must somehow be informed that the connection has been broken. Both the socket interface and TLI provide an asynchronous mode for managing such events. Asynchronous mode is most useful for applications that expect long delays between events and have other tasks that they can perform in the meantime.

A socket is put into asynchronous mode by calling *fcntl()* and specifying O_NDELAY or O_NONBLOCK. Once in asynchronous mode, all relevant primitives — *send()*, *read()*, etc. — return EWOULDBLOCK whenever they encounter situations that would have caused them to block if they had been in synchronous mode.

The TLI non-blocking mode is also specified with the O_NDELAY or O_NONBLOCK flag in *fcntl()*. The O_NDELAY and O_NONBLOCK flags can be used when the transport provider is initially opened with the *t_open()* function, or later with the *fcntl()* call. If the TLI blocking mode is used, these cause the error code, EAGAIN to be returned.

There are different levels of asynchronous operation. Specifying O_NDELAY or O_NONBLOCK puts a socket into non-blocking mode. For true asynchronous operation, however, it is also necessary to test for asynchronous events. Socket-based applications normally use *select* to test for asynchronous events.

TLI –based applications should use *poll* to test for asynchronous events. *select()* is supported only for compatibility with older applications.

Both TLI and sockets provide mechanisms for asynchronous event notification. Sockets uses *fcntl()* to request that the system issue a SIGIO signal when it becomes possible to perform I/O on a given file descriptor. TLI uses the I_SETSIG in a *ioctl* call. This causes the system to send the process a SIGPOLL signal when the I/O event specified actually occurs. The TLI mechanism allows users to specify precisely the type of I/O operation they want to be signaled on (see the description of *streamio()* in the "Network Reference Manual").

A process that issues functions in synchronous mode must still be able to recognize certain asynchronous events immediately and act on them if necessary. Eight such asynchronous events are specified for TLI and cover both connection-oriented and connectionless modes (see *t_look* in the "Network Reference Manual"). TLI routines that encounter trouble return the special transport error TLOOK. The user can then use the *t_look()* function to identify the event that generated the error. Alternatively, the transport user can use *t_look()* to poll the transport endpoint periodically for asynchronous events. If a sockets function encounters trouble, the primitive will return an *errno* value directly.

3.5.3 Error handling

TLI attempts to separate communications errors from system errors by defining two levels of errors:

- Library level errors

  Each library function has one or more error returns and indicates failure with a –1. The variable *t_errno* of type *int* holds the specific error number when such an failure occurs. This value is set when errors occur but is not cleared by successful library calls. It should therefore be tested only after an error has been indicated. A diagnostic function *t_error()* is provided for printing out information on the current transport error.

- System errors

  The external variable *errno* reports system errors. Such errors can, of course, affect TLI functioning. When they do *t_errno* is set to TSYSERR and *errno* is set to indicate the specific system error that occurred. The state of the transport provider may change if a transport error occurs.

  The socket interface provides a similar facility with *getsockopt()* when called with an option of SO_ERROR.

3.5.4 Tabular comparison

The next table shows some approximate TLI/–sockets equivalents. The comment field describes the differences. Where there is no comment, either the functions are the same or there is no equivalent function in one or the other interface.

TLI function	Socket function	Comments
t_open()	*socket()*	
-	*socketpair()*	
t_bind()	*bind()*	*t_bind* sets the maximum queue length for pending connections; *bind()* does not. Under sockets, this is specified in the *listen()* call.
t_optmgmt()	*getsockopt()*, *setsockopt()*	*t_optmgmt()* handles only transport options. *getsockopt()* and *setsockopt()* can handle options at the transport layer, the socket layer or any other protocol layer.
t_unbind()	-	
t_close()	*close()*	
t_getinfo()	*getsockopt()*	*t_getinfo()* returns information about the transport layer. *getsockopt()* can return information about the options of the transport layer and socket layer.
t_getstate()	-	
t_getname()	*getsockname()*	
t_sync()	-	
t_alloc()	-	
t_free()	-	
t_look()	-	*getsockopt* with the SO_ERROR option returns the same kind of error information as *t_look()*.
t_error()	*perror()*	
t_connect()	*connect()*	A *connect()* can be done without first binding the local socket. The transport endpoint must be bound before calling *t_connect()*. A *connect()* can be done on a connectionless socket to set the default address for datagrams.
t_rcvconnect()	-	
t_listen()	*listen()*	*t_listen()* waits for connection requests; *listen()* merely sets the length of the queue.
t_accept()	*accept()*	
t_snd()	*send()*, *sendto()*,	*sendto()* and *sendmsg()* operate in

	sendmsg()	connection mode as well as datagram mode.
t_rcv()	*recv()*, *recvfrom()*, *recvmsg()*	*recvfrom()* and *recvmsg()* operate in connection mode as well as datagram mode.
t_snddis()	-	
t_rcvdis()	-	
t_sndrel()	*shutdown()*	
t_rcvrel()	-	
t_sndudata()	*sendto()*, *sendmsg()*	
t_rcvudata()	*recvfrom()*, *recvmsg()*	
t_rcvuderr()	-	
read(), *write()*	*read()*, *write()*	In TLI, you must push the *tirdwr* module before calling *read()* or *write()*; in sockets, it is sufficient just to call *read()* or *write()*.

Table 14: Correspondences between TLF functions and Socket functions

3.6 BSD and STREAMS sockets

In Reliant UNIX Versions up to Version 5.45 the TCP/IP protocol stack was available in two variants; one based on 4.3BSD and the other based on Streams. The Streams implementation was originally a component of UNIX SVR4 (System V Release 4). For this reason there were many TCP/IP modules in both variants. The BSD stack mainly offered, in comparison with the Streams stack, better performance for the socket interface and in some particular cases offered better adaptation to the original semantics of the sockets interface, which was actually designed for a non-Streams environment. The Streams stack supported, from the user's viewpoint, not only the sockets interface, but also TLI and XTI interfaces as well as TPI interfaces (Transport Provider Interface) in the kernel.

In Reliant UNIX 5.45 the TCP/IP protocol stack has been redesigned and standardized. TCP/IP modules are based on 4.4BSD and these modules occur only once in the system. The performance of the socket interface is at least as good as the BSD stack in the previous versions. The Streams interfaces for TPI, TLI and XTI are made available at the beginning of the TCP/IP layer by means of adaptation modules.

The standardized stack in Reliant UNIX 5.45 is referred to here as the "enhanced" protocol stack. The "enhanced" protocol stack was carefully designed in order to ensure compatibility with existing applications.

In special cases where the precise behavior before Reliant UNIX Version 5.45 is required, there is an option for switching the system back to the old two-way implementation. This is referred to here as the "fallback" solution. For more details, see the Section "The "fallback" solution".

As mentioned above, in some rare cases there is a distinction between the behavior of the BSD stack implementation and the behavior of the Streams stack. In order to be able to differentiate between both variants, the terms "BSD behavior" and "BSD sockets" or "Streams behavior" and "STREAMS sockets" are used in the next sections.

To support the "enhanced" stack, changes have been made to the *libsocket* and *libxnet* libraries and to the *inetd* daemon.

The new libraries *libsocket* and *libxnet* can determine which type of kernel is currently running ("fallback" or "enhanced" protocol stack) and adapt its behavior accordingly.

The main improvements in socket behavior for the "enhanced" stack are as follows:

- the kernel processes additional socket types (also applies to *libxnet*),
- enhanced configuration options.

The following are available for monitoring the enhanced configuration options:

- the command *checksockconf* for checking the configuration,
- the command *test_enhanced* for checking which kernel is running.

3.6.1 Selecting and activating the BSD or STREAMS sockets

There are two points at which the BSD socket implementation can be made accessible to an application:

- When the application is bound:

  The application is bound with the *libbsocket* library instead of *libsocket*.

  $(CC) -o $(PROGRAM) $(LDFLAGS) $(OBJECTS) $(LIBRARIES) \
  -lbsocket -lnsl

- When the application (bound with *libsocket* or *libxnet*) is executed, as described below.

  The STREAMS socket implementation can only be selected explicitly from an application (which is bound with *libsocket* or with *libxnet*) at the time of execution.

  With the "enhanced" stack there are two options for influencing the behavior of *libsocket* (or *libxnet*) at the time of execution:

  - with the system-wide configuration file */etc/inet/socklib.conf* or
  - on a process-specific basis with the environment variable LIBSOCKET_BSD (semantically, this is used slightly differently than in previous versions).

3.6.1.1 The environment variable LIBSOCKET_BSD

Environment variables are process-specific; for this reason LIBSOCKET_BSD takes precedence over the system-wide configuration file */etc/inet/socklib.conf*. If the environment variable is set to

- Y or y, BSD sockets are requested for TCP and UDP endpoints where possible (as was the case for the "fallback" protocol stack; this means for the AF_INET family and the types SOCK_STREAM or SOCK_DGRAM).
- N or n, STREAMS sockets are requested where possible.

3.6.1.2 The configuration file /etc/inet/socklib.conf

The file */etc/inet/socklib.conf* is not used in the context of a "fallback" stack solution.

The configuration file */etc/inet/socklib.conf* only contains entries for socket types for which BSD behavior is required, because Streams behavior represents a "fallback" solution by default. For this reason, the */etc/inet/socklib.conf* file can be kept to a minimum size while still being comprehensive.

The directory */etc/inet* contains three files with a possible system configuration. The configuration file actually being used - */etc/inet/socklib.conf* - should be a copy of one of these files:

socklib.conf.default
: This file contains a standard configuration *libsocket/libxnet*. It results in BSD sockets being used for the AF_INET family and the types SOCK_STREAM and SOCK_DGRAM. This behavior corresponds to a "fallback" solution when the environment variable LIBSOCKET_BSD is set to Y.

socklib.conf.bsd
: This file contains a sample configuration for *libsocket/libxnet* in order to ensure that BSD sockets are always used for the AF_INET family.

socklib.conf.stream
: This file contains a sample configuration for *libsocket/libxnet* in order to ensure that STREAMS sockets are always used for all families.

When the system is delivered the file */etc/inet/socklib.conf* has the same contents as the file */etc/inet/socklib.conf.default*.

For more information on the file *socklib.conf*, see the "Reliant UNIX 5.45 Network Reference Manual".

3.6.1.3 Summary of configuration options

For applications using the *libsocket/libxnet* libraries, the two tables below show which socket is selected when using the environment variable LIBSOCKET_BSD (table 15) and which socket is selected when using the configuration file *socklib.conf* (table 16).

	LIBSOCKET_BSD = Y (y)	**LIBSOCKET_BSD = N (n)**
	Independent of *socklib.conf*	
TCP	BSD	STREAMS
UDP	BSD	STREAMS
ICMP	STREAMS	STREAMS
RAWIP	STREAMS	STREAMS

Table 15: Socket selection of applications when LIBSOCKET_BSD is set

A prerequisite for the use of the configuration file *socklib.conf* is that the environment variable LIBSOCKET_BSD is not set or containing an invalid value.

	socklib.conf is:		
	Copy of socklib.conf.*x*	**Different configuratio**	**Not available**

				ns	
	x=default	*x=bsd*	*x=stream*		
TCP	BSD	BSD	STREAMS	Any	BSD
UDP	BSD	BSD	STREAMS	Any	BSD
ICMP	STREAMS	BSD	STREAMS	Any	BSD
RAWIP	STREAMS	BSD	STREAMS	Any	BSD

Table 16: Socket selection of applications with the file socklib.conf

The "Different configurations" column shows the flexibility provided by the configuration file *socklib.conf*.

In terms of downwards compatibility, the behavior of a system with the "enhanced" protocol stack for applications using the environment variable LIBSOCKET_BSD is the same as that in previous system releases.

3.6.2 The commands test_enhanced and checksockconf

The enhanced configuration options of *libsocket* and *libxnet* often make it difficult to determine which type of socket is used. To make this easier for you and to check the rules that affect your environment, there are two additional commands - *test_enhanced* and *checksockconf*.

The following examples demonstrate the syntax and information provided by these commands.

$ test_enhanced

enhanced

$ pr -n2 -t /etc/inet/socklib.conf

□□1□□□□xx□
□□2□□□□# This file is a test file□
□□3□□□□#Many of the included entries as well□
□□4□□□□libsocket - - - bsd□
□□5□□□□This line should bring an error□
□□6□□□□yy□
□□7□□□□# This one not, the following empty line neither□
□□8□
□□9□□□□libsocket inet stream□
□10□□□□libsocket inet dgram

$ checksockconf

□□Check kernel ability: enhanced protocol stack supported
□□Check environment: LIBSOCKET_BSD set to Y, this implies:□
□□- libxnet: BSD sockets for AF_INET with SOCK_STREAM or□
□□□□SOCK_DGRAM, otherwise□
□□- libxnet: STREAMS sockets□
□□- libsocket: BSD sockets for AF_INET with SOCK_STREAM or□
□□□□SOCK_DGRAM, otherwise□
□□- libsocket: STREAMS sockets□□Check library configuration file /etc/inet/socklib.conf:□
□□(provided for information only, since overruled by□
□□LIBSOCKET_BSD)□□- line 1: should have between 2 an 4 words - ignored□□- line 5: doesn't apply to any library - ignored□□- line 6: should have between 2 an 4 words - ignored

□□Entries for libxnet:□□- no entry found for 'libxnet' - STREAMS sockets assumed□□Entries for libsocket:□□- line 4: family name '-' invalid - ignored□□- Valid entries:□□□□line□□9:□libsocket inet stream□□□□□□□□□□□□(BSD sockets for family AF_INET and type SOCK_STREAM)□□□□line 10:□libsocket inet dgram□□□□□□□□□□□□(BSD sockets for family AF_INET and type SOCK_DGRAM)□□-STREAMS sockets for all other combinations$

3.6.3 The behavior of daemons

A particular situation is presented by applications that are started on a server by the *inetd* daemon (e.g. *in.ftpd*). Since the *inetd* is activated by the Service Access Controller (sac) at the system start, the correct environment must already have been provided at that stage. The behavior of the daemons started with *inetd* is controlled as follows (irrespective of the value of the LIBSOCKET_BSD environment variable or the contents of the file */etc/inet/socklib.conf*):

- there is no way of ensuring that BSD or STREAMS sockets are the default for all daemons on a system-wide basis,
- service-specific entries (BSD or STREAMS) can be made in the *inetd* configuration file */etc/inetd.conf*. The default is STREAMS.

  To obtain BSD sockets for a specific service, the character string **/BSD** must be appended to the socket type as follows:

  ftp stream/BSD tcp nowait root /usr/sbin/in.ftpd in.ftpd

  It is also possible (for symmetry reasons) to specify /STREAMS as an extension:

  login stream/STREAMS tcp nowait root /usr/sbin/in.rlogind in.rlogind

  This, however, does not change the default behavior.

Exceptions

- *in.rlogind* can be called in two different modes: in compatibility mode (with the **-n** option forced) or in turbo mode (default); this requires TPI endpoints and cannot therefore use BSD sockets.

  If the file */etc/inetd.conf* contains the following line (this is **not** the case when the system is delivered), the *-n* option is forced implicitly under the assumption that the selection of BSD sockets has been carefully considered. This prevents the daemon from crashing.

  login stream/BSD tcp nowait root /usr/sbin/in.rlogind in.rlogind
- Daemons from other vendors, which are started using *inetd*, can be influenced by the contents of the file */etc/inet/socklib.conf* if (and only if) they generate new sockets in addition to those inherited from *inetd*.

  It is not technically possible to force a certain type of behavior for sockets generated by a process, without it having an affect on the corresponding subprocesses.

  Problems in this area can be solved on a service-specific basis (e.g. by explicitly setting the environment variable LIBSOCKET_BSD to **n** if the daemon requires TPI endpoints, the default setting in the file */etc/socklib.conf* is BSD and the daemon generates its own new sockets).

3.6.4 The "fallback" solution

The "fallback" solution can easily be restored on any system with an "enhanced" protocol stack by

- modifying the value of the NETWORK_LINK_DEFAULT variable in the file */etc/conf/cf.d/net.default*,
- regenerating the kernel and
- restarting the system.

The same mechanism can be used to switch back to the "enhanced" stack (modification of the NETWORK_LINK_DEFAULT variable, calling the *idbuild* command and restarting the system).

3.6.4.1 Configuration options

Within the "fallback" solution you can only influence the behavior of *libsocket* at the time of execution by setting the LIBSOCKET_BSD environment variable to **Y** or **y**. This means that BSD sockets are specifically requested

for the AF_INET family and the types SOCK_STREAM or SOCK_DGRAM wherever possible.

The behavior of *libsocket* is not influenced by any configuration file (*socklib.conf* is ignored) and nothing influences the behavior of *libxnet*.

The following table contains a summary of the configuration options for *libsocket*:

	LIBSOCKET_BSD equal to Y or y	**LIBSOCKET_BSD not equal to Y or y or is not set**	
		socklib.conf avail-able	*socklib.conf* not available
TCP	BSD	STREAMS	STREAMS
UDP	BSD	STREAMS	STREAMS
ICMP	STREAMS	STREAMS	STREAMS
RAWIP	STREAMS	STREAMS	STREAMS

Table 17: Configuration options for libsocket

3.6.4.2 The commands test_enhanced and checksockconf

In the case of a "fallback" solution, the commands *test_enhanced* and *checksockconf* may return the following information, for instance:

$ test_enhanced

fallback

$ ckecksockconf

Check kernel ability: fallback protocol stack supported
Regardless of environment:
- libxnet: STREAMS sockets

□□Check environment: LIBSOCKET_BSD set to Y, this implies:
□□- libsocket: BSD sockets for AF_INET with SOCK_STREAM or
□□□ SOCK_DGRAM, otherwise
□□- libsocket: STREAMS sockets
$

3.6.4.3 The behavior of daemons

The daemons started by *inetd* always use STREAMS sockets. This behavior can be changed and the use of BSD sockets forced by

- either generating the file */etc/saf/inetd/_config* with the following contents:

  assign LIBSOCKET_BSD=Y
- or by starting two *inetd* daemons with different configuration files.

The extended syntax in the file */etc/inetd.conf* is ignored (i.e. the extended character string /BSD or /STREAMS is not taken into account).

3.6.5 Differences between BSD and STREAMS sockets

Although the implementations of STREAMS sockets and BSD sockets are very largely compatible, there are some differences that application programmers should be aware of. These differences are described in the following table.

BSD sockets	STREAMS sockets
System calls for connection establishment	
connect()	
If *connect()* is called for an unbound socket, the protocol determines whether the socket is assigned a name (address) before the connection is established.	If *connect()* is called for an unbound socket, that socket is always assigned an address by the transport service.
A second *connect()* for a socket, after the first one failed, returns EINVAL.	A second *connect()* for a socket, after the first one failed, returns EPIPE.
accept()	
EINVAL is returned if an *accept()* UDP socket is transferred as an argument.	EOPNOTSUPP is returned if an *accept()* UDP socket is transferred as an argument.
select()	
select() sets the read bit for a waiting connection request.	*select()* sets the read and write bit for a waiting connection request.
System calls for file transfer	
sendto()	
sendto() with a broadcast address returns EACCES if broadcasting was not previously allowed with *setsockopt*(SO_BROADCAST). The datagram is not sent.	*sendto()* with a broadcast address does not return an error if broadcasting was not previously allowed with *setsockopt*(SO_BROADCAST). The datagram is not sent.
write()	
write() fails and sets *errno* to ENOTCONN if an unbound socket is specified.*write()* can be used with sockets of the type SOCK_DGRAM to send zero length data.	A *write()* call is successfully executed, but the data is lost. In this case, the SO_ERROR socket error option is set to ENOTCONN. Calling *write()* returns -1 and sets *errno* to ERANGE. Data with zero length should be sent by the functions *send()*, *sendto()* or *sendmsg()*.
read()	
A *read()* call fails and sets *errno* to ENOTCONN if *read()* is used to establish a connection via an unbound socket.	*read()* returns zero bytes if the socket is in blocking mode. If the socket is in non-blocking mode, it returns a -1 with *errno* set to EAGAIN.
sendmsg() and *readmsg()*	
If the MSG_PEEK flag has been set when *sendmsg()* is called, and access rights are available, the access rights will be copied, leaving them available	If the MSG_PEEK flag is specified in a call to *recvmsg()*, and access rights are available, the access rights will be transferred to the user buffer

for reading by a subsequent call to *recvmsg()*.	associated with the receiving socket. They are then deleted, and the transferring socket has no further access to them. They are therefore unavailable to a subsequent call to *recvmsg()*. All data to which the access permissions are assigned is copied to the user buffer and is not available to *recvmsg()*.
System calls for obtaining information	
getsockname()	
getsockname() will work when a previously existing connection has been closed.	*getsockname()* will return -1 and *errno* will be set to EPIPE if a previously existing connection has been closed.
poll()	
If *connect()* fails for a non-blocking socket because there is no listener for the server port, *poll()* sets the flag POLLOUT.	If *connect()* fails for a non-blocking socket because there is no listener for the server port, *poll()* sets the flag POLLERR.
ioctl() and *fcntl()*	
SIOCSPGRP/FIOSETOWN/F_SETOWN()	
The entries SIOCSPGRP, FIOSETOWN and F_SETOWN in an *ioctl()* call and F_SETOWN in an *fcntl()* call take as an argument a positive process id or negative process group identifying the intended recipient list of subsequent SIGURG and SIGIO signals.	This is not the case in System V, Release 4. The only acceptable arguments to these system calls is the caller's process id or a negative process group which has the same absolute value as the caller's process id. In other words, the only recipient of SIGURG and SIGIO signals is the calling process.
Local management	
bind()	
bind() uses the credentials of the calling user to determine whether the desired port number can be used.	A call to *socket()* causes the user's credentials to be remembered and used to validate addresses used in *bind()*.
setsockopt()	
setsockopt() can be used at any time during the life of a socket.	A *setsockopt()* call for a transport provider corresponding to the status diagram defined by the TPI transport provider interface fails if it is executed for a socket that is not bound to a local address. If a socket is unbound and *setsockopt()* is called, the operation is executed successfully in the AF_INET domain but fails in the AF_UNIX domain.
shutdown()	

If *shutdown* is called with the value of *how* equal to zero, further attempts to receive data will return zero bytes (EOF).	Calling *shutdown* with the value of *how* the value of how equal to zero will not cause further attempts to receive data to return zero bytes if the *read(2)* system call is used and the socket is in nonblocking mode.
If *shutdown()* is called with the value of *how* equal to 2, further attempts to receive data will return EOF. Attempts to send data will return -1 with *errno* set to EPIPE with a SIGPIPE issued.	In this case *read* will return -1 with *errno* set to EAGAIN. If one of the socket's receive calls is used, the correct result (EOF) will be returned. The same results will occur, except that attempts to send data using the *write* system call will cause *errno* to be set to EIO. As in the above case, if a socket system call is used, the correct *errno* will be returned.
Signals	
SIGIO	
SIGIO is delivered every time new data are appended to the socket input queue.	SIGIO is delivered only when data are appended to a socket queue that was previously empty.
SIGURG	
A SIGURG is delivered every time new data is anticipated or actually arrives.	A SIGURG is delivered only when there is no urgent data already pending.
S_ISSOCK	
The ISSOCK macro takes the mode of a file descriptor as an argument. It returns 1 if the descriptor represents a socket and 0 otherwise.	The ISSOCK macro is not available. The ISSOCK macro does not exist. In System V Release 4, a socket is defined as a file descriptor associated with a streams special file that contains a socket module.
S_IFSOCK()	
This file mask identifies a socket descriptor.	There is no file mask for sockets. The *#define* statement does not exist.
Miscellaneous	
If an invalid buffer is specified in a function, the function will normally return -1 with *errno* set to EFAULT. If ls -l is executed in a directory that contains a socket belonging to the UNIX domain, an *s* will be printed on	If an invalid buffer is specified in a function, a memory dump is frequently created. If ls -l is executed in a directory that contains a socket belonging to the UNIX domain, a *p* will be printed on

the left side of the mode field. Executing ls -F will cause an equals sign (=) to be printed after any filename that represents a UNIX domain socket.	the left side of the mode field. Nothing will be printed after a filename that represents a UNIX domain socket.

Table 18: Differences between BSD sockets and STREAMS sockets

3.7 Traditional sockets and Xsockets

There are two variants of the user interface for sockets, firstly the traditional "sockets" interface and secondly the "Xsockets" interface. The Xsockets interface is an extended and X/Open-standardized version of the traditional socket interface.

3.7.1 Socket libraries

Depending on whether applications use the traditional sockets or Xsockets, you must use different libraries for linking.

Applications that use traditional sockets are linked with the *libsocket* library:

cc *prog.c* -lsocket -lnsl

Applications that use Xsockets must be linked with the *libxnet* library. Depending on the default used, the instruction

```
#define _XOPEN_SOURCE 500          /* UNIX98 */
```

or

```
#define _XOPEN_SOURCE              /* UNIX95 */
#define _XOPEN_SOURCE_EXTENDED 1
```

or

```
#define _XOPEN_SOURCE              /* XPG4 without UNIX95 */
```

must be contained in the beginning of the source code. The source program can then be bound as follows:

cc *prog.c* -lxnet

If these agreements are not available in the source program, they can also be transferred during binding:

cc -D_XOPEN_SOURCE=500 *prog.c* -lxnet

or

cc -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 *prog.c* -lxnet

or

cc -D_XOPEN_SOURCE *prog.c* -lxnet

3.7.2 Compatibility of sockets and Xsockets

Most applications are linked in the same way with Xsockets and run in the same way as with traditional sockets. The differences between the two socket variants relate to

- the *msghdr* structure
- address length semantics
- error codes

The T/TCP protocol can only be used with traditional BSD sockets; T/TCP is not supported by Xsockets. See also the Section "Traditional sockets and T/TCP".

This section deals with the compatibility of Xsockets and the traditional socket interface.

The msghdr structure

To send or receive data over a network connection, many applications use the functions *send()*, *sendto()*, *write()*, *recv()* or *read()*. Some applications also use the functions *sendmsg()* and *recvmsg()* in order to use as few parameters as possible that need to be made available directly.

The functions *sendmsg()* and *recvmsg()* use the *msghdr* structure which comprises the following elements:

Data type	Sockets	Xsockets	Meaning
*void **	*msg_name*	*msg_name*	Optional address
size_t	*msg_namelen*	*msg_namelen*	Address length
*struct iovec **	*msg_iov*	*msg_iov*	Array
int	*msg_iovlen*	*msg_iovlen*	Number of elements in *msg_iov*
*void **	-	*msg_control*	Auxiliary data
size_t	-	*msg_controllen*	Size of auxiliary data buffer
int	-	*msg_flags*	Flags for the received message
caddr_t	*msg_accrights*	-	Access rights sent or received
int	*msg_accrightslen*	-	Length of access rights

Table 19: msghdr structure

The *msg_accrights* and *msg_accrightslen* fields of the traditional sockets have been replaced by the *msg_control* and *msg_controllen* fields for the Xsockets. Xsockets thus has the additional option of sending and receiving access rights for the handling of auxiliary data. Moreover, this interface definition for auxiliary data can be further extended in the future.

Address length

In the *accept*, *getpeername* and *getsockname* functions the Xsocket parameter *address_len* specifies the length of the *sockaddr* structure for input. For output *address_len* specifies the length for display of

- the address of the connecting socket
- the address of the partner on the specified socket
- the locally bound name

In the case of the traditional sockets *address_len* contains the size of the name that is returned.

Error codes

Xsockets contain three new errors codes that apply for a series of different functions:

- EOPNOTSUPP: the operation is not supported by the socket protocol
- EAFNOSUPPORT: the specified address is not valid for the address family of the specified socket
- ENOBUFS: insufficient system resources are available to complete the function call

The Online Manpages and Network Reference Manual for Xsockets describes a series of new error codes that also apply for the traditional sockets:

- EINTR: the function was interrupted by a signal
- EMFILE {OPEN_MAX}: file descriptors are opened in the calling process
- ENFILE: the maximum number of file descriptors in the system are already opened
- ECONNRESET: the connection was closed by the remote system.
- EHOSTUNREACH: the destination host cannot be reached (probably because the system is out of operation or routing is not functioning)
- ENETDOWN: the local interface used to reach the destination host is not in operation
- ENETUNREACH: there is no route to the network

For some error conditions there is more than one error code. The traditional sockets return only one error code but Xsocket applications must be able to react to each of the possible error codes.

The following table provides an overview of the differences between the error codes of traditional sockets and Xsockets:

Function	Sockets error code	Xsockets error code	Error condition
accept	EAGAIN	EAGAIN EWOULDBLOCK	O_NONBLOCK is set for the socket file descriptor so that no connections are accepted
bind	EINVAL	EDESTADDRREQ	The address argument is a null pointer (for address family AF_UNIX).
		ENOENT	A component of the pathname does not specify an existing file or the pathname is an empty string (for address family AF_UNIX).
		EISCONN	The socket is already bound.

connect	ENOTSOCK	EACCESS	Search permission for a component of the path prefix is denied, or write access to the specified socket is denied (for address family AF_UNIX).
getpeernam e	ENOTCONN	EINVAL	The socket is not bound.
listen	EINVAL	EINVAL	The socket is already bound or has been closed.
	EOPNOTSUPP	EDESTRADDRREQ	The socket is not bound to any local address and the protocol does not support any listening for connection requests for an unbound socket.
	EPROTO	EINVAL	The socket is already bound
recv, recvfrom, recvmsg	EAGAIN	EINVAL	The flag MSG_OOB is set but no out-of-band data is available
setsockopt		EINVAL	The socket was closed
send, sendto, sendmsg	EMSGSIZE	EINVAL	Only for *sendmsg*: the sum of the values for *iov_len* exceeds *size_t*
	ENOMEM	EAGAIN	Not sufficient memory available to satisfy the request
	EINVAL	EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket
	EINVAL	EDESTADDRREQ	The socket is not bound to a local address and the protocol does not support any listening for connection requests
	EINVAL	ENOTCONN	A receive attempt is made on a connection-oriented socket that is not bound

Table 20: Error codes for traditional sockets and Xsockets

3.7.3 Xsockets extensions

recv, recvfrom, recvmsg

The functions *recv*, *recvfrom* and *recvmsg* support the flag value MSG_WAITALL that serves to determine the volume of returned data.

If *recvmsg* terminates successfully, the *msg_flag* elements of the message header show special features of the message received:

MSG_EOR	"End-of-record" was received (if supported by the protocol)
MSG_OOB	Out-of-band data was received
MSG_TRUNC	Normal data was not complete
MSG_CTRUNC	Control data was not complete

getsockopt

The additional option SO_ACCEPTCONN is supported. With the help of this option it is possible to ascertain whether listening for connection requests is enabled. The result of this option stores a data type *int* and has only write permission.

shutdown

The following values have been added to *sys/socket.h*:

- SHUT_RD makes receive operations impossible
- SHUT_WR makes send operations impossible
- SHUT_RDWR makes send and receive operations impossible

4 Remote procedure calls

The remote procedure calls (RPC) mechanism allows users in the network to use procedure calls designed to hide the details of underlying networking mechanisms. RPC is transport independent and thus able to take advantage of whatever kinds of networking mechanisms may be available, e.g. TCP/IP or ISO. RPC implements a logical client–to–server communications system designed specifically for the support of network applications. Generic facilities, such as *rpcbind*, associate network services with universal network addresses.

With RPC, the client makes a procedure call that sends data packets to the server, as necessary. When these packets arrive, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

Programming with RPC produces programs that are designed to run within a client/server network model. Such programs use RPC mechanisms to avoid the details of interfacing to the network, and provide network services to their callers without requiring that the caller be aware of the existence and function of the underlying network. For example, a program can simply call *rusers()*, a C routine that returns the number of users on a remote machine. The caller is not explicitly aware of using RPC – the call to *rusers()* is as simple as a call to *malloc()*.

This chapter deals only with the C interface to RPC, but remote procedure calls can be made from any language. Note too that although this section only describes the use of RPC for communication between processes on different machines, RPC works just as well for communication between different processes on the same machine.

The following sections provide overviews of the key components and characteristics of RPC:

- "RPC versions and numbers" – RPC uses a program number, program version, procedure number tuple to uniquely identify procedures that can be called via RPC.
- "Transport service selection" – Programs can be written to operate via specific transport services and transport service types, or can be written to operate over system- or user-chosen transport services.
- "The *rpcbind* service" – *rpcbind* is a facility used to associate network services with universal network addresses.

- "The lower RPC levels" – The lower RPC levels available to client and server programs allow for greater control of RPC communications.
- "External data representation (XDR)" – Data transmitted between RPC clients and servers is encoded in XDR transfer syntax.

4.1 RPC versions and numbers

Each RPC procedure is uniquely identified by a program number, version number, and procedure number.

The program number identifies a group of RPC procedures, each of which has a different procedure number and which together form the RPC service. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number does not have to be assigned.

RPC programs must be assigned program numbers according to rules detailed in Program number assignment in the chapter entitled "XDR/RPC protocol specification" in this manual.

4.2 Transport service selection

Transport service selection is a simple means by which users may dynamically select transport services. It is based on two mechanisms:

- The file */etc/netconfig* lists the transport services available on the host and identifies them by type.
- The optional environment variable NETPATH.

To create a service for a particular transport, an application must interface to RPC at a level below the *top level*, i.e., the level composed of *clnt_create()* and its associated routines. Only then can it specify the types of transport services that it prefers. See the Section "The lower RPC levels" for details about the various RPC levels.

The */etc/netconfig* file contains several lines, each of which corresponds to an available transport. Here are some possible entries:

```
# The Network Configuration File.
#
# Each entry is of the form:
#
# network_id semantics flags protofamily protoname device
# nametoaddr_libs
# ticlts tpi_clts v loopback - /dev/ticlts /usr/lib/straddr.so
ticots tpi_cots v loopback - /dev/ticots /usr/lib/straddr.so
ticotsord tpi_cots_ord v loopback - /dev/ticotsord /usr/lib/straddr.so
tcp tpi_cots_ord v inet tcp /dev/tcp /usr/lib/tcpip.so
udp tpi_clts v inet udp /dev/udp /usr/lib/tcpip.so
icmp tpi_raw - inet icmp /dev/icmp /usr/lib/tcpip.so
rawip tpi_raw - inet - /dev/rawip /usr/lib/tcpip.so
```

At this point, we just want to mention a few details about */etc/netconfig*.

- Each entry contains a name (the first field) by which the transport is commonly known.
- Each entry also contains one or more flags (the third field) that identifies it by type - the *v* flag, for example, identifies any transport that is 'visible.'
- The last field names one or more modules that can be linked during run time that contain the name-to-address translation routines associated with the transport (see Section "Name-to-address mapping").
- The *loopback* transport services are local transports which are only available to local clients and servers. The registering of services is made more secure by the fact that *rpcbind* only allows *loopback* transports.

The format of NETPATH is simple: an ordered list of network identifiers separated by colons (:) (for example: udp:tcp:starlan). By setting NETPATH, the user can specify the order in which the application should try the various transport services. If NETPATH is not set, the system defaults to all visible transport services specified in */etc/netconfig*, in the order they appear.

Applications can choose to ignore a user's NETPATH. RPC divides selectable transport services into the following types:

netpath	Choose from those transport services that have been specified in the NETPATH environment variable. If NETPATH is not set, the system defaults to all visible transport services specified in */etc/netconfig*, in the order they appear.
" "	(null) - same as selecting *netpath*.
visible	Choose those transport services that have the visible flag (*v*) set in their */etc/netconfig* entries.
circuit_v	Same as *visible*, but restricted to connection-oriented transport services.
datagram_v	Same as *visible*, but restricted to connectionless transport services.
circuit_n	Choose from whatever is defined in NETPATH, but restrict to connectionless transport services.
datagram_n	Choose from whatever is defined in NETPATH, but restrict to connectionless transport services.
udp	(Obsolete. For backwards compatibility.) - specifies Internet User Datagram Protocol (UDP).

tcp Transmission Control Protocol (TCP).

When a transport-dependent application begins execution, it begins by calling the *setnetconfig()*, *getnetconfig()* and *endnetconfig()* routines, using them to search */etc/netconfig* for a transport of appropriate type. This information is then stored in local data structures of type *struct netconfig* and is available for later use.

setnetconfig(), *getnetconfig()*, and *endnetconfig()* are described on the *getnetconfig* manual page; the transport service selection file */etc/netconfig* is described on the *netconfig* manual page.

Taken together, these mechanisms allow a fine degree of control over transport service selection: a user can specify a preferred transport, and if it is reasonable, applications will use it. In cases where the specified transport is inappropriate (as, for example, when a remote server does not support a specified transport) the application should automatically try others with the right characteristics.

4.2.1 Name-to-address mapping

Each transport has an associated set of routines that convert between universal network addresses (string representations of transport addresses) and their local address representation. These universal addresses are passed around within the RPC system (for example, between *rpcbind* and a client). When any programming interface to the transport layer is made, a transport-specific name-to-address translation routine is called to convert the universal address into local form. Each transport has associated with it a run-time loadable library that contains the name-to-address translation routines associated with it. The library names are listed in */etc/netconfig*. The main translation routines are:

netdir_getbyname	Assigns host names and services to a transport-specific address or a set of addresses.
netdir_getbyaddr	Assigns transport-specific addresses to hosts and services.
uaddr2taddr	Converts universal addresses to transport-specific addresses.
taddr2uaddr	Converts transport-specific addresses to universal addresses.

For more details on these routines, see the description of *netdir* in the "Network Reference Manual".

4.3 The rpcbind service

Client programs need to be able to locate server programs, to be precise, they need a way of obtaining the addresses of server programs. RPC is transport-independent and therefore makes no assumptions regarding the structure of a network address. It works with universal addresses which are merely defined to be character strings terminated by the ASCII character NULL. RPC converts these universal addresses to local transport addresses using routines that are specific to the particular transport. For instance, in the case of services that operate on TCP/IP, the universal addresses are converted to IP addresses and TCP port numbers. You will find more details of these routines in the description of *netdir*.

Operating systems provide (different) mechanisms which enable a process with a network address to wait for incoming messages. Messages are sent to the transport address from which the receiving process fetches them. The value of transport addresses is that the recipients of messages can be specified independently of the conventions used by the receiving operating system. The *rpcbind* protocol specifies a network service which enables clients to use a standard method of obtaining the transport addresses of those RPC services that are supported by the server.

4.3.1 Address registration

In the Internet domain, *rpcbind* is always assigned the port number 111 (TCP and UDP). Unfortunately, this simple solution is not acceptable on all transport services.

rpcbind registers its own address on each of the transport services supported by the host. *rpcbind* is the only RPC service whose address must be known. The address must be well-known for a given transport because

rpcbind is responsible for registering the addresses of other RPC services and making those addresses available to RPC clients. Thus, services make their addresses available to clients by registering their addresses with their host's *rpcbind* daemon. Thereafter, the addresses of the services are available to *rpcinfo* and to programs using library routines specified in *rpcbind*.

RPC-based services do not usually receive their transport addresses until run time, and they then register with *rpcbind*. Neither they nor their clients can make any assumptions about what those transport addresses will be. *rpcbind* is started by the system administrator. Both server programs and client programs use *rpcbind* services.

Although client and server programs and client and server machines are usually distinct, they need not be. A server program can also be a client program, as when an NFS server uses an *rpcbind* server. Likewise, when a client program directs a remote procedure call to its own machine, the machine acts as both client and server.

As part of its initialization, a server program communicates with its host's *rpcbind* daemon to register itself in the host's registered-address map. Whereas server programs communicate with *rpcbind* to update address maps, clients communicate with it to request those maps. To find a remote program's address, a client sends an RPC call message to a server machine's *rpcbind* daemon; if the remote program is on the server, the daemon returns the relevant address in an RPC reply message. The client program can then send RPC call messages to that address.

The *rpcbind* protocol (for details, see the Chapter "XDR/RPC protocol specification") provides a procedure, RPCBPROC_CALLIT(), with which *rpcbind* can assist a client in making a remote procedure call. A client program passes the target procedure's program number, version number, procedure number (for a discussion of these numbers, see the chapter entitled Programming with RPC) and arguments in an RPC call. *rpcbind* then looks up the address of the desired RPC service and sends an RPC call with the arguments received from the client to the target procedure.

When the target procedure returns results, RPCBPROC_CALLIT() passes them on to the client program. It also returns the target procedure's address so the client can later call it directly.

The RPC library provides an interface to all *rpcbind* procedures. Some of the RPC library procedures also call *rpcbind* automatically for client and server programs.

4.3.2 The rpcinfo command

The command *rpcinfo* outputs the current RPC registration information for the daemon *rpcbind*; administrators can use it to delete registrations. *rpcinfo* can be used to find all the RPC services registered on a specified host and to report their universal addresses and the transport services for which they are registered. It can also be used to call (*ping*) a specific version of a specific program on a specific host using a specific transport, and to report whether a response is received. For details, see *rpcinfo* in the "Networking Reference Manual".

4.4 The lower RPC levels

There are various levels at which it is possible to interface to the RPC library services. These levels are described in detail in the section entitled Section "Programming with RPC". Understanding the lower levels of RPC is helpful but not necessary if you plan to use *rpcgen* to generate your RPC applications. The use of *rpcgen* is described in the section entitled rpcgen. In Figure 2 you can see the lower-level interfaces on the client side, that are available for creating an object for the transport services; Figure 3 shows the creation of an object for transport services for an RPC server. Note the similarities in both hierarchies.

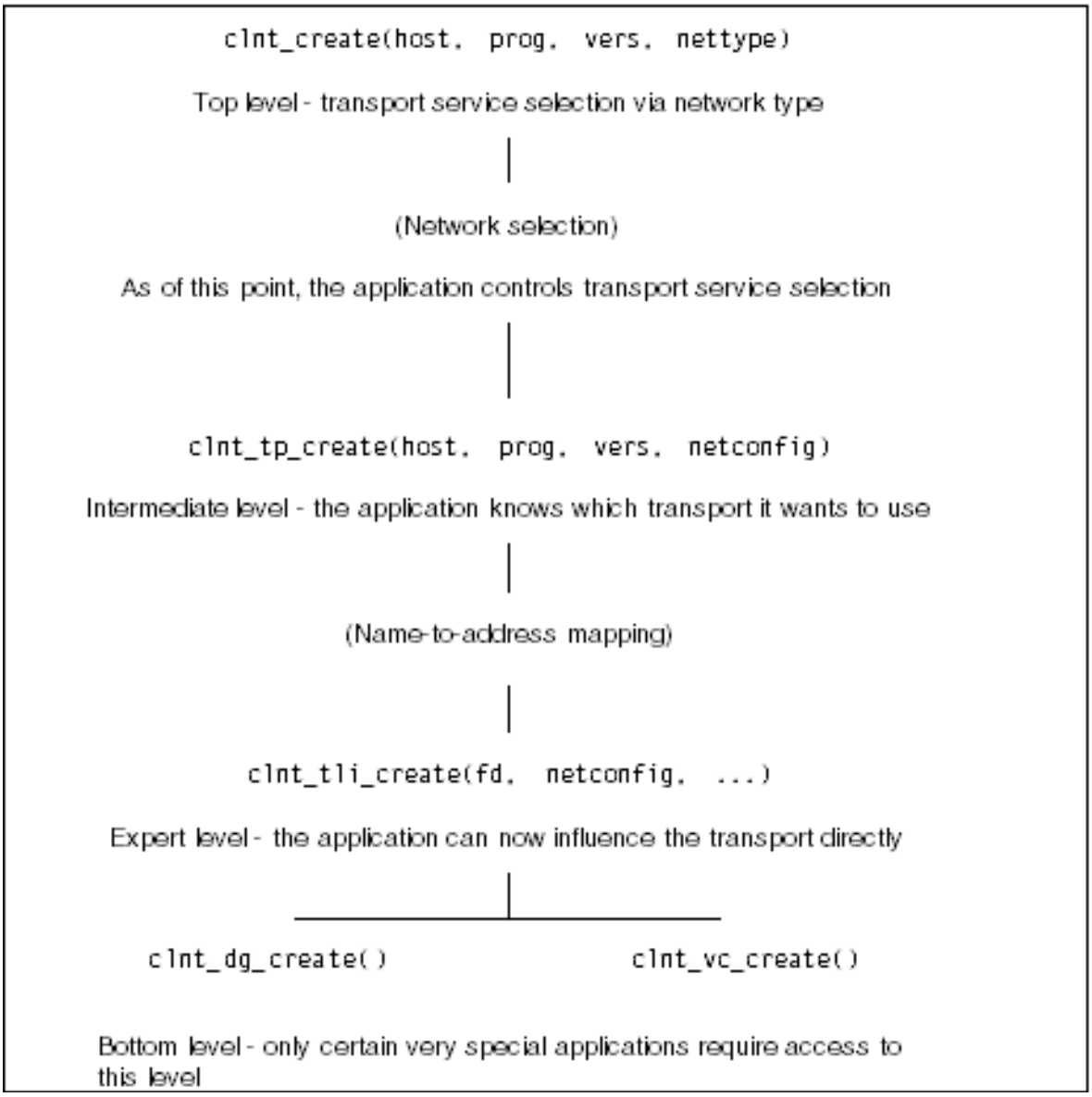


Figure 2: Lower levels of RPC, client end

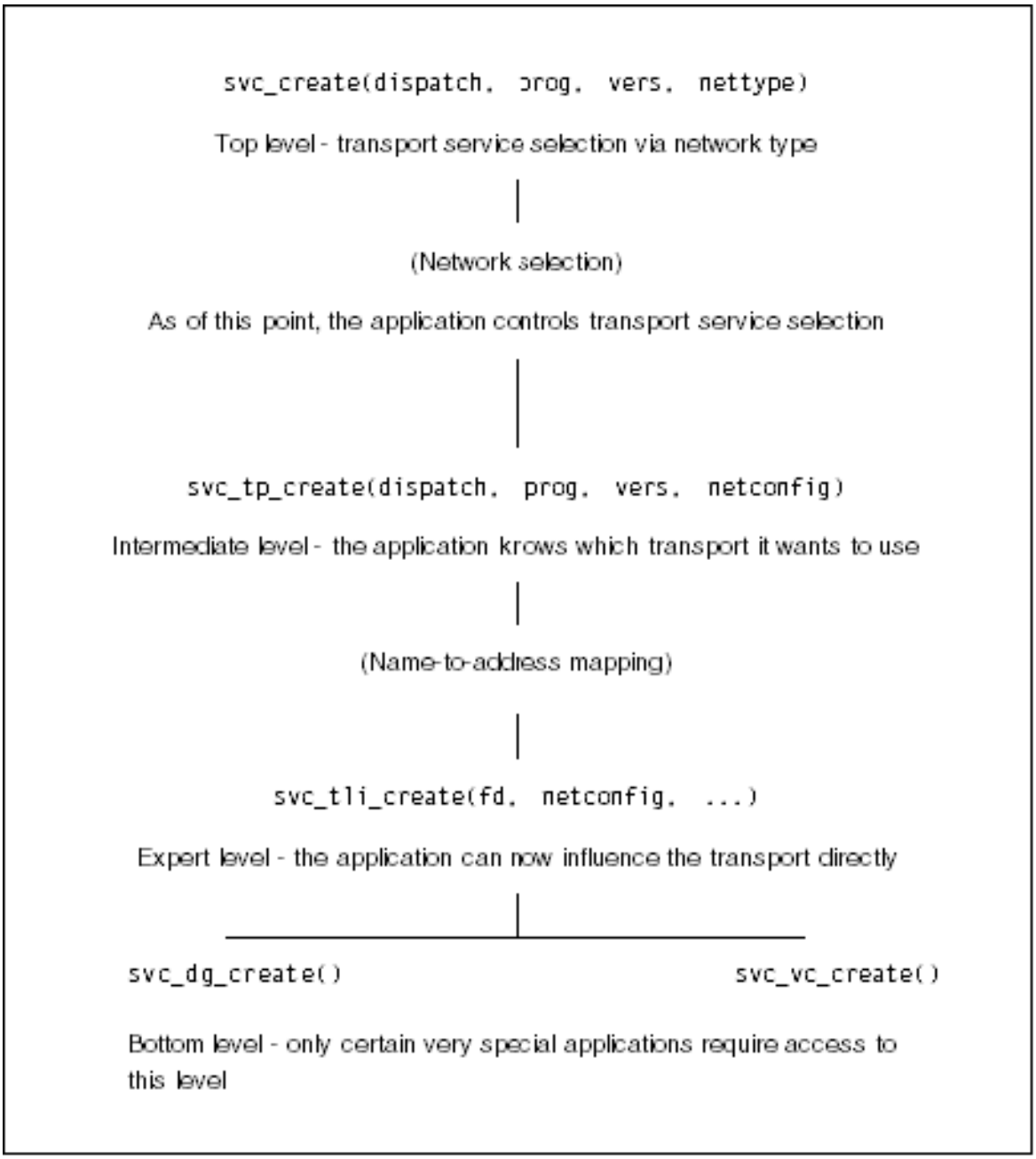


Figure 3: Lower levels of RPC, server end

4.5 External data representation (XDR)

RPC uses XDR (External Data Representation), a protocol for the machine-independent encoding of data. XDR is useful for transferring data between different computer architectures.

Regardless of different machines' byte orders or representation structures, RPC converts data to XDR representation when it is sent. The process of converting from a machine-specific representation to XDR format is called encoding, and the reverse process is called decoding.

For a discussion of XDR, see Chapter "XDR/RPC protocol specification"

4.6 rpcgen

rpcgen is a compiler. It accepts a remote program interface definition written in a language known as an RPC language. The RPC language is similar to C. *rpcgen* generates RPC programs in C from the definitions written in the RPC language. This output includes:

- client routines as stub functions
- a server skeleton
- XDR filter routines for both parameters and results
- a header file that contains common definitions
- (optionally) dispatch tables that the server can use to check authorizations and then invoke service routines.

rpcgen's output files can be compiled and linked in the usual way. The client stubs interface with the RPC library and effectively hide the transport from their callers. The server skeleton similarly hides the transport from the server procedures that are to be invoked by remote clients.

The developer writes server procedures (in any language that observes system calling conventions) and links them with the server skeleton produced by *rpcgen* to get an executable server program. To use a remote program, you write an ordinary main program that makes local procedure calls to the client stubs produced by *rpcgen*. Linking this program with stubs produced by *rpcgen* creates an executable program. (At present the main program must be written in C.)

rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server skeleton.

rpcgen reduces the development time that would otherwise be spent encoding routines on the lower RPC levels and subsequently debugging them. For speed-critical applications, though, *rpcgen* allows you to mix low-level code with high-level code. Hand-written routines can be linked with the *rpcgen* output without any difficulty. However, the *rpcgen* output can also be taken as a starting point and rewritten, as necessary. For a discussion of RPC programming without *rpcgen*, see the section entitled Programming with RPC.

4.6.1 Converting local procedures into remote procedures

Assume an application that runs on a single machine. Suppose we want to convert it to run over the network. Here we will show such a conversion by way of a simple example program that prints a message to the console. The source file for the original program might look like:

```
/* printmsg.c: print a message on the console */
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
char *message;
if (argc != 2) {
fprintf(stderr, "usage: %s <message>\n", argv[0]);
exit;
}
```

```
message = argv[1];
if (!printmessage(message)) {
fprintf(stderr, "%s: couldn't print your message\n",
argv[0]);
exit;
}
printf("Message Delivered!\n");
exit(0);
}
```

```
/* Print a message to the console. */
/* Return a boolean indicating whether the message was actually printed.
*/
printmessage(msg)
char *msg;
{
FILE *f;
f = fopen("/dev/console", "w");
if (f == NULL) {
return (0);
}
fprintf(f, "%s\n", msg);
fclose(f);
return (1);
}
```

On the local machine, this program could be compiled and executed as follows:

```
$ cc printmsg.c -o printmsg
$ printmsg "Hello, there."
Message delivered!
$
```

If the *printmessage()* function were turned into an RPC procedure, it could be called from anywhere in the network. It is not difficult to convert a procedure into an RPC procedure.

In the context of RPC programming, it has become acceptable to use the term "procedure" to refer to a C-language **function**. The terms are used interchangeably in this description.

In general, it is necessary to figure out what the types are for all procedure inputs and outputs. Here, the procedure *printmessage()* takes a string as input, and returns an integer as output. Knowing this, we can write a protocol specification in RPC language that describes the RPC version of *printmessage()*. The RPC language source code for such a specification would look like this:

```
/* msg.x: Remote message printing protocol */
program MESSAGEPROG {
version MESSAGEVERS {
int PRINTMESSAGE(string) = 1;
} = 1;
} = 0x20000001;
```

RPC procedures are always declared as being part of an RPC program. The example above is the declaration for an entire RPC program that contains a single procedure: PRINTMESSAGE.

In this example, the PRINTMESSAGE procedure is declared to be procedure 1, in version 1 of the RPC program whose number is 0x20000001. Refer to Program number assignment in the Chapter "XDR/RPC protocol specification" for guidance on the assignment of program numbers.

All RPC services contain a 0 procedure; this operation does nothing except be successfully executed. Calling a 0 procedure is also referred to as executing a *ping* on an RPC program. Pinging is used to verify the existence of an RPC service.

Using *rpcgen*, no null procedure (procedure 0) need be written because *rpcgen* generates it automatically.

Notice that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is *string* and not *char* as it would be in C. This is because a *char* in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a *string*.

There are just two more things to write:

- the RPC procedure itself
- the main client program that calls it

Here's one possible definition of an RPC procedure to implement the PRINTMESSAGE procedure we declared above:

```
/* msg_proc.c: implementation of the RPC procedure */
* "printmessage" */
#include <stdio.h>
#include <rpc/rpc.h   /* always needed  */
#include "msg.h"      /* msg.h will be generated by rpcgen */
/*
/* Remote version of "printmessage" */
*/
```

```
int *      printmessage_1(msg)
char **msg;
{
static int result;  /* must be static! */
FILE *f;
f = fopen("/dev/console", "w");
if (f == NULL) {
result = 0;
return (&result);
}
fprintf(f, "%s\n", *msg);
fclose(f);
result = 1;
return (&result);
}
```

Notice that the declaration of the RPC procedure *printmessage_1()* differs from that of the local procedure *printmessage()* in three ways: It takes a pointer to a string instead of a string itself. This is true of all RPC procedures: they always take pointers to their arguments rather than the arguments themselves.

It returns a pointer to an integer instead of an integer itself. This is also characteristic of RPC procedures: they return pointers to their results.

When *rpcgen* is used, it is essential to have *result* (in this example) declared as *static*.

In the code generated by *rpcgen*, the *result* address is converted to XDR format *after* the RPC procedure returns. If the result were declared local to the RPC procedure, references to its address would be invalid after the RPC procedure returned. So the result *must* be declared *static* when *rpcgen* is used.

It has _1 appended to its name. In general, all RPC procedures called by *rpcgen* are named by the following rule: the procedure name in the program definition (here PRINTMESSAGE is converted to all lower-case letters, an underbar (_) is appended to it, and the version number (here 1) is appended.

The last thing to do is declare the main client program that will call the RPC procedure. Here is one possibility:

```
/*
* rprintmsg.c: RPC version of "printmsg.c"
*/
#include <stdio.h>
#include <rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* msg.h will be generated by rpcgen */
main(argc, argv)
int argc;
char *argv[];
{
CLIENT *cl;
int *result;
char *server;
char *message;
if (argc != 3) {
fprintf(stderr,
"usage: %s server message\n", argv[0]);
exit;
}
/*
* Save values of command line arguments
*/
server = argv[1];
message = argv[2];
/*
* Create client "handle" used for calling MESSAGEPROG on the
* server designated on the command line.
*/
cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS,
"visible");
if (cl == NULL) {
/*
* Couldn't establish connection with server.
* Print error message and die.
*/
clnt_pcreateerror(server);
exit;
}
/*
* Call the RPC procedure "printmessage" on the server
*/
result = printmessage_1(&message, cl);
if (result == NULL) {
```

```
/*
* An error occurred while calling the server.
* Print error message and die.
*/
clnt_perror(cl, server);
exit;
}
/*
* Okay, we successfully called the RPC procedure.
*/
if (*result == 0)
{
/*
* Server was unable to print our message.
* Print error message and die.
*/
fprintf(stderr, "%s: %s couldn't print your message\n",
argv[0], server);
exit;
}
/*
* The message got printed on the server's console
*/
printf("Message delivered to %s!\n", server);
exit(0);
}
```

There are four things to note here:

- First a client handle is created using the RPC library routine *clnt_create()*. The client handle is a communication end point for RPC clients similar to a client socket. This client handle will be passed to the stub routines that call the remote procedure. (The client handle can be created in other ways as well, see the Section "Programming with RPC" for details.)
- The last parameter to *clnt_create()* is *visible*, which specifies that any transport noted as *visible* in */etc/netconfig* can be used.
- When the RPC procedure *printmessage_1()* is called in *msg_proc.c*, there is another argument for the client handle. It also returns a pointer to the result instead of the result itself.
- The RPC procedure can fail in two ways. The RPC mechanism itself can fail or, there can be an error in the execution of the remote procedure. In the former case, the remote procedure (in this case *printmessage_1()*) returns with a NULL. In the later case, however, the details of error reporting are application dependent. Here, the error is being reported via **result*.

Here's how to put all the pieces together:

```
$ rpcgen msg.x
$ cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
$ cc msg_proc.c msg_svc.c -o msg_server -lnsl
```

Two programs are compiled here: the client program *rprintmsg* and the server program *msg_server*.

Before doing this, *rpcgen* was used to fill in the missing pieces.

Here is what *rpcgen* (called without any flags) did with the input file *msg.x*:

1. It created a header file called *msg.h* that contained *#define* statements for MESSAGEPROG, MESSAGEVERS and PRINTMESSAGE for use in the other modules.
2. It created the client "stub" routines in the *msg_clnt.c* file. Here there is only one, the *printmessage_1()* routine, that was called from the *rprintmsg* client program. If the name of an *rpcgen* input file is *FOO.x*, the client

stubs output file is called *FOO_clnt.c*.

3. It created the server program in *msg_svc.c* that calls *printmessage_1()* from *msg_proc.c*. The rule for naming the server output file is similar to the previous one: for an input file called *FOO.x*, the output server file is named *FOO_svc.c*.

Once created, the server should be copied to a remote machine and run. (If the machines are homogeneous, the server can be copied as a binary. Otherwise, the source files will need to be copied to and compiled on the remote machine.) For this example, the remote machine is called remote and the local machine is called local. The server is started from the shell on the remote system:

remote$ msg_server

Server processes, like *msg_server*, created with *rpcgen* always run in the background. It is not necessary to start the server in the background with an ampersand (&). Servers generated by *rpcgen* can also be invoked with port monitors like *listen* and *inetd*, instead of from the command line.

Thereafter, a user on local can print a message on the console of system remote as follows:

local$ rprintmsg remote "Hello, there."

Using *rprintmsg*, a user can print a message on any system console (including the local console) if the server *msg_server* is running on the target system.

4.6.2 Generating XDR routines with rpcgen

The previous example illustrated the automatic generation of client and server RPC code. *rpcgen* may also be used to generate XDR routines, i.e., the routines necessary to convert local data structures into XDR format and vice-versa.

This example presents a complete RPC service: a remote directory listing service, built using *rpcgen* not only to generate stub routines, but also to generate the XDR routines.

Here is the protocol description file:

```
/*
* dir.x: Remote directory listing protocol
*/
const MAXNAMELEN = 255;              /* maximum length of a
* directory entry */
*/
typedef string nametype<MAXNAMELEN>;  /* a directory entry */
typedef struct namenode *namelist;    /* a link in the
* listing
*/
/*
* A node in the directory listing
*/
struct namenode {
nametype name;         /* name of directory entry */
namelist next;         /* next entry */
};
/*
* The result of a READDIR operation.
*/
union readdir_res switch (int errno) {
case 0:
namelist list;  /* no error: return directory
* listing
*/
default:
void;     /* error occurred: nothing else to return*/
```

```
};
/*
* The directory program definition
*/
```

```
program DIRPROG {
version DIRVERS {
readdir_res
READDIR(nametype) = 1;
} = 1;
} = 0x20000076;
```

Types (like *readdir_res* in the example above) can be defined using the *struct*, *union* and *enum* keywords. These keywords should not be used in later declarations of variables of those types. For example, if you define a union with the name *foo*, you should declare using only *foo* and not *union foo*.

Running *rpcgen* on *dir.x* creates four output files. First are the basic three described previously: those containing the header file, client stub routines and server skeleton.

The fourth contains the XDR routines necessary for converting instances of declared data types from host platform representation into XDR format, and vice-versa. These routines are output in the file *dir_xdr.c*.

For each data type used in the *.x* file, *rpcgen* assumes that the RPC/XDR library contains a routine whose name is the name of the data type, prepended by *xdr_* (e.g. *xdr_int*). If a data type is defined in the *.x* file, then *rpcgen* generates the required *xdr_* routine.

If there are no such data types definitions in an RPC source file (e.g. *msg.x*), then an *_xdr.c* file will not be generated.

If you want to write a *.x* source file that uses a data type not supported by the RPC/XDR library, and you want to deliberately omit defining the type (in the *.x* file); you must provide the*xdr_* routine. This is a way for you to provide your own customized *xdr_* routines. See the Section "Programming with RPC" for more details on passing arbitrary data types.

Here is the server-side implementation of the READDIR procedure.

```
/*
* dir_proc.c: remote readdir implementation
*/
#include <rpc/rpc.h>     /* Always needed */
#include <dirent.h>
#include "dir.h" >       /* Created by rpcgen */
extern int errno;
extern char *malloc();
extern char *strdup();
readdir_res *readdir_1(dirname)
nametype *dirname;
```

```
{
DIR *dirp;
struct dirent *d;
namelist nl;
namelist *nlp;
static readdir_res res;  /* must be static! */
/*
* Open directory
*/
dirp = opendir(*dirname);
if (dirp == NULL) {
res.errno = errno;
return (&res);
}
/*
* Free previous result
*/
xdr_free(xdr_readdir_res, &res);
/*
* Collect directory entries.
* Memory allocated here will be freed by xdr_free
* next time readdir_1 is called
*/
nlp = &res.readdir_res_u.list;
while (d = readdir(dirp)) {
nl = *nlp = (namenode *) malloc(sizeof(namenode));
nl->name = strdup(d->d_name);
nlp = &nl->next;
}
*nlp = NULL;
/*
* Return the result
*/
res.errno = 0;
closedir(dirp);
return (&res);
}
```

Here is the client-side program to call the server:

```
/*
* rls.c: Remote directory listing client
*/
#include <stdio.h>
#include <rpc/rpc.h>        /* always need this */
#include "dir.h"            /* will be generated by rpcgen */
extern int errno;
main(argc, argv)
int argc;
char *argv[];
{
CLIENT *cl;
char *server;
char *dir;
readdir_res *result;
namelist nl;
if (argc != 3) {
fprintf(stderr, "usage: %s host directory\n",
argv[0]);
exit;
}
/*
* Remember what our command line arguments refer to
*/
server = argv[1];
dir = argv[2];
/*
* Create client handle used for calling MESSAGEPROG on the
* server designated on the command line.  We tell the RPC
* package to use any visible transport when contacting the
* server. */
cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
if (cl == NULL) {
/*
* Couldn't establish connection with server.
* Print error message and die.
*/
clnt_pcreateerror(server);
exit(1);
```

```
}
/*
* Call the remote procedure readdir on the server
*/
result = readdir_1(&dir, cl);
if (result == NULL) {
/*
* An error occurred while calling the server.
* Print error message and die.
*/
clnt_perror(cl, server);
exit(1);
}
/*
* Okay, we successfully called the remote procedure.
*/
if (result->errno != 0) {
/*
* A remote system error occurred.
* Print error message and die.
*/
errno = result->errno;
perror(dir);
exit(1);
}
/*
* Successfully got a directory listing.
* Print it out.
*/
for (nl = result->readdir_res_u.list; nl != NULL;
nl = nl->next) {
printf("%s\n", nl->name);
}
exit(0);
}
```

Again using the hypothetical systems named local and remote, the files can be compiled and run as follows:

```
remote$  rpcgen dir.x
remote$  cc -c dir_xdr.c
remote$  cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
remote$  cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
remote$  dir_svc
```

After installing an executable copy of *rls* on system *local*, a user on that system can list the contents of */usr/share/lib* on system remote as follows:

```
local$  rls remote /usr/share/lib
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

4.6.3 Using preprocessing directives

The *rpcgen* compiler supports C and other preprocessing features.

C preprocessing is performed on *rpcgen* input files before they are compiled. All C preprocessing directives are legal within a *.x* file. Five symbols may be defined by *rpcgen*, depending on the type of output file being generated. The symbols are:

Symbol	Usage
RPC_HDR	For header-file output
RPC_XDR	For XDR routine output
RPC_SVC	For server-skeleton output
RPC_CLNT	For client stub output
RPC_TBL	For index table output

The *rpcgen* compiler provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly into the output file, without any interpretation of the line.

The % feature is not always useful, owing to a limitation: The *rpcgen* compiler may not place the lines where the you intended them.

Here is a simple example that illustrates *rpcgen* preprocessing features:

```
/*
* time.x: Remote time protocol
*/
program TIMEPROG {
version TIMEVERS {
unsigned int TIMEGET(void) = 1;
} = 1;
} = 0x20000044;
#ifdef RPC_SV
%int *
%timeget_1()
%{
%   static int thetime;
%
%   thetime = time(0);
%   return (&thetime);
%}
#endif
```

4.7 Common RPC programming techniques

This section suggests some encoding and *rpcgen* usage techniques.

Transport types	*rpcgen* can produce code for specific transport types (or even specific transport services).
Timeout changes	Client default timeout periods can be changed.
Authentication	Clients may authenticate themselves to servers; interested servers can examine client authentication information.
define statements	C preprocessing symbols can be defined in *rpcgen* command lines
Broadcasts	Servers need not send NULL replies to broadcasts.
Port monitor support	A port monitor rather than the RPC server can monitor the server port.
Dispatch tables	Programs can access server dispatch tables
Debugging	Clients and servers created with *rpcgen* can be linked and run on a single system for debugging purposes.

4.7.1 Transport types

The *rpcgen* compiler takes optional arguments that allow you to specify a desired transport type or even a specific network identifier. For details about transport service selection, see the Section "Programming with RPC".

The *-s* flag creates a server that responds to requests on all transport services of a specified type. For example, the invocation

```
rpcgen -s datagram_n prot.x
```

writes a server to standard output that uses all the connectionless transport services specified in the NETPATH environment variable (or in */etc/netconfig*, if NETPATH is not defined or does not specify any connectionless transport services).

Similarly, the *-n* flag creates a server that responds only to requests via the transport service specified by a

single transport identifier.

Be careful using servers created by *rpcgen* with the *-n* flag. Because network identifiers are host-specific, the server produced may not run on other hosts in the expected way.

4.7.2 Timeout changes

After sending a request to the server, a client program waits for a default amount of time (25 seconds) to receive a reply. This timeout may be changed using the *clnt_control()* routine (see the description of *rpc* in the "Networking Reference Manual).

Here is a small code fragment to illustrate the use of *clnt_control()*:

```
struct timeval tv;
CLIENT *cl;
cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "visible");
if (cl == NULL) {
exit(1);
}
tv.tv_sec = 60;  /* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

4.7.3 Client authentication

The client create routines do not, by default, have any facilities for client authentication, but the client may sometimes want (or be required) to authenticate itself to the server. Doing so is trivial, and looks about like this:

```
CLIENT *cl;
cl = client_create("somehost", SOMEPROG, SOMEVERS, "visible");
if (cl != NULL) {
/* To set AUTH_SYS style authentication */
cl->cl_auth = authsys_createdefault();
}
```

Servers can evaluate this authentication information. For example, getting authentication information is important to servers that want to achieve some level of security. This extra information is supplied to the server in an RPC (see the structure of *svc_req* in the Section "Authentication").

Here is an example of a remote procedure whose server checks client authentication information. This is a rewrite of the *printmessage_1()* that is developed in the Section "rpcgen". The rewritten procedure will only allow root users to print a message to the console:

```
int *
printmessage_1(msg, rq)
char **msg;
struct svc_req   *rq;
{
static int result;   /* Must be static */
FILE *f;
struct authsys_parms *aup;
aup = (struct authsys_parms *)rq->rq_clntcred;
if (aup->aup_uid != 0) {
result = 0;
return (&result);
}
/*
* Same code as before.
*/
}
```

4.7.4 Define statements in the command line

The *rpcgen* compiler provides a means for defining C preprocessing symbols and assigning values to them from the command line. Command-line define statements can, for example, be used to compile conditional code. This means the code is compiled only when the DEBUG symbol is defined. For example:

```
$ rpcgen -DDEBUG proto.x
```

4.7.5 Server response to broadcasts

When a procedure is known to have been called via broadcast RPC, and the called procedure determines that it cannot provide the client with a useful response, it is usually best for the server to send no reply back to the client. This reduces network traffic.

To prevent the server from replying, a remote procedure can return NULL as its result. The server code generated by *rpcgen* will detect this and not send out a reply.

Here is an example of a procedure that replies only if it is possible to access the */etc/exports* file:

```
void *
reply_if_nfsserver_1()
{
char notnull;   /* just here so we can use its
* address
*/
if (access("/etc/exports", F_OK) < 0) {
return (NULL);  /* prevent RPC from replying */
}
/*
* assign notnull a non-null value so
* RPC will send out a reply
*/
return ((void *)&notnull);
}
```

If a procedure returns type *void **, it must return a non-NULL pointer if it wants RPC to send a reply.

4.7.6 Port monitor support

Port monitors such as *inetd* and *listen* can monitor ports for specified RPC services. Whenever a request comes in for a particular service, the port monitor spawns a server process to provide for it. After the call has been serviced, the server can exit. This has the advantage of conserving system resources: fewer blocked processes waiting for work.

It may be useful for services to wait for a specified interval after satisfying a service request, on the chance that another request will follow. If there is no call within the specified time, the server will exit and some port monitors, like *inetd*, will continue to monitor for the server. If a later request for the service occurs, the port monitor will give the request to a waiting server process (if any), rather than spawning a new process.

When monitoring for a server, some port monitors, like *listen*, always spawn a new process in response to a service request. If it is known that a server will be used with such a monitor, the server should exit immediately on completion.

By default, services created using rpcgen wait for 120 seconds after servicing a request before exiting. You can, however, change that interval with the -K flag.

```
$ rpcgen -K 20 proto.x
```

Here the server will wait only for 20 seconds before exiting. To create a server that exits immediately, -K 0 can be used. To create a server that never exits (a normal server), the appropriate argument is -K -1.

All servers generated by *rpcgen* assume the following support from port monitors:

- The name of the transport service is passed through the environment variable NLS_PROVIDER.
- The connection is passed via the file descriptor 0.

See "Using port monitors" in the Section "Using port monitors" for a further discussion of port monitors.

4.7.7 Dispatch tables

It is sometimes useful for programs to have access to the dispatch tables used by the RPC package. For example, the server dispatch routine may need to check authorization and then invoke the service routine; or a client library may want to deal with the details of storage management and XDR data conversion.

When invoked with the *-T* option, *rpcgen* generates RPC dispatch tables for each program defined in the protocol description file, *proto.x*, in the file *proto_tbl.i*. For the sample protocol description file, *dir.x*, given in the section entitled Section "Generating XDR routines with rpcgen", a dispatch table file created by *rpcgen* would be called *dir_tbl.i*. The suffix *.i* stands for "index".

Each entry in the dispatch table is a *struct rpcgen_table*, defined in the header file *proto.h* as follows:

```
struct rpcgen_table {
char       *(*proc)();
xdrproc_t   xdr_arg;
unsigned    len_arg;
xdrproc_t   xdr_res;
unsigned    len_res;
};
```

where

proc	is a pointer to the service routine,
xdr_arg	is a pointer to the input (argument) *xdr_* routine,
len_arg	is the length in bytes of the input argument,
xdr_res	is a pointer to the output (result) *xdr_* routine, and
len_res	is the length in bytes of the output result.

The table, named *dirprog_1_table* for the *dir.x* example, is indexed by procedure number. The variable *dirprog_1_nproc* contains the number of entries in the table.

An example of how to locate a procedure in the dispatch tables is shown by the routine *find_proc()*:

```
struct rpcgen_table *
find_proc(proc)
long        proc;
{
if (proc >= dirprog_1_nproc)
/* error */
else
return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, that service routine is usually not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the *rpcgen* service routine initializer is RPCGEN_ACTION(proc_ver).

This way, the same dispatch table can be included in both the client and the server. Use the following *define* when compiling the client:

```
#define RPCGEN_ACTION(routine)  0
```

and use this define when compiling the server:

```
#define RPCGEN_ACTION(routine) routine
```

4.7.8 Debugging with rpcgen

When programming with *rpcgen*, the client program and the server procedure can be tested together as a single program by linking them with each other rather than with the client and server stubs. To do this, calls to RPC library routines (e.g. *clnt_create()*), have to be commented out, and client-side routines have to call server routines directly. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger. After the program is working, the client program can be linked to the *rpcgen*-created client stubs and the server procedures can be linked to the *rpcgen*-created server skeleton.

4.8 RPC language reference

The RPC language is an extension of the XDR language. The sole extension is the addition of the *program* and *version* types.

The RPC language is similar to the C language. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

4.8.1 Definitions

An RPC language file consists of a series of definitions.

definition-list:
definition ;
definition ; definition-list

It recognizes six types of definitions:

Definition:
enum-definition
const-definition
typedef-definition
struct-definition
union-definition
program-definition

4.8.2 Enumerations

RPC/XDR enumerations have the same syntax as C enumerations.

enum-definition:
enum *enum-ident* {
enum-value-list
}

enum-value-list:
enum-value
enum-value, enum-value-list

enum-value:
enum-value-ident
enum-value-ident = value

Here is a short example of an RCP/XDR *enum*, and the C *enum* to which it gets compiled.

```
enum colortype {          enum colortype {
RED = 0,                  RED = 0,
GREEN = 1,
→              GREEN = 1,
BLUE = 2                  BLUE = 2,
};                        };
typedef enum colortype colortype;
```

4.8.3 Constants

RPC/XDR symbolic constants may be used wherever an integer constant is used, for example, in array size specifications.

const-definition:
const *const-ident* = *integer*

For example, the following defines a constant, DOZEN, equal to 12.

```
const DOZEN = 12;
→  #define DOZEN 12
```

4.8.4 Typedefs

RPC/XDR typedefs have the same syntax as C typedefs.

typedef-definition:
typedef *declaration*

Here is an example that defines an *fname_type* used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>;
→ typedef char *fname_type;
```

4.8.5 Declarations

In RPC/XDR, there are four kinds of declarations:

- *simple* declaration
- *fixed-array* declaration
- *variable-array* declaration
- *pointer* declaration

A *simple* declaration corresponds to a simple C declaration:

Simple declaration:

type_identifier variable_identifier

Example:

```
colortype color;
→ colortype color;
```

A *fixed-array* declaration (declaration of arrays of fixed length) is like a C array declaration:

Fixed-array declaration:

type_identifier variable_identifier [*value*]

Example:

```
colortype palette[8];
→ colortype palette[8];
```

For a *variable-array* declaration (a declaration of arrays of variable length) RPC/XDR use angle brackets.

variable-array declaration:

type_identifier variable_identifier < *value* >

type_identifier variable_identifier < >

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;     /* at most 12 items */
int widths<>;        /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into *struct* declarations. For example, the *heights* declaration gets compiled into the following *struct*:

```
struct {
u_int heights_len;   /* number of items in array */
int *heights_val;    /* pointer to array         */
} heights;
```

Note that the number of items in the array is stored in the *_len* component and the pointer to the array is stored in the *_val* component. The first part of each of these component's names is the same as the name of the declared RPC/XDR variable.

pointer declarations are made in RPC/XDR exactly as they are in C. Address pointers cannot really be sent over the network, but RPC/XDR pointers are useful for sending recursive data types such as lists and trees. The type is actually called "optional-data", not "pointer", in XDR language.

pointer-declaration:

*type-ident *variable-ident*

Example:

listitem *next;
→ listitem *next;

4.8.6 Structures

An RPC/XDR *struct* is declared almost exactly like its C counterpart. It looks like the following:

struct-definition:

struct *struct-ident* {
declaration-list
}

declaration-list:

declaration ;
declaration ; declaration-list

As an example, here is an RPC/XDR structure for a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {          struct coord {
int x;
→          int x;
int y;                  int y;'
};                    };
typedef struct coord coord;
```

The output is identical to the input, except for the added *typedef* at the end of the output. This allows one to use *coord* instead of *struct coord* when declaring items.

4.8.7 Unions

RPC/XDR unions are discriminated unions, and look different from C unions. They are more analogous to Pascal variant records than they are to C unions.

union-definition:

union *union-ident* switch (*simple declaration*) {
case-list
}

case-list:

case *value* : *declaration* ;
case *value* : *declaration* ; *case-list*
default : *declaration* ;

Here is an example of a type that might be returned as the result of a read data operation: if there is no error, return a block of data; otherwise, don't return anything.

```
union read_result switch (int errno) {
case 0:
opaque data[1024];
default:
void;
};
```

It gets compiled into the following:

```
struct read_result {
int errno;
union {
char data[1024];
}  read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output structure has the name as the type name, with an additional trailing *_u*.

4.8.8 Programs

RPC/XDR programs are declared using the following syntax:

program-definition:

program *program-ident* {
version-list
} = *value*

version-list:

version ;
version ; version-list

version:

version *version-ident* {

procedure-list
} = *value*

procedure-list:
procedure
procedure ; procedure-list

procedure:
type-ident procedure-ident (*type-ident*) = *value*

For example:

```
/*
* time.x: Get or set the time.  Time is represented as
* seconds since 0:00, January 1, 1970.
*/
program TIMEPROG {
version TIMEVERS {
unsigned int TIMEGET(void) = 1;
void TIMESET(unsigned) = 2;
} = 1;
} 0x20000044;
```

This file compiles into these *#defines* in the output header file:

```
#define TIMEPROG 0x20000044'
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

4.8.9 Special cases

There are a few exceptions to the rules described above.

C has no built-in boolean type. However, the RPC library uses a boolean type called *bool_t* that can be either TRUE or FALSE. Things declared as type *bool* in the RPC/XDR language are compiled into *bool_t* in the output header file.

Example:

```
bool married;
→ bool_t married;
```

C has no built-in string type either. Instead, it uses the null-terminated *char* * declaration. In the RPC/XDR language, strings are declared using the *string* keyword, and compiled into type *char* * in the output header file.

The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the NULL character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;
→     char *name;
string longname<>;
→     char *longname;
```

Data whose type is not further defined is referred to in RPC and XDR as "opaque data". It it is declared as byte sequences of fixed or variable length.

Examples:

```
opaque diskblock[512];
```

```
→ char diskblock[512];
opaque filedata<1024>;
→ struct {
u_int filedata_len;
char *filedata_val;
} filedata;
```

In a *void* declaration, the variable is not named. The declaration is just *void* and nothing else. "void" declarations can only occur in two places: "union" definitions and "program" definitions (as the argument or result of a remote procedure).

4.9 Programming with RPC

The RPC package provides a multi-level application programming interface for development of network applications using remote procedure calls.

At the simplified interface (the highest level), the package provides great transparency, but offers only limited control over the underlying communications mechanisms. Program development with this simplified interface is rapid, and it is directly supported by the *rpcgen* compiler.

The interfaces to the lower levels of the RPC package provide increasing control over RPC communications. Programs that use them pay for the power in terms of greater complexity of code. Effective programming at the lower levels requires knowledge of computer network fundamentals.

In order of increasing control and complexity, these levels are called the top level, intermediate level, expert level and bottom level.

This chapter is intended for programmers who wish to write network applications using remote procedure calls, and who want to use or understand the RPC mechanisms usually hidden by the *rpcgen* protocol compiler.

4.9.1 An overview of the RPC package

The RPC interface can be seen as being divided into several distinct levels. The highest level is general, and provides for no fine control of any kind. The lower levels (four can be usefully distinguished) are available for use as necessary, and provide increasingly detailed levels of control.

For a complete specification of the routines in the RPC library, see the *rpc* and related manual pages.

The simplified interface

Here, you doesn't need to consider the characteristics of the underlying transport or operating system details. You simply make remote procedure calls to routines on other machines, and need specify only the type of transport that they wish to use. The selling point here is simplicity. It is this level that allows RPC to pass the "hello world" test - that simple things should be simple. The routines at this level are used for most applications.

Included in the simplified interface are only three basic RPC routines:

rpc_reg() *rpc_reg()* registers a routine as an RPC routine and obtains a unique, system-wide procedure-identification number for it.

rpc_call() *rpc_call()* executes the RPC call in full.

rpc_broadcast() Like *rpc_call()*, except that it broadcasts its call message across all transport services of the specified type.

The top level

At the top level, the interface is still simple, but you do have to create client and server handles before making a call. Like the routines in the simplified interface, the routines here require a *nettype* argument that specifies the transport.

The top level consists of two routines for creating handles:

clnt_create() Creates a client handle. The programmer tells *clnt_create()* where the server is located and the type of transport to use to get to it.

svc_create() Creates server handles for all the transport services of the specified *nettype*.The programmer tells *svc_create()* which dispatch routine is to be used. The dispatch routine of an RPC service is a function that, on the basis of the request parameters, branches to the procedure responsible for the request.

The simplified interface and the top level of RPC, while simple, are not very flexible. They do not allow the choice of a specific transport. At these levels, all routines just take a *nettype* argument, which serves to define the class of transport to be used. On the client side, programs do transport service selection, and hence may be efficient, depending on *nettype*. On the server side, programs may have to listen on many transport services, and hence may waste system resources.

In both of these cases, however, efficiency can be improved by judicious definition of the NETPATH environment variable. If you wish the application to run on all transport services, this is the interface that should be used.

The intermediate level

The RPC intermediate level and the two levels below it allow many details to be controlled by the programmer, and for that reason their use is necessary for special applications. Programmers should only go down to the level necessary for the control needed. Programs written at these lower levels are more complicated, but also more efficient.

The intermediate level differs from the two levels above it in that it allows you to specify directly the transport to be used. It consists of two routines:

clnt_tp_create() Creates a client handle for a specified transport.

svc_tp_create() Creates a server handle for a specified transport.

The expert level

The expert level consists of a larger set of routines with which you can specify more parameters, but those parameters are still all directly transport related. It includes the following routines:

clnt_tli_create() Creates a client handle for a specified transport, allowing fine control of the client characteristics.

svc_tli_create() Creates a server handle for a specified transport, allowing

	fine control of the server characteristics.
rpcb_set()	Provides an interface to *rpcbind* that establishes the assignment of an RPC service to a network address.
rpcb_unset()	Deletes an assignment established by *rpcb_set*.
rpcb_getaddr()	Provides a programmatic interface to *rpcbind*, one that returns the transport address of a specified RPC service.
svc_reg()	Associates a given program and version number pair with a given dispatch routine.
svc_unreg()	Destroys an association of the type established by *svc_reg*.

The bottom level

The bottom level consists of routines called when you require full control, even down to the smallest details of the transport options. It consists of the following routines:

clnt_dg_create()	Creates an RPC client handle for the specified RPC program, using a connectionless transport service.
svc_dg_create()	Creates an RPC server handle, using a connectionless transport service.
clnt_vc_create()	Creates an RPC client handle for the specified RPC program, using a connection-oriented transport service.
svc_vc_create()	Creates an RPC server handle, using a connection-oriented transport service.

4.10 The simplified interface to RPC

The section entitled Section "RPC library-based network services" describes the use of existing RPC services by means of C functions contained in the RPC library.

Some RPC services are available as RPC programs rather than in the form of C functions. Section "Remote procedure call and registration" describes the fundamentals of using these services and creating new services.

4.10.1 RPC library-based network services

Imagine writing a program that needs to know how many users are logged into a remote machine. This can be done by calling an RPC library routine, *rnusers()*, as illustrated below:

```
#include <stdio.h>
/*
* a program that calls rnusers()
*/
main(argc, argv)
int argc;
char **argv;
{
int num;
if (argc != 2) {
fprintf(stderr, "call: %s hostname\n", argv[0]);
exit;
}
if ((num = rusers(argv[1])) < 0) {
fprintf(stderr, "error: rusers\n");
exit;
}
printf("%d users on %s\n", num, argv[1]);
exit(0); }
```

For *rnusers()* to work, the *rpc.usersd* daemon must be running on the remote machine. RPC library routines such as *rusers()* are in the RPC services library *librpcsvc.a*.

Thus, the program above should be compiled with

```
$ cc program.c -lrpcsvc -lnsl
```

Here are some of the RPC service library routines available to the C programmer:

rnusers(), *rusers()*	Return information about users on remote machine
rwall()	Write to specified remote machines
spray()	Spray packets to a specific machine

4.10.2 Remote procedure call and registration

The simplest interface to the RPC functions is based on the routines *rpc_call()*, *rpc_reg()*, and *rpc_broadcast()*. These functions provide direct access to the RPC facilities, and are appropriate for programs that do not require fine levels of control.

Using the simplified interface, the number of remote users can be gotten as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
/*
* a program that calls the RUSERSPROG RPC program
*/
main(argc, argv)
int argc;
char **argv;
{
unsigned long nusers;
int clnt_stat;
```

```
if (argc != 2) {
fprintf(stderr, "usage: rusers hostname\n");
exit();
}
if (clnt_stat = rpc_call(argv[1],
RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
xdr_void, 0, xdr_u_long, &nusers, "visible") != 0) {
clnt_perrno(clnt_stat);
exit(1);
}
printf("%d users on %s\n", nusers, argv[1]);
exit(0);
}
```

The rpc_call() routine

The simplest way of making RPC calls is with the RPC library routine *rpc_call()*. It has nine parameters.

- The first is the name of the remote server machine.
- The next three parameters are the program, version, and procedure numbers. Together, they identify the RPC procedure to be called.
- The fifth and sixth parameters are an XDR filter for encoding the procedure arguments and a pointer to the arguments that must be passed to the RPC procedure.
- The next two parameters are an XDR filter for decoding the results returned by the RPC procedure and a pointer to the place where the procedure's results are to be stored.
- Finally, there is the *nettype* specifier, the transport service type.

Arguments and results consisting of several parts are embedded in structures. If *rpc_call()* completes successfully, it returns zero; otherwise, it returns a nonzero value. The return codes (of type *enum clnt_stat*, cast to an *int* in the previous example) are found in *<rpc/clnt.h>*.

Because data types may be represented differently on different machines, *rpc_call()* needs both the XDR filter and the pointer to the RPC argument. The same applies to the result. For RUSERSPROC_NUM, the return value is of the type *unsigned long*, so *rpc_call()* has *xdr_u_long()* as its first return parameter, which says that the result is of type *unsigned long*; and *&nusers* as its second return parameter, which is a pointer to where the long result will be placed. Because RUSERSPROC_NUM takes no argument, the argument parameter of *rpc_call()* is *xdr_void()*.

If *rpc_call()* gets no answer within a certain time, it returns with an error code. In the example, the call is tried via all the transport services listed in */etc/netconfig* that are flagged as visible. The procedure on the remote machine corresponding to the above RUSERSPROC_NUM procedure might look like this:

```
char *
rusers()
{
static unsigned long nusers;
/*
* Code here to compute the number of users
* and place result in variable nusers.
*/
return((char *)&nusers);
}
```

It takes as an argument a pointer to the input parameters of the call of the RPC procedure (ignored in our example), and it returns a pointer to the result. For this reason, both the input argument and the return value are converted to *char* *.

The rpc_reg() routine

Normally, a server registers all the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. The code for this can be created with *rpgen* including a dispatch routine and support for port monitors. But programmers can also create servers themselves using *rpc_reg()*, and it is appropriate that they do so if they have simple applications, like the one shown as an example here. However, the server can contain only one RPC procedure.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
char *rusers();
main ()
{
if (rpc_reg(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
rusers, xdr_void, xdr_u_long, "visible") == -1) {
fprintf(stderr, "Couldn't Register\n");
exit(1);
}
svc_run();              /* Never returns */
fprintf(stderr, "Error: svc_run returned!\n");
exit(1);
}
```

The *rpc_reg()* routine registers a C procedure with the RPC service RUSERSPROC with the version RUSERSVERS. In addition, the (only) procedure number RUSERSPROC_NUM is assigned to the C function *rusers()*, which implements the remote procedure. The registration is done for each of the transport services of the specified type or, if the type parameter is NULL, for all the transport services named in NETPATH. *xdr_void* and *xdr_u_long* name the XDR filters for the RPC procedure's arguments and results, respectively. (Arguments or results that consist of several parts are passed as structures.) The last parameter specifies the desired *nettype*. Note that, when using *rpc_reg()*, you are not required to write your own dispatch routines.

The *svc_run()* routine is used at all levels of RPC programming. Strictly speaking, it does not belong to this or to any other level.

After registering the local procedure, the server program's main procedure calls *svc_run()*. In turn, this function calls the *rusers()* procedure for incoming RPC calls. The arguments and results of *rusers()* are encoded and decoded by means of the given XDR filters, without the need for you to explicitly call the filters.

Use of any data type

In the previous example, the RPC call returned a single *unsigned long*. RPC can handle any data structures. To do this, it uses the XDR (External Data Representation) format. Data are exchanged between RPC clients and servers in this format. The process of converting from a particular machine representation to XDR format is called encoding, and the reverse process is called decoding.

The type field parameters of *rpc_call()* and *rpc_reg()* can name an XDR primitive procedure, like *xdr_u_long()* in the previous example, or a procedure supplied by you (that may take a maximum of two parameters). XDR has these built-in primitive type routines:

xdr_int()	*xdr_u_int()*	*xdr_enum()*
xdr_long()	*xdr_u_long()*	*xdr_bool()*
xdr_short()	*xdr_u_short()*	*xdr_wrapstring()*
xdr_char()	*xdr_u_char()*	

The routine *xdr_string()* also exists, but takes more than two parameters. It cannot, therefore, be used with *rpc_call()* and *rpc_reg()*, which only pass two parameters to their XDR routines. *xdr_wrapstring()* has only two parameters, and is thus OK. It, in turn, calls *xdr_string()*.

As an example of a user-defined type routine, if you wanted to send the structure:

```
struct simple {
int a;
short b;
} simple;
```

then *rpc_call()* would be called as:

```
rpc_call(hostname, PROGNUM, VERSNUM, PROCNUM,
xdr_simple, &simple ...);
```

where *xdr_simple()* is written as:

```
#include <rpc/rpc.h>
#include "simple.h"
bool_t
xdr_simple(xdrsp, simplep)
XDR *xdrsp;
struct simple *simplep;
{
```

```
if (!xdr_int(xdrsp, &simplep->a))
return (FALSE);
if (!xdr_short(xdrsp, &simplep->b))
return (FALSE);
return (TRUE);
}
```

An XDR routine returns nonzero (true in the C sense) if it completes successfully, and zero otherwise. A complete description of XDR is provided in the Chapter "XDR/RPC protocol specification". Note that the above routine could have been generated automatically by using the *rpcgen* compiler.

In addition to the built-in primitives, there are some other prefabricated building blocks:

xdr_array()	*xdr_bytes()*	*xdr_reference()*
xdr_vector()	*xdr_union()*	*xdr_pointer()*
xdr_string()	*xdr_opaque()*	

To send a variable array of integers, the array might be packaged as a structure like this:

```
struct varintarr {
int *data;
int arrlnth;
} arr;
```

and sent by an RPC call such as:

```
rpc_call(hostname, PROGNUM, VERSNUM, PROCNUM,
xdr_varintarr, &arr...);
```

with *xdr_varintarr()* defined as:

```
bool_t
xdr_varintarr(xdrsp, arrp)
XDR *xdrsp;
struct varintarr *arrp;
{
return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
MAXLEN, sizeof(int), xdr_int));
}
```

The *xdr_array()* routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, one can use *xdr_vector()*, which serializes fixed-length arrays.

```
int intarr[SIZE];
bool_t
xdr_intarr(xdrsp, intarr)
XDR *xdrsp;
int intarr[];
{
return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
xdr_int));
}
```

XDR always converts quantities to 4-byte multiples when encoding. Thus, if either of the examples above involved characters (i. e. 8-bit sizes) instead of integers, each character would still occupy 32 bits. The byte-wise conversion of characters to XDR format is done using *xdr_bytes()*. *xdr_bytes()* has four parameters, which correspond to the first four parameters of *xdr_array()*.

For null-terminated strings, there is the *xdr_string()* routine, which is the same as *xdr_bytes()* without the length

parameter. On serializing it gets the string length from *strlen()*, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written *xdr_simple()* as well as the built-in functions *xdr_string()* and *xdr_reference()*, which chases pointers:

```
struct finalexample {
char *string;
struct simple *simplep;
} finalexample;
bool_t
xdr_finalexample(xdrsp, finalp)
XDR *xdrsp;
struct finalexample *finalp;
{
if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
return (FALSE);
if (!xdr_reference(xdrsp, &finalp->simplep,
sizeof(struct simple), xdr_simple))
return (FALSE);
return (TRUE);
}
```

Note that we could as easily call *xdr_simple()* here instead of *xdr_reference()*.

4.11 The lower levels of RPC

In the examples given for programming at the simplified interface, RPC takes care of almost as many details as would the *rpcgen* compiler. RPC does so by choosing defaults for almost everything, including the transport protocol.

This section shows how to control these details by using lower levels of the RPC library. The reader is assumed to be familiar with the Transport Level Interface (TLI).

There are several reasons for using lower levels of RPC:

- A program may need to directly control the selection of the transport protocol, which at the simplified interface level, can be done only by use of the NETPATH variable.
- A program may need to allocate and free memory while serializing or deserializing with XDR routines. There are no facilities for doing so available at the higher level.

clnt_call(), *clnt_destroy()*, *clnt_control()*, *clnt_perrno()*, *clnt_pcreateerror()*, *clnt_perror()* and *svc_destroy()* can be used at any of the lower levels of RPC. They are not available at the simplified-interface level only because they require the caller to have a transport handle.

The following sections illustrate programming at the lower levels of RPC.

The Section "The top level" describes RPC interfaces that allow for control of transport service selection by type.

The Section "The intermediate level" describes those interfaces that allow you to choose a specific transport.

The Section "The expert level" describes routines that:

- allow program control of client and server characteristics
- provide an interface to *rpcbind*

Finally, the Section "The bottom level" describes routines that control most details of transport options.

4.11.1 The top level

At the top level, the application can specify the type of transport that it wants to use, but not an individual transport. This level differs from the simplified interface to RPC in that the application is responsible for creating its own transport handles, on both the client and server sides.

The client side

Assume we have the following header file:

```
/*
* time_prot.h
*/
#include <rpc/types.h>
struct timev {
int second;
int minute;
int hour;
};
typedef struct timev timev;
bool_t xdr_timev(xdrsp, resp)
XDR *xdrsp;
struct timev *resp;
{
if (!xdr_int(xdrsp, &resp->second))
return (FALSE);
if (!xdr_int(xdrsp, &resp->minute))
return (FALSE);
if (!xdr_int(xdrsp, &resp->hour))
return (FALSE);
return (TRUE);
}
#define TIME_PROG ((u_long)76)
#define TIME_VERS ((u_long)1)
#define TIME_GET ((u_long)1)
```

The following code implements the client side of a trivial date service, written at the top level:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"
#define TOTAL (30)
/*
*  Caller of trivial date service
*  usage: calltime hostname
*/
main(argc,argv)
int argc;
char *argv[];
{
struct timeval timeout;
CLIENT *client;
enum clnt_stat stat;
struct timev timev;
char *nettype;
if (argc != 2 && argc != 3) {
fprintf(stderr,"usage: %s host [nettype]\n",argv[0]);
}
if (argc == 2)
nettype = "netpath";    /* Default */
else
nettype = argv[2];
client = clnt_create(argv[1], TIME_PROG, TIME_VERS,
nettype);
if (client == NULL) {
clnt_pcreateerror("Couldn't create client");
exit;
}
timeout.tv_sec = TOTAL;
timeout.tv_usec = 0;
stat = clnt_call(client, TIME_GET, xdr_void, NULL,
xdr_timev, &timev, timeout);
if (stat != RPC_SUCCESS) {
clnt_perror(client, "Call failed");
exit(1);
}
printf("%s: %02d:%02d:%02d GMT\n", nettype, timev.hour,
timev.minute, timev.second);
exit(0);
}
```

Note that if the program is started and the transport type *nettype* is not specified in the command line, the code assigns it as a pointer to the string *netpath*. Whenever the routines in the RPC library encounter this string, they consult the NETPATH environment variable for the user's list of acceptable network identifiers. If the client handle cannot be created, the reason for the failure can be printed using *clnt_pcreateerror()*, or the error status can be obtained via the global variable *rpc_createerr*. After the client handle is created, clnt_call() is used to make the remote call. It takes as arguments the number of the RPC procedure, an XDR filter for the argument, the argument pointer, an XDR filter for the result, the result pointer and the timeout period of the call. Normally, the latter should not be 0. In this particular example there are no arguments, and thus *xdr_void()* has been specified.

The server end

Here's the code for the time server:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"
static void time_prog();
main(argc,argv)
int argc;
char *argv[];
{
int transpnum;
char *nettype;
if (argc == 2)
nettype = argv[1];
else
nettype = "netpath";     /* Default */
transpnum = svc_create(time_prog, TIME_PROG, TIME_VERS,
nettype);
if (transpnum == 0) {
fprintf(stderr,"%s: cannot create %s service.\n",
argv[0], nettype);
exit(1);
}
svc_run();
}
```

```
/*
* The server dispatch function
*/
static void
time_prog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
struct timev rslt;
long thetime;
switch (rqstp->rq_proc)
{
case NULLPROC:
svc_sendreply(transp, xdr_void, NULL);
return;
case TIME_GET:
break;
default:
svcerr_noproc(transp);
return;
}
thetime = time(0);
rslt.second = thetime % 60;
thetime /= 60;
rslt.minute = thetime % 60;
thetime /= 60;
rslt.hour = thetime % 24;
if (! svc_sendreply(transp, xdr_timev, &rslt)) {
svcerr_systemerr(transp);
}
}
```

svc_create() returns the number of transport services created for server handles. *time_prog()* is the dispatch function called by *svc_run()* whenever there's a call for its associated program and version number.

Here the RPC procedure takes no arguments. Had arguments been required,

svc_getargs(transport, XDR_filter, argument_pointer)

could have been used to decode the arguments. In such cases, *svc_freeargs()* should be used to free up the arguments after the actual call has been made. The server reply results are sent back to the client using *svc_sendreply()*.

It is recommended that *rpcgen* be used to generate the dispatch function which can later be customized.

When *rpcgen* is used to generate the dispatch function, *svc_sendreply()* is called only after the actual procedure has returned, and hence it is essential to have *rslt* (in this example) declared as *static* within that actual procedure. In this example, *rslt* is not declared as static because *svc_sendreply()* is called from within the dispatch function.

4.11.2 The intermediate level

At the intermediate level, the application directly chooses the transport it wishes to use.

The client end

The following code implements the client side of the same time service shown above, but written to the intermediate level of the RPC package.

Here, the programmer requires the user to name, on the command line, the transport over which the call will be made:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>    /* For netconfig structure */
#include "time_prot.h"
#define TOTAL (30)
/*
*  Caller of trivial date service
*  usage: calltime hostname netid
*/
main(argc,argv)
int argc;
char *argv[];
{
struct netconfig *nconf;
/* Declarations from previous example */
if (argc != 3) {
fprintf(stderr,"usage: %s host netid\n",argv[0]);
}
nettype = argv[2];
if ((nconf = getnetconfigent(nettype)) == NULL) {
fprintf(stderr, "Bad netid type: %s\n", nettype);
exit;
}
```

```
client = clnt_tp_create(argv[1], TIME_PROG, TIME_VERS,
nconf);
if (client == NULL) {
clnt_pcreateerror("Could not create client");
exit(1);
}
/* Same as previous example after this point */
}
```

The *netconfig* structure can be obtained by a call to *getnetconfigent*(*nettype*). (More detailed information can be found in the "Networking Reference Manual" under *getnetconfig*). At this level, the program must explicitly make all decisions about network-selection.

The server end

Here's the corresponding server. The administrator who starts the service is required to name, on the command line, the transport over which the service is provided:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>        /* For netconfig structure */
#include "time_prot.h"
static void time_prog();
/* Service to supply Greenwich mean time */
/* usage: server netid */
main(argc,argv)
int argc;
char *argv[];
{
SVCXPRT *transp;
struct netconfig *nconf;
if (argc == 1) {
fprintf(stderr, "usage: server netid \n");
exit(1);
}
if ((nconf = getnetconfigent(argv[1])) == NULL) {
fprintf(stderr, "Could not find info on %s\n",
argv[1]);
exit(1);
}
```

```
transp = svc_tp_create(time_prog, TIME_PROG, TIME_VERS,
nconf);
if (transp == NULL) {
fprintf(stderr,"%s: cannot create %s service.\n",
argv[0], argv[1]);
exit(1)
}
svc_run();
}
static void
time_prog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
/* Code identical to Top Level version */
}
```

4.11.3 The expert level

At the expert level, transport service selection is done exactly as at the intermediate level. The only difference here is in the level of control that the application has over the details of the transport's configuration. Control at this level is much greater. These examples illustrate that control, which is exercised using the *clnt_tli_create()* and *svc_tli_create()* routines.

The client end

Here is the client side of some code that implements a version of *clntudp_create()* (the routine for creating a client handle for the UDP transport service) by means of *clnt_tli_create()*. The example shows the selection of a transport on the basis of the desired transport family. *clnt_tli_create()* is normally used to create a client handle when:

- the application wants to pass an open file descriptor, which may or may not be bound
- you want to feed the server's address to the client
- you want to specify the send and receive buffer size (here, 8800 bytes)

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
```

```
/*
* In an earlier implementation of RPC, the only transports
* supported were TCP/IP and UDP/IP. clntudp_create was based
* on the Berkeley sockets in this implementation. The present
* code, however, is based on the TLI.
*/
CLIENT *
clntudp_create(raddr, prog, vers, wait, sockp)
struct sockaddr_in *raddr;      /* Remote address */
u_long prog;                    /* Program number */
u_long vers;                    /* Version number */
struct timeval wait;            /* Time to wait */
int *sockp;                     /* fd pointer */
{
CLIENT *cl;                     /* Client handle */
int madefd = FALSE;             /* Is fd opened here */
int fd = *sockp;                /* fd */
struct t_bind *tbind;           /* bind address */
struct netconfig *nconf;        /* netconfig structure */
void *handlep;
if ((handlep = setnetconfig()) == 0) {  /* No transports
* available */
rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
return ((CLIENT *)NULL);
}
/*
* Search the transport table until a connectionless one is
* found that belongs to the INET family and has the name
* UDP
*/
while (nconf = getnetconfig(handlep)) {
if ((nconf->nc_semantics == NC_TPI_CLTS) &&
(strcmp(nconf->nc_protofmly, NC_INET) == 0) &&
(strcmp(nconf->nc_proto, NC_UDP) == 0))
break;
}
if (nconf == NULL) {
rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
goto err;
}
if (fd == RPC_ANYSOCK) {
fd = t_open(nconf->nc_device, O_RDWR, NULL);
if (fd == -1) {
rpc_createerr.cf_stat = RPC_SYSTEMERROR;
goto err;
}
madefd = TRUE;          /* The fd was opened here */
}
if (raddr->sin_port == 0) {   /* remote addr unknown */
u_short sport;
/*
* rpcb_getport() is a user provided routine
* which will call rpcb_getaddr and translate
* the netbuf address to port number.
```

```
*/
sport = rpcb_getport(raddr, prog, vers, nconf);
if (sport == 0) {
rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
goto err;
}
raddr->sin_port = sport;
}
/* Transform sockaddr_in to netbuf */
tbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR);
if (tbind == NULL) {
rpc_createerr.cf_stat = RPC_SYSTEMERROR;
goto err;
}
(void) memcpy(tbind->addr.buf, (char *)raddr,
(int)tbind->addr.maxlen);
tbind->addr.len = tbind->addr.maxlen;
/* Bind endpoint of a reserved address */
(void) bind_resv(fd);
cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog,
vers, 8800, 8800);
(void) endnetconfig(handlep);  /* Close the netconfig file */
(void) t_free((char *)tbind, T_BIND);
if (cl) {
*sockp = fd;
if (madefd == TRUE) {
/* fd should be closed while destroying the handle */
(void) CLNT_CONTROL(cl, CLSET_FD_CLOSE, NULL);
}
/* Set the retry time */
(void) CLNT_CONTROL(cl, CLSET_RETRY_TIMEOUT, &wait);
return (cl);
}
err:
if (madefd == TRUE)
(void) t_close(fd);
(void) endnetconfig(handlep);
return ((CLIENT *)NULL);
}
```

Transport service selection is done using the library functions *setnetconfig()*, *getnetconfig()*, and *endnetconfig()*. (Note that *endnetconfig()* is not called until after the call to *clnt_tli_create()*, near the end of the example.) *clntudp_create()* can be passed an open *fd,* but if not (*fd == RPC_ANYSOCK*), it will open its own using the *netconfig* structure for UDP.

If the remote address is not known, (*raddr->sin_port == 0*), then it is obtained from the remote *rpcbind*. Note the *bind_resv()* call, which serves to bind the client's endpoint to a reserved address. This call is necessary because there is no notion of a reserved address in RPC under TLI, as there is in both TCP and UDP. The implementation of this routine is of no interest here, because it is entirely transport specific. What is of interest is the scaffolding necessary to call it.

After the client handle has been created, you can customize it appropriately by calling *clnt_control()*. The RPC library is instructed to close the file descriptor as soon as the client handle is released by means of *clnt_destroy()*. In addition, the retry timeout period is set.

The server end

Below is the corresponding server code. It implements *svcudp_create()* in terms of *svc_tli_create()*, and calls the user provided *bind_resv()* to bind the transport endpoint to a reserved address. *svc_tli_create()* is normally used when the application needs a fine degree of control, and especially if it is necessary to:

- .pass an open file descriptor to the application
- pass the user's name
- set the send and receive buffer sizes (here being set to 8800 bytes)

The *fd* argument may be unbound when passed in. If it is, then it is bound to a given address, and the address is stored in a handle. If the name is set to NULL, and if the *fd* is initially unbound, it will be bound to any suitable address.

It is the responsibility of the programmer to use *rpcb_set()* to register the service with *rpcbind*.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
/*
* On the server side
*/
```

```
SVCXPRT *
svcudp_create(fd)
register int fd;
{
struct netconfig *nconf;
SVCXPRT *svc;
int madefd = FALSE;
int port;
void *handlep;
if ((handlep = setnetconfig()) == 0) { /* No transports
* available */
nc_perror("server");
return ((SVCXPRT *)NULL);
}
/*
* Try all the transport services till it gets one which is
* connectionless, where family is INET and name is UDP
*/
while (nconf = getnetconfig(handlep)) {
if ((nconf->nc_semantics == NC_TPI_CLTS) &&
(strcmp(nconf->nc_protofmly, NC_INET) == 0) &&
(strcmp(nconf->nc_proto, NC_UDP) == 0))
break;
}
if (nconf == NULL) {
endnetconfig(handlep);
fprintf(stderr, "UDP transport not available\n");
return ((SVCXPRT *)NULL);
}
if (fd == RPC_ANYFD) {
fd = t_open(nconf->nc_device, O_RDWR, NULL);
if (fd == -1) {
(void) endnetconfig();
(void) fprintf(stderr,
"svcudp_create: could not open connection\n");
return ((SVCXPRT *)NULL);
}
madefd = TRUE;
}
/*
* Bind Endpoint to a reserved address
*/
port = bind_resv(fd);
svc = svc_tli_create(fd, nconf, (struct t_bind *)NULL,
8800, 8800);
(void) endnetconfig(handlep);
```

```
if (svc == (SVCXPRT *)NULL) {
if (madefd)
(void) t_close(fd);
return ((SVCXPRT *)NULL);
}
if (port == -1)
/* Specifically set xp_port now */
svc->xp_port =
((struct sockaddr_in *)svc->xp_ltaddr.buf)->sin_port;
else
svc->xp_port = port;
return (svc);
}
```

The transport service selection here is done in a similar way as in *clntudp_create()*. *svcudp_create()* is set up to receive an open *fd*, but if it does not, it will open one itself using the selected *netconfig* structure. *bind_resv()* is a user-provided function that binds the *fd* to a reserved port if the caller is a superuser.

4.11.4 The bottom level

At the bottom-level interface to RPC, the application can control all options, transport-related and otherwise. *clnt_tli_create()*, and the other expert-level RPC interface routines are implemented on top of these bottom-level routines. The programmer should not normally be using these low-level routines. These routines are responsible for creating their own data structures, their own buffer management, the creation of their own RPC headers, etc. Callers of these routines like the expert level routine *clnt_tli_create()* are responsible for initializing the *cl_netid* and *cl_tp* fields within the client handle. The bottom level routines *clnt_dg_create()* and *clnt_vc_create()* are themselves responsible for populating the *clnt_ops* and *cl_private* fields. For a created handle, *cl_netid* is the network identifier (e.g. udp) of the transport and *cl_tp* is the device name of that transport (e.g. */dev/udp*).

4.12 Low-level data structures

For reference, here are the client- and server-side RPC handles, as well as an authentication structure.

```
/*
* Client rpc handle.
* Created by individual implementations
* Client is responsible for initializing auth
*/
typedef struct {
AUTH    *cl_auth;          /* authenticator */
struct clnt_ops {
enum clnt_stat  (*cl_call)(); /* call RPC procedure */
void    (*cl_abort)();    /* abort a call */
void    (*cl_geterr)();   /* get error code */
bool_t  (*cl_freeres)();  /* free results */
void    (*cl_destroy)();  /* destroy this structure */
bool_t  (*cl_control)();  /* control functions */
} *cl_ops;
caddr_t     cl_private;       /* private stuff */
char        *cl_netid;        /* transport identifiere */
char        *cl_tp;           /* device name */
} CLIENT;
```

The client-side handle contains an authentication structure. For a client program to authenticate itself, it must initialize the *cl_auth* field to an appropriate authentication structure:

```
/*
* Auth handle, interface to client side authenticators.
*/
typedef struct {
struct  opaque_auth     ah_cred;    /* credentials */
struct  opaque_auth     ah_verf;    /* verifier */
struct auth_ops {
void    (*ah_nextverf)();
int     (*ah_marshal)();    /* nextverf & encode */
int     (*ah_validate)();   /* validate verifier */
int     (*ah_refresh)();    /* refresh credentials */
void    (*ah_destroy)();    /* destroy this structure */
} *ah_ops;
caddr_t ah_private;             /* pointer to
* authentication data */
} AUTH;
```

Within the AUTH structure, *ah_cred* contains the caller's credentials, and *ah_verf* contains the information necessary to verify those credentials. (See under the Section "Authentication"").

Here is the server-side transport handle:

```
/*
* Server side transport handle
*/
typedef struct {
int       xp_fd;    /* associated file descriptor */
u_short   xp_port;  /* associated port number (obsolete) */
struct xp_ops {
bool_t  (*xp_recv)();      /* receive incoming requests */
enum xprt_stat (*xp_stat)(); /* get transport status */
```

```
bool_t   (*xp_getargs)();  /* get arguments */
bool_t   (*xp_reply)();    /* send reply */
bool_t   (*xp_freeargs)(); /* free mem allocated for args */
void    (*xp_destroy)();   /* destroy this struct */
} *xp_ops;
int            xp_addrlen; /* remote addr. length Obsolete */
char           *xp_tp;     /* transport device name */
char           *xp_netid;  /* transport identifier */
struct netbuf   xp_ltaddr;  /* local transport address */
struct netbuf   xp_rtaddr;  /* remote transport address */
char           xp_raddr[16]; /* remote address. Obsolete */
struct opaque_auth xp_verf; /* raw response verifier */
caddr_t        xp_p1;      /* private: for use by svc ops */
caddr_t        xp_p2;      /* private: for use by svc ops */
caddr_t        xp_p3;      /* private: for use by svc lib */
} SVCXPRT;
```

xp_fd is the file descriptor associated with the handle. Two or more server handles can share the same file descriptor.

xp_netid is the transport identifier (e.g. udp) of the transport on which this handle was created and *xp_tp* is the device name associated with that transport.

xp_ltaddr is the server's own name, while *xp_rtaddr* is the address of the remote caller and hence may change from call to call.

xp_netid, *xp_tp* and *xp_ltaddr* are initialized by *svc_tli_create()* and other expert-level routines.

The rest of the fields are initialized by the bottom-level server routines *svc_dg_create()* and *svc_vc_create()*.

4.13 Program testing with pseudo-RPC

There are two pseudo-RPC routines available for program testing. These routines, *clnt_raw_create()* and *svc_raw_create()*, exist to help the developer debug and test the non-network-specific functionality of an application before running it over a real network.

Here's an example of their use:

```
/*
* A simple program to increment a number by 1
*/
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>
struct timeval TIMEOUT = {0, 0};
static void server();
main (argc, argv)
int argc;
char **argv;
{
CLIENT *cl;
SVCXPRT *svc;
int num = 0, ans;
if (argc == 2)
num = atoi(argv[1]);
svc = svc_raw_create();
if (svc == NULL) {
fprintf(stderr, "Could not create server handle\n");
exit(1);
```

```
}
svc_reg(svc, 200000, 1, server, 0);
cl = clnt_raw_create(200000, 1);
if (cl == NULL) {
clnt_pcreateerror("raw");
exit(1);
}
if (clnt_call(cl, 1, xdr_int, &num, xdr_int, &ans,
TIMEOUT) != RPC_SUCCESS) {
clnt_perror(cl, "raw");
exit(1);
}
printf("Client: number returned %d\n", ans);
exit(0) ;
}
```

```
static void
server(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
int num;
switch(rqstp->rq_proc) {
case 0:
if (svc_sendreply(transp, xdr_void, 0) == NULL) {
fprintf(stderr, "error in null proc\n");
exit (1);
}
return;
case 1:
break;
default:
svcerr_noproc(transp);
return;
}
if (!svc_getargs(transp, xdr_int, &num)) {
svcerr_decode(transp);
return;
}
num++;
if (svc_sendreply(transp, xdr_int, &num) == NULL) {
fprintf(stderr, "error in sending answer\n");
exit (1);
}
return;
}
```

Note the following points:

- The server is not registered with *rpcbind*, and *svc_run()* is not called. The last parameter to *svc_reg()* is 0, which means that it will not register with *rpcbind*.
- All the RPC calls occur within the same thread of control.
- It is necessary that the server be created before the client.
- *svc_raw_create()* takes no parameters.
- The server dispatch routine is the same as it is for normal RPC servers.

4.14 Advanced RPC programming techniques

This section addresses areas of occasional interest to programmers using the lower level interfaces of the RPC package. The topics discussed are:

select() on the server side	If calling *svc_run()* is not feasible, a server can call the dispatcher directly.
RPC broadcast	Details of the broadcast mechanism are described.
Batching	Efficiency is gained if a series of calls can be batched.
Authentication	Two methods in common use are described.
Port monitors	Details are provided for interfacing with the *inetd* and *listener* port monitors.

4.14.1 select() at the server end

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling *svc_run()*.

If the other activity involves waiting on a file descriptor, however, the *svc_run()* call will not work.

Below is the code for *svc_run()*. Note that *svc_fdset* is a bit mask of all the file descriptors that the RPC uses for services. The mask can change every time a RPC library routine is called, because descriptors are constantly being opened and closed:

```
void
svc_run ()
{
fd_set readfds;
extern int errno;
for (;;) {
readfds = svc_fdset;
switch (select(_rpc_dtbsize(), &readfds,
(fd_set *)0, (fd_set *)0, (struct timeval *)0)) {
case -1:
if (errno == EINTR) {
continue;
}
```

```
/*
* log an error: svc_run: select failed
*/
return;
case 0:
continue;
default:
svc_getreqset(&readfds);
}
}
}
```

A process can bypass *svc_run()* and call the dispatch routine directly by means of *svc_getreqset()*. If the process knows the file descriptors of its transport endpoints, it can use its own *select()*, which then serves to monitor the RPC file descriptors as well as other descriptors.

4.14.2 RPC broadcast

rpcbind is a daemon that converts RPC program numbers into network addresses comprehensible to any transport provider. *rpcbind* supports broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

- Normally, RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be performed on connectionless protocols that support broadcasting, such as UDP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to *rpcbind*'s network address. Thus, only services that register themselves with *rpcbind* are accessible via the broadcast RPC mechanism.
- The size of broadcast requests is limited to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

The following illustrates how *rpc_broadcast()* is used and describes its arguments:

```
#include <rpc/clnt.h>
#include <rpc/rpcb_clnt.h>
. . .
enum clnt_stat                clnt_stat;
. . .
clnt_stat = rpc_broadcast(prognum, versnum, procnum,
inproc, in, outproc, out, eachresult, nettype)
u_long    prognum;       /* program number */
u_long    versnum;       /* version number */
u_long    procnum;       /* procedure number */
xdrproc_t inproc;        /* xdr routine for args */
caddr_t   in;            /* pointer to args */
xdrproc_t outproc;       /* xdr routine for results */
caddr_t   out;           /* pointer to results */
resultproc_t   eachresult;/* call with each result gotten */
char      *nettype;      /* transport type */
```

The procedure *eachresult()* is called each time a valid result is obtained. It returns a boolean that specifies whether the user wants more responses.

```
bool_t done;
. . .
done = eachresult(resultsp, raddr, nconf)
caddr_t resultsp;
struct netbuf *raddr;    /* Addr of responding machine */
struct netconfig *nconf; /* Transport which responded */
```

If *done* is TRUE, then broadcasting stops and *rpc_broadcast()* returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT.

4.14.3 Batching

The RPC architecture is designed so that clients send a call message, and wait for servers to reply that the call succeeded. This implies that clients do not perform any other tasks while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC batching has the following characteristics:

- Each RPC call requires no response from the server, and the server does not send a response message.
- A transport that guarantees the reliable and unequivocal transfer of the data is used (e.g. TCP).

Because the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Because the batched calls are buffered, the client should occasionally do a non-batched call to flush the pipeline.

An example of batching follows. Assume a string rendering service has two similar calls: one renders a string and returns *void* results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like this:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"
void windowdispatch();
main ()
{
int num;
num = svc_create(windowdispatch, WINDOWPROG,
WINDOWVERS, "tcp");
if (num == 0){
fprintf(stderr, "can't create an RPC server\n");
exit(1);
}
svc_run();  /* Never returns */
fprintf(stderr, "should never reach this point\n");
}
void
windowdispatch(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
```

```
{
char *s = NULL;
switch (rqstp->rq_proc) {
case NULLPROC:
if (!svc_sendreply(transp, xdr_void, 0))
fprintf(stderr, "can't reply to RPC call\n");
return;
case RENDERSTRING:
if (!svc_getargs(transp, xdr_wrapstring, &s)) {
fprintf(stderr, "can't decode arguments\n");
/*
* Tell caller an error occurred
*/
svcerr_decode(transp);
break;
}
/*
* Code here to render the string s
*/
if (!svc_sendreply(transp, xdr_void, NULL))
fprintf(stderr, "can't reply to RPC call\n");
break;
case RENDERSTRING_BATCHED:
if (!svc_getargs(transp, xdr_wrapstring, &s)) {
fprintf(stderr, "can't decode arguments\n");
/*
* We are silent in the face of protocol errors
*/
break;
}
/*
* Code here to render string s, but send no reply!
*/
break;
default:
svcerr_noproc(transp);
return;
}
/*
* Now free string allocated while decoding arguments
*/
svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course, the service could also have only one procedure that takes the string and a boolean value that specifying whether the procedure should respond. To take advantage of batching (using the code above), the client must make RPC calls on a TCP-based transport. The calls must have the following attributes:

- the XDR routine for the result must be zero (NULL)
- the RPC call's timeout must be zero

Here is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string (EOF):

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"
main(argc, argv)
int argc;
char **argv;
{
struct timeval total_timeout;
register CLIENT *client;
enum clnt_stat clnt_stat;
char buf[1000], *s = buf;
if ((client = clnt_create(argv[1], WINDOWPROG,
WINDOWVERS, "tcp")) == NULL) {
clnt_pcreateerror("clnt_create");
exit;
}
total_timeout.tv_sec = 0;
total_timeout.tv_usec = 0;
while (scanf("%s", s) != EOF) {
clnt_call(client, RENDERSTRING_BATCHED,
xdr_wrapstring, &s, xdr_void, NULL, total_timeout);
}
/* Now flush the pipeline */
total_timeout.tv_sec = 20;
clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
xdr_void, NULL, total_timeout);
if (clnt_stat != RPC_SUCCESS) {
clnt_perror(client, "rpc");
exit;
}
clnt_destroy(client);
exit(0);
}
```

Because the server sends no message, the client cannot be notified of any of the failures that may occur.

Batching performance

The following illustrates the benefits of batching. The above example was used to output all the lines in a 2000-line file. The service did nothing more than read them. The example was run in four configurations with the following results:

Configuration	Time
Machine to itself, regular RPC	50 seconds
Machine to itself, batched RPC	16 seconds
Machine to another, regular RPC	52 seconds

Machine to another, batched RPC	10 seconds

Running *fscanf()* on the same file only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution.

4.14.4 Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Some network services, such as a network filesystem, require stronger security than what has been presented so far.

Every RPC call is subjected to a style of authentication by the RPC package on the server. Similarly, the RPC client package generates and sends authentication parameters suitable for the style of authentication in effect. The default authentication type is AUTH_NONE (no authentication).

Just as different transport services can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients.

The authentication subsystem of the RPC package is open ended. That is, numerous styles of authentication are easy to support; programmers can design their own authentication style and easily configure the RPC package to support it.

In addition to AUTH_NONE, the RPC package already supports the following authentication styles:

AUTH_SYS: An authentication style based on traditional System V operating system process permissions authentication.

AUTH_SHORT: An alternate form of AUTH_SYS used by some servers for efficiency. Client programs using AUTH_SYS authentication should be prepared to receive AUTH_SHORT response verifiers from some servers. See "Authentication protocols" in the Chapter "XDR/RPC protocol specification" for details.

AUTH_NONE: the client end

When a caller creates a new RPC client handle as in:

clnt = clnt_create(host, prognum, versnum, nettype)

the appropriate transport instance defaults the associated authentication handle to be

clnt->cl_auth = authnone_createi;

If the programmer creates a new style of authentication, the programmer is responsible for destroying it with *auth_destroy*(*clnt->cl_auth*). This should always be done, to conserve memory.

AUTH_NONE: the server end

Service implementors have a harder time dealing with authentication issues because the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request passed to the dispatch routine:

```
/*
* An RPC request
*/
struct svc_req {
u_long        rq_prog;     /* RPC program number */
u_long        rq_vers;     /* RPC version number */
u_long   r     q_proc;     /* desired procedure number */
struct opaque_auth rq_cred;  /* raw credentials from wire */
caddr_t rq_clntcred;        /* credentials (read only) */
SVCXPRT        *rq_xprt;    /* associated transport */
};
```

The *rq_cred* is mostly opaque, except for one field of interest: the style or flavor of authentication credentials:

```
/*
* Authentication info.  Mostly opaque to the programmer.
*/
struct opaque_auth {
enum_t       oa_flavor;   /* style of credentials */
caddr_t      oa_base;     /* address of more auth stuff */
u_int        oa_length;   /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

- The request's *rq_cred* is well formed. Thus the programmer may check the request's *rq_cred.oa_flavor* and determine the authentication style the caller used. The programmer may also wish to inspect the other fields of *rq_cred* if the style is not one supported by the RPC package.
- The request's *rq_clntcred* field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only AUTH_NONE, AUTH_SYS and AUTH_SHORT styles are currently supported, so (currently) *rq_clntcred* could be cast only as a pointer to an *authsys_parms* or *short_hand_verf* structure. If *rq_clntcred* is NULL, the service implementor may wish to inspect the other (opaque) fields of *rq_cred* if the service knows about a new type of authentication that the RPC package does not know about.

AUTH_SYS authentication

The RPC client can choose to use AUTH_SYS style authentication by setting clnt–>cl_auth after creating the RPC client handle:

```
clnt->cl_auth = authsys_create_default();
```

This causes each RPC call associated with *clnt* to carry with it the following authentication credentials structure:

```
/*
* AUTH_SYS style credentials.
*/
struct authsys_parms {
u_long     aup_time;    /* credentials creation time */
char       *aup_machname;   /* host name where client is */
uid_t      aup_uid;         /* client's effective uid */
gid_t      aup_gid;         /* client's current group id */
u_int      aup_len;         /* element length of aup_gids */
gid_t      *aup_gids;       /* array of groups user is in */
};
```

These fields are set by *authsys_create_default()* by invoking the appropriate system calls.

The following shows the server for an RPC procedure, RUSERPROC_*n*, that computes the number of users on the network. As a trivial demonstration of authentication, this server checks AUTH_SYS credentials and does not respond to callers whose *uid* is 16:

```
nuser(rqstp, transp)
struct svc_req *rqstp;
VCXPRT *transp;
{
struct authsys_parms *SYS_cred;
uid_t uid;
unsigned long nusers;
/*
* we don't care about authentication for null proc
*/
```

```
if (rqstp->rq_proc == NULLPROC) {
if (!svc_sendreply(transp, xdr_void, 0)) {
fprintf(stderr, "can't reply to RPC call\n");
return (1);
}
return;
}
/*
* now get the uid
*/
switch (rqstp->rq_cred.oa_flavor) {
case AUTH_SYS:
SYS_cred =
(struct authsys_parms *)rqstp->rq_clntcred;
uid = SYS_cred->aup_uid;
break;
case AUTH_NONE:
default:
svcerr_weakauth(transp);
return;
}
switch (rqstp->rq_proc) {
case RUSERSPROC_n:
/*
* make sure caller is allowed to call this proc
*/
if (uid == 16) {
svcerr_systemerr(transp);
return;
}
/*
* Code here to compute the number of users
* and assign it to the variable nusers
*/
if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
fprintf(stderr, "can't reply to RPC call\n");
□□□□□□□□return (1);
}
return;
default:
svcerr_noproc(transp);
return;
}
}
```

A few things should be noted here:

- It is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero).
- The server should call *svcerr_weakauth()* if the authentication parameter's type is not suitable for the service.
- The service protocol itself should return status for access denied; in this example, the protocol does not have such a status, so the service primitive *svcerr_systemerr()* is called instead.

RPC deals only with authentication and not with individual services' access control. The services themselves

must establish access control policies and reflect these policies as return statuses in their protocols.

4.14.5 Using port monitors

An RPC server can be started from port monitors such as *inetd* and *listener*. These port monitors listen for requests for the services, and spawn servers in response to those requests. The file descriptor of the transport endpoint is passed to the started server process as the *fd* descriptor. In the case of *inetd*, after the server has serviced the request, it may exit immediately or wait a given interval of time for another request to come in.

In the case of *listener*, servers should exit immediately because *listener* will always spawn a new process rather than give a request to a waiting server process.

The following routine can be used to create a service:

transp = svc_tli_create(0, nconf, NULL, 0, 0)

nconf is the *netconfig* structure of the transport on which the request came in.

Because the port monitors have already registered the service with *rpcbind*, there is no need for the service to register itself. Nevertheless, it must call *svc_reg()*:

svc_reg(transp, PROGNUM, VERSNUM, dispatch, NULL)

The *netconfig* structure here is NULL.

Programmers should study *rpcgen*-generated server stubs to better see the sequence in which these routines are called.

For connection-oriented transport services, the following routine provides a lower level interface:

transp = svc_fd_create(0, recvsize, sendsize);

The file descriptor passed here is 0. The user may set the value of *recvsize* or *sendsize* to any appropriate buffer size. If they use a 0 in either case, a system default size will be chosen. This routine should be used by application servers that do not do any listening of their own, i.e., servers that simply do their job and return.

Using inetd

The format of entries in */etc/inetd.conf* for RPC services is as follows:

rpc_prog/vers socket_type rpc/proto flags uid pathname args

where *rpc_prog* is the symbolic name of the program as it appears in */etc/rpc*, *vers* is the version number, *socket_type* is one of *dgram* or *stream* with the options /BSD or /STREAMSfor connectionless or virtual circuit transport, respectively, *proto* is transport protocol, such as tcp or udp and must make sense with respect to the specified *socket_type*; *flags* is one of *wait* or *nowait*, *uid* must exist in */etc/passwd*, *pathname* is the full pathname of the server daemon and *args* are arguments to be passed to the daemon when it is invoked. For example:

mountd dgram rpc/udp wait root /usr/sbin/rpc.mountd rpc.mount

For more information, see the description of *inetd.conf* in the "Networking Reference Manual".

Using listener

We will assume here that the reader already knows the details of setting up the *listener* process and of using *pmadm*. The following shows how to use *pmadm* to add RPC services:

pmadm -a -p *pm_tag* -s *svctag* -i *id* -v *ver* \
-m 'nlsadmin -c *command* -D -R *prog:vers*'

Here *-a* means to add a service, *-p pm_tag* specifies a *listener* tag, *-s svctag* is the name of the server for *listener*, *-i id* is the */etc/passwd* user ID assigned to service *svctag, -v ver* is the version number of the *listener* file, and *-m* specifies the *nlsadmin* command for invoking the service. *nlsadmin* may have additional arguments. For example, to add version 1 of an RPC program server named *rusersd* the *pmadm* command might be:

pmadm -a -p tcp -s rusers -i root -v 4 \
>□□□□□-m 'nlsadmin -c /usr/sbin/rpc.ruserd -D -R 100002:1'

Here, the command is given root permissions, installed in version 4 of the *listener* database file, and is made available over TCP transport services.

Because of the complexity of the arguments and options that can follow the *pmadm –a* invocation, it may be convenient to use a command script or the menu system to add RPC services. If you use the menu system, enter *sysadm ports*, then choose the *port_services* option. After adding a service, the *listener* must be reinitialized before the service will be available. This is accomplished by stopping, then restarting the *listener*, as follows (note that *rpcbind* must be running):

sacadm -k -p *pmtag* # stop the listener
sacadm -s -p *pmtag* # start the listener

For more information, see *listen*, *pmadm*, *sacadm* and *sysadm* manual pages in the "Networking Reference Manual" and the "System Administrator's Guide".

4.15 Advanced examples

This section contains examples. "Versions" shows how to register multiple versions of a remote procedure. The Section "Connection-oriented transport services" shows a remote copy program. The Section "Callback procedures" shows how a server can be made to place a "client call" back to a client that calls it. The Section "Memory allocation with XDR" illustrates how this is done.

4.15.1 Versions

By convention, the first version number of program PROG is PROGVERS_ORIG and the most recent version is PROGVERS.

Suppose there is a new version of the *ruser* program that returns an *unsigned short* rather than a *long*. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions would do a double register. The same server handle would be used for both of these registrations.

```
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_ORIG,
nuser, nconf)) {
fprintf(stderr, "can't register RUSER service\n");
exit(1);
}
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_SHORT,
nuser, nconf)) {
fprintf(stderr, "can't register RUSER service\n");
exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
unsigned long nusers;
unsigned short nusers2;
switch (rqstp->rq_proc) {
case NULLPROC:
if (!svc_sendreply(transp, xdr_void, 0)) {
fprintf(stderr, "can't reply to RPC call\n");
return (1);
}
return (0);
case RUSERSPROC_NUM:
/*
* Code here to compute the number of users
* and assign it to the variable nusers
*/
nusers2 = nusers;
switch (rqstp->rq_vers) {
case RUSERSVERS_ORIG:
if (!svc_sendreply(transp, xdr_u_long,
&nusers)) {
fprintf(stderr,"can't reply to RPC call\n");
}
break;
case RUSERSVERS_SHORT:
if (!svc_sendreply(transp, xdr_u_short,
```

```
&nusers2)) {
fprintf(stderr,"can't reply to RPC call\n");
}
break;
}
default:
svcerr_noproc(transp);
return;
}
return (0);
}
```

4.15.2 Connection-oriented transport services

Here is an example that copies a file from one system to another. The initiator of the RPC *send* call takes its standard input and sends it to the server *receive*, which prints it on standard output. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
* The xdr routine:   on decode, read from wire, write onto fp
*                    on encode, read from fp, write onto wire
*/
#include <stdio.h>
#include <rpc/rpc.h>
xdr_rcp(xdrs, fp)
XDR *xdrs;
FILE *fp;
```

```
{
unsigned long size;
char buf[BUFSIZ], *p;
if (xdrs->x_op == XDR_FREE)/* nothing to free */
return 1;
while (1) {
if (xdrs->x_op == XDR_ENCODE) {
if ((size = fread(buf, sizeof(char), BUFSIZ,
fp)) == 0 && ferror(fp)) {
fprintf(stderr, "can't fread\n");
return (1);
}
}
p = buf;
if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
return 0;
if (size == 0)
return 1;
if (xdrs->x_op == XDR_DECODE) {
if (fwrite(buf, sizeof(char), size,
fp) != size) {
fprintf(stderr, "can't fwrite\n");
return (1);
}
}
}
}
```

Note that in the following two screens, the serializing and deserializing is done only by *xdr_bytes()*.

```
/* The sender routines */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
main(argc, argv)
int argc;
char **argv;
{
int xdr_rcp();
if (argc < 2) {
fprintf(stderr, "usage: %s servername\n", argv[0]);
exit(1);
}
```

```
if (callcots(argv[1], RCPPROG, RCPPROC, RCPVERS,
xdr_rcp, stdin, xdr_void, 0) != 0) {
exit(1);
}
exit(0);
}
callcots(host, prognum, procnum, versnum, inproc, in, outproc, out)
char *host, *in, *out;
xdrproc_t inproc, outproc;
{
enum clnt_stat clnt_stat;
register CLIENT *client;
struct timeval total_timeout;
if ((client = clnt_create(host, prognum, versnum,
"circuit_v")) == NULL) {
perror("clnt_create");
return (-1);
}
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
clnt_stat = clnt_call(client, procnum,
inproc, in, outproc, out, total_timeout);
clnt_destroy(client);
if (clnt_stat != RPC_SUCCESS) {
clnt_perror("callcots");
}
return ((int)clnt_stat);
}
/*
* The receiving routines
*/
#include <stdio.h>
#include <rpc/rpc.h>
main ()
{
int rcp_service(), xdr_rcp();
if (svc_create(rpc_service, RCPPROG, RCPVERS,
"circuit_v") == 0) {
fprintf("svc_create: error\n");
exit(1);
}
svc_run();  /* never returns */
fprintf(stderr, "svc_run should never return\n");
}
```

```
rcp_service(rqstp, transp)
register struct svc_req *rqstp;
register SVCXPRT *transp;
{
switch (rqstp->rq_proc) {
case NULLPROC:
if (svc_sendreply(transp, xdr_void, 0) == 0) {
fprintf(stderr, "err: rcp_service");
return (1);
}
return;
case RCPPROC:
if (!svc_getargs(transp, xdr_rcp, stdout)) {
svcerr_decode(transp);
return (1);
}
if (!svc_sendreply(transp, xdr_void, 0)) {
fprintf(stderr, "can't reply\n");
return (1);
}
return (0);
default:
svcerr_noproc(transp);
return (1);
}
}
```

Note that on the server side no explicit action was taken after receiving the arguments. This is because *xdr_rcp()* did all the necessary dirty work automatically.

4.15.3 Callback procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back to the client process. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an rpc call to the window program, so that it can inform the user that a breakpoint has been reached.

To do an RPC callback, a program number is needed to make the RPC call. Because this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. In the following example, the routine *gettransient()* returns a valid program number in the transient range, and registers it with *rpcbind*. The call to *rpcb_set()* is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
gettransient(vers, nconf, address)
int vers;
struct netconfig *nconf;
struct netbuf *address;
{
static int prog = 0x40000000;
while (! rpcb_set(prog++, vers, nconf, address))
continue;
```

```
return (prog - 1);
}
```

The following program illustrates how to use the *gettransient()* routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an ALARM signal in this example), it sends a callback RPC call, using the program number it received earlier.

```
/*
* client
*/
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include "example.h"
main(argc, argv)
int argc;
char **argv;
{
```

```
SVCXPRT *xprt;
struct netconfig *nconf;
int prognum;
enum clnt_stat stat;
if (argc != 3) {
fprintf(stderr, "usage: clnt host transport\n");
exit (1);
}
nconf = getnetconfigent(argv[2]);
if (nconf == NULL) {
fprintf(stderr, "unknown transport\n");
exit (1);
}
xprt = svc_tli_create(RPC_ANYFD, nconf,
(struct t_bind *)NULL, 0, 0);
if (xprt == (SVCXPRT *)NULL) {
fprintf(stderr, "could not create server handle\n");
exit (1);
}
prognum = gettransient(1, nconf, &xprt->xp_ltaddr);
fprintf(stderr, "client gets prognum %d\n", prognum);
if (svc_reg(xprt, prognum, 1, callback, NULL)
== FALSE) {
fprintf(stderr, "could not register service\n");
exit(1);
}
stat = rpc_call(argv[1], EXAMPLEPROG, EXAMPLEVERS,
EXAMPLEPROC_CALLBACK, xdr_int, &prognum, xdr_void,
NULL, NULL);
if (stat != RPC_SUCCESS) {
clnt_perrno(stat);
exit(1);
}
svc_run();
exit(1);
}
callback(rqstp, transp)
register struct svc_req *rqstp;
register SVCXPRT *transp;
{
```

```
switch (rqstp->rq_proc) {
case 0:
if (!svc_sendreply(transp, xdr_void, 0)) {
fprintf(stderr, "err: exampleprog\n");
return (1);
}
return (0);
case 1:
if (!svc_getargs(transp, xdr_void, 0)) {
svcerr_decode(transp);
return (1);
}
fprintf(stderr, "client got callback\n");
if (!svc_sendreply(transp, xdr_void, 0)) {
fprintf(stderr, "err: exampleprog");
return (1);
}
return (0);
}
return (2);
}
```

This example shows how the parameters required for calling *gettransient()* are first created by calling *getnetconfigent()* and *svc_tli_create()*. After creating the handle, *svc_reg()* is called (with the last parameter given as NULL) to register the dispatch function with the dispatcher. Once the server side is ready, it then notifies the actual server of its dynamic program number with RPC EXAMPLEPROC_CALLBACK. On success it then waits for requests from the remote server.

In the following example, the server makes an RPC call to the client on an ALARM signal, but only if the client has passed the program number to the server. This server example illustrates the simplicity of the code when one is using *rpc_reg()*.

```
/*
* server
*/
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
#include "example.h"
char *getnewprog();
char *hostname;
int docallback();
int pnum;      /* program number for callback routine */
main ()
```

```
{
if (argc != 2) {
fprint(stderr, "usage: server hostname\n");
exit (1);
}
hostname = argv[1];
rpc_reg(EXAMPLEPROG, EXAMPLEVERS, EXAMPLEPROC_CALLBACK,
getnewprog, xdr_int, xdr_void, NULL);
signal(SIGALRM, docallback);
alarm(10);
svc_run();
fprintf(stderr, "Error: svc_run shouldn't return\n");
}
char *
getnewprog(pnump)
char *pnump;
{
pnum = *(int *)pnump;
return NULL;
}
docallback ()
{
int ans;
if (pnum == 0) {
alarm (10);
return;
}
ans = rpc_call(hostname, pnum, 1, 1, xdr_void, 0,
xdr_void, 0, NULL);
if (ans != RPC_SUCCESS) {
fprintf(stderr, "server: ");
clnt_perrno(ans);
}
}
```

4.15.4 Memory allocation with XDR

XDR routines not only do encoding and decoding, they also do memory allocation. The second parameter of *xdr_array()* is a pointer to an array, rather than the array itself.

If it is NULL, then *xdr_array()* allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine *xdr_chararr1()*, which deals with a fixed array of bytes with length SIZE:

```
xdr_chararr1(xdrsp, chararr)
XDR *xdrsp;
char chararr[];
{
char *p;
int len;
p = chararr;
len = SIZE;
return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If memory has already been allocated in *chararr*, the routine can be called from the server as follows:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

To have XDR to do the allocation, the routine must be rewritten in the following way:

```
xdr_chararr2(xdrsp, chararrp)
XDR *xdrsp;
char **chararrp;
{
int len;
len = SIZE;
return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
* Use the result here
*/
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

Note that, after being used, the character array should normally be freed with *svc_freeargs()*. *svc_freeargs()* will not attempt to free any memory if the variable indicating it is NULL. For example, in the routine *xdr_finalexample()*, given earlier, if *finalp->string* were NULL, then it would not be freed. The same is true for *finalp->simplep*.

To summarize:

- Each XDR routine is responsible for serializing, deserializing, and freeing memory.
- When an XDR routine is called from *rpc_call()*, the serializing part is used.
- When called from *svc_getargs()*, the deserializer is used.
- When called from *svc_freeargs()*, the memory deallocator is used.

When building simple programs like those given as examples in this section, you do not have to worry about the three modes.

5 Network services

5.1 Transport service selection

In order for network applications to be portable to different environments, the application process must have a standard interface for the various transport services. Transport service selection provides a simple and consistent interface that allows user applications (e.g. TLI programs) to select transport services, thus enabling applications to remain independent of transport services and the underlying networks. System V Networking Services applications that allow the user to influence the choice of networks will use the standard interface outlined here.

Transport service selection routines may be used by the client portion of an application when the application initiates communication with a peer application on another machine; they may also be used by the server portion of an application. Servers can accept requests via multiple transport services.

Choosing among the available transport services is the responsibility of the application. The transport service selection mechanism is intended to make that selection uniform and simple.

5.1.1 How transport service selection works

The transport service selection component is built around

- a transport service configuration file (the */etc/netconfig* file) that contains an entry for each transport service on the system, and
- an optional NETPATH environment variable that can be set by the user and contains an ordered list of transport service identifiers. These identifiers match the *netconfig* field *transport ID* (*network ID*) and are used as links to the records in the *netconfig* file. The *netconfig* file is described in the next section.

The programming interface for transport service selection consists of a set of routines that organize access to the configuration file. One set of library routines handles only the *netconfig* entries identified by the NETPATH environment variable. These routines are described below in the section entitled Routines that access netconfig via NETPATH and on the *getnetpath* manual page in the "Networking Reference Manual".

Applications should use the routines that access *netconfig* via NETPATH because they allow users to control transport service selection. Only if this is not desired should routines that access the *netconfig* file directly be used.

These routines are described below in the section entitled "Routines that access *netconfig* directly" (...) and on the *getnetconfig* manual page of the "Networking Reference Manual".

5.1.2 The netconfig file

The *netconfig* file describes all the transport services on a given machine. All of the entries in the *netconfig* file contain the following fields:

transport ID	*semantics*	*flag*	*protocol family*
protocol name	*device file*	*dynamic libraries*	

transport ID (network-id)	A meaningful representation of a transport service name (for example, tcp or udp). Applications that require the name of a transport service obtain the name from this field.
semantics	The semantics of a specific transport service. Valid semantics are: *tpi_clts* connectionless *tpi_cots* connection-oriented *tpi_cots_ord* connection-oriented with orderly release Applications that require certain semantics, such as a virtual connection, can use this field to determine if the transport service has the required semantics.

flag	Flags of the transport service. The currently defined flags are v (visible), b (broadcast) and - (empty flag). The v flag is described in the Section "The NETPATH environment variable"". The b flag indicates that broadcasts can be sent via the transport service.
protocol family	The protocol family name of the transport service, for example, inet, osinet or loopback. See also the *netconfig* manual page in the "Networking Reference Manual".
protocol name	The protocol name of the transport service. If the protocol family is inet, then *protocol name* is tcp, udp, or icmp. Otherwise, the value of *protocol name* is a hyphen (–). See the *netconfig* manual page in the "Networking Reference Manual".
transport service device file	The full pathname of the special file to open when accessing the transport service.
dynamic libraries	Names of the dynamic libraries. This field contains the comma-separated full pathnames of the libraries that contain name-to-address mapping routines. No spaces may be entered after the commas.

The fields correspond to elements of the *struct netconfig* structure. Pointers returned by library routines for transport service selection are pointers to *netconfig* entries in *struct netconfig* format. The *netconfig* structure is shown in the next example.

The netconfig structure

```
struct   netconfig {
char   *nc_netid;   /* transport service identifier */
unsigned long   nc_semantics;   /* semantics of the
* transport service
*/
unsigned long   nc_flag;       /* flags for the protocol */
char   *nc_protofmly;          /* protocol family */
char   *nc_proto;              /* protocol-dep. if family
* inet */
char   *nc_device;             /* special file name*/
unsigned long   nc_nlookups;   /* no. of entries
* in nc_lookups
*/
char   **nc_lookups;           /* names of the
* dynamic libraries
*/
unsigned long   nc_unused[8];
};
```

Valid transport service IDs are defined by the system administrator, who is responsible for ensuring that transport service IDs are locally unique. If they are not, some transport service selection routines cannot operate in a well-defined manner. For example, it is not possible to know which transport service *getnetconfigent*("*udp*") will use if there are two *netconfig* entries with the transport service ID "udp".

The system administrator also determines the order of the entries in the *netconfig* file. Some routines that retrieve entries from */etc/netconfig* return entries in order, beginning at the top of the file. The order of transport services in the *netconfig* file therefore becomes the default transport service search path for applications using these routines.

The *netconfig* file and the *struct netconfig* structure are described in greater detail on the *netconfig* manual page in the "Networking Reference Manual".

5.1.3 The NETPATH environment variable

In most cases the user isn't interested in which transport service is used for a specific operation. Typically an application uses the default transport service search path established by the system administrator to locate an available network. However, when a user wants to influence the choices made by an application, the application can modify the NETPATH shell variable and use routines that carry out transport service selection using the NETPATH variable.

NETPATH is similar to the PATH variable. It consists of a colon-separated list of transport service IDs. Each transport service ID in the NETPATH variable corresponds to the *transport ID* field of a record in the *netconfig* file, for example

NETPATH=udp:tcp

The set of default transport services differs for the routines that access *netconfig* via the NETPATH environment variable (described in the next section) and the routines that access *netconfig* directly (described later in this chapter). The available transport services for routines that access *netconfig* via NETPATH consists of the visible transport services in the *netconfig* file. For routines that access *netconfig* directly, all the transport services in the *netconfig* file are available. A transport service is visible if the system administrator has included a v flag in the flag field of its *netconfig* entry.

5.1.4 Routines that access netconfig via NETPATH

The three routines described in this section access the transport service configuration file indirectly through the NETPATH environment variable. The user is thus able to specify one or more transport services the application is to use and, if more than one transport service is specified, in what order they are to be tried. These routines have the following syntax:

```
#include <netconfig.h>
void * setnetpath()
struct netconfig *
getnetpath(handlep)
void * handlep;
int
endnetpath(handlep)
void * handlep;
```

setnetpath() returns a pointer to a list of the *netconfig* entries specified in the NETPATH variable. The pointer (also known as a handle) is used when traversing this database with *getnetpath()*. Each call to *setnetpath()* returns a different pointer.

setnetpath() must be called before the first call to *getnetpath()* and can also be called at any other time. *setnetpath()* returns NULL if the *netconfig* file is not present.

When first called, *getnetpath()* returns a pointer to the *netconfig* entry that corresponds to the first component of the NETPATH variable. NETPATH components are read from left to right. On each subsequent call, *getnetpath()* returns a pointer to the *netconfig* entry that corresponds to the next component of the NETPATH variable. *getnetpath()* returns NULL when the end of the list returned by NETPATH is reached. A call to *getnetpath()* without an initial call to *setnetpath()* will produce an error. *getnetpath()* takes a handle as an argument.

getnetpath() silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the *netconfig* file. If the NETPATH variable is unset, *getnetpath()* behaves as if NETPATH were set to the sequence of default transport services in the *netconfig* file, in the order in which they are listed.

endnetpath() is called to free the list returned by *setnetpath()* when processing is complete. It returns 0 on success and −1 on failure. *endnetpath()* will fail if *setnetpath()* was not called previously. *endnetpath()* accepts *handlep* as an argument.

```
void *handlep;
if ((handlep = setnetpath()) == NULL) {
nc_perror("setnetpath failed");
exit(1);
}
while ((netconfigp = getnetpath(handlep)) != NULL) {
/*
*   netconfigp now describes a transport service
*/
printf ("name of the transport service: %s \n",
netconfigp --> nc_netid);
}
endnetpath(handlep);
```

5.1.5 Routines that access netconfig directly

Five functions access the transport service configuration file, */etc/netconfig*. They have the following syntax:

```
#include <netconfig.h>
void *
setnetconfig()
struct netconfig *
```

```
getnetconfig(handlep)
void * handlep;
int
endnetconfig(handlep)
void * handlep;
struct netconfig *
getnetconfigent(netid)
char * netid;
int
freenetconfigent(netconfigp)
struct netconfig * netconfigp;
```

A call to *setnetconfig()* returns a pointer to be used when traversing the database with *getnetconfig()*.This database pointer is also called a handle. Each call to *setnetconfig()* returns a different database pointer. Each call to *getnetconfig()* returns the next entry in *netconfig* associated with the handle returned by *setnetconfig()*.

When first called, *getnetconfig()* returns a pointer to the first entry in the *netconfig* file. On each subsequent call, *getnetconfig()* returns a pointer to the next entry in the file; it can thus be used to search the entire *netconfig* file. *getnetconfig()* returns NULL to the calling routine at end of file.

endnetconfig() may be called to free the handle when processing is complete. *endnetconfig()* may not be called before *setnetconfig()*. *endnetconfig()* returns 0 on success and −1 on failure (for example, if *setnetconfig()* was not called previously).

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

getnetconfigent(netid) returns a pointer to the *struct netconfig* structure corresponding to *netid*. It returns NULL if *netid* is invalid (that is, does not name an entry in the *netconfig* database). *freenetconfigent()* frees the structure returned by *getnetconfigent()*.

setnetconfig() must not be called before *getnetconfigent()*.

```
/*
*  assume udp is a transport service ID on this machine
*/
```

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

5.2 Name-to-address mapping

The name-to-address mapping feature allows an application to obtain the address of a machine or service regardless of the transport service used. The name-to-address mapping feature consists of the following routines:

- *netdir_getbyname*
- *netdir_getbyaddr*
- *netdir_free*
- *netdir_mergeaddr*
- *taddr2uaddr*
- *uaddr2taddr*
- *netdir_options*

Each routine takes a pointer to a *struct netconfig*, which describes a transport service. Each library entered in */etc/netconfig* is tried. UNIX (Reliant UNIX 5.45) contains libraries for the Internet and loopback protocols and a library for accessing DNS.

The libraries are:

tcpip.so	Contains the name-to-address mapping routines for the Internet protocol suite.
resolv.so	Contains the DNS (Domain Name Server) mapping routines for the Internet protocol suite.
straddr.so	Contains the name-to-address mapping routines for the loopback protocols.

The routines are described below in the section entitled Using the name-to-address mapping routines, and on the *netdir* manual page in the "Networking Reference Manual". The *struct netconfig* structure is described on the *netconfig* manual page in the "Networking Reference Manual".

5.2.1 The name-to-address mapping libraries

Files for each of the libraries must be created and maintained by the system administrator.

The routines in the *tcpip.so* dynamic library create addresses from the */etc/hosts* and */etc/services* files. The */etc/hosts* file contains at least two fields, the machine's IP address and the machine name. For example:

```
192.11.108.01     bilbo
192.11.108.16     elvis
```

The */etc/services* file contains two fields, a service name and a port number with one of two protocol specifications, either tcp or udp. For example:

```
rpcbind     111/udp
rpcbind     111/tcp
login      513/tcp
listen     1025/tcp
```

When an application uses this library to request the address of a service on a particular host, the host name must be entered in the */etc/hosts* file and the service name must be entered in the */etc/services* file. If one or the

other is not, an error will be returned by the name-to-address mapping routines.

The routines in the *straddr.so* dynamic library use addresses from files that use the same format as those of *tcpip.so*. The files used by *straddr.so* are */etc/net/transport/hosts* and */etc/net/transport/services*. *transport* is the local name of the transport service (specified in the *transport ID* field (*network ID*) of the */etc/netconfig* file). For example, the *hosts* file for ticlts would be */etc/net/ticlts/hosts*, and the service file for ticlts would be */etc/net/ticlts/services*. straddr.so expects strings as addresses. The */etc/net/transport/hosts* file contains a string that is considered to be the machine address followed by the machine name. For example:

```
bilboaddr       bilbo
elvisaddr       elvis
frodoaddr       frodo
```

The */etc/net/transport/services* file contains service names followed by strings identifying the ports of the services. For example:

```
rpcbind     rpc
listen      serve
```

The routines create the full string address by combining the host address and the service port, separating the two with a dot (.). For example, the address of the listen service on bilbo would be *bilboaddr.serve*, and the address of the *rpcbind* service on bilbo would be *bilboaddr.rpc*.

When an application requests the address of a service on the local host and uses *straddr.so*, the host name must appear in */etc/net/transport/hosts* and the service name must appear in */etc/net/transport/services*. If one or the other does not, the name-to-address mapping routines will return an error.

5.2.2 Using the name-to-address mapping routines

```
int
netdir_getbyname(config, service, nd_addrs)
struct netconfig       *config;
struct nd_hostserv     *service;
struct nd_addrlist     **addrs;
int
netdir_getbyaddr(config, service, netaddr)
struct netconfig        *config;
struct nd_hostservlist  **service;
struct netbuf           *netaddr;
void
netdir_free(ptr, type)
char    *ptr;
int    ident;;
int
netdir_mergeaddr(config, mrg_uaddr, s_uaddr, c_uaddr)
struct netconfig *config;
char **mrg_uaddr, *s_uaddr, *c_uaddr;
char *
taddr2uaddr(config, addr)
struct netconfig       *config;
struct netbuf          *addr;
struct netbuf *
uaddr2taddr(config, uaddr)
struct netconfig *config;
char *uaddr;
```

```
int
netdir_options(netconfig, option, fd, pointer_to_args)
struct netconfig       *netconfig;
int                    option;
int                    fd;
char                   *point_to_args;
void
netdir_perror(s)
char    *s;
char *
netdir_sperror()
```

netdir_getbyname

The *netdir_getbyname()* routine maps the machine and service name specified in the *nd_hostserv* structure to a collection of addresses of the type understood by the transport identified in the *netconfig* structure. The *nd_addrlist* parameter returns a pointer to the addresses. Each address has the form of a *netbuf* structure. Usually, the list contains only an address, e.g. in the form of the Internet address for the machine and the TCP port number for the service.

netdir_getbyaddr

The *netdir_getbyaddr()* routine maps addresses into machine and service names. Given an address in the *netbuf* parameter, this routine returns a list of machine name and service name pairs that have this address (a pointer to the list of machine and service name pairs is returned in the *nd_hostservlist* parameter).

netdir_free

The *netdir_free()* routine frees the structures allocated by the name-to-address translation routines. The *type* parameter can take the following values:

ND_HOSTSERVLIST	Pointer to *nd_hostservlist* structure allocated by *netdir_getbyaddr*()
ND_ADDRLIST	Pointer to *nd_addrlist* structure allocated by *netdir_getbyname*()

taddr2uaddr

The *taddr2uaddr()* routine translates the address given in the *netbuf* structure and returns a "universal address" representation of the address. A universal address is a machine architecture-independent character representation of the address.

uaddr2taddr

The *uaddr2taddr* routine translates a universal address, i.e. a string, back into a *netbuf* structure. The *netconfig* parameter specifies which transport service the address is valid for.

netdir_options

The *netdir_options()* routine provides an interface to transport service-specific capabilities (for example, if application programs want to send a broadcast and check first whether the transport service used supports this).

The *netconfig* structure specifies a transport service. The *option* argument specifies the transport service-specific action to take. The third argument is a file descriptor (which may or may not be used depending upon *option*). The fourth argument is a a pointer to operation-specific data.

The following values may be used for *option*:

ND_SET_BROADCAST
: This option sets the transport provider for the sending of broadcasts (provided the transport provider supports broadcasts). *fd* is a file descriptor into the transport provider (for example, the result of a

t_open() on */dev/udp*). *point_to_args* is not used. If the operation is successful, broadcast operations can be executed on *fd*.

ND_SET_RESERVEDPORT
: This option binds the *fd* file descriptor to a reserved port. *fd* is a file descriptor for the transport service and must not yet be bound. If *point_to_args* is NULL, *fd* will be bound to a freely selected reserved port. If *point_to_args* is a pointer to a *netbuf* structure, an attempt will be made to bind *fd* to the specified address.

ND_CHECK_RESERVEDPORT
: This concept is used to verify that an address corresponds to a reserved port. *fd* is not used. *point_to_args* is a pointer to a *netbuf* structure that contains the address. This option returns 0 only if the address specified in *point_to_args* is reserved.

ND_MERGEADDR
: This option is used to transform a locally meaningful address into an address that client machines can use, i.e. to which a client can initiate connection establishment. For example, the Transmission Control Protocol (TCP) has a local address of 0.0.0.0. ND_MERGEADDR can be used to translate the 0.0.0.0 address into the "real" address of the machine that is understood by client machines. *fd* is not used with this option. *point_to_args* is a pointer to an *nd_mergearg* structure, which has the following form:

```
struct nd_mergearg {
char *s_uaddr;  /* server's universal address */
char *c_uaddr;  /* client's universal address */
char *m_uaddr;  /* the result */
}
```

netdir_perror

The *netdir_perror()* routine writes to standard output an error message stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

netdir_sperror

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

5.2.3 Program example

The following example demonstrates the use of the transport service selection and name-to-address mapping functions:

```
#include <netconfig.h>
#include <netdir.h>
#include <tiuser.h>
/***  The following is a segment of a procedure.  ***/
struct nd_hostserv  nd_hostserv;  /* contains host and service
* information */
struct netconfig   *netconfigp;   /* contains information about
* each network */
struct nd_addrlist *nd_addrlistp; /* list of addresses for the
* service */
struct netbuf  *netbufp;    /* the address of the service */
int            i;           /* counts the number of addresses */
void           *handlep;    /* a handle into transport service
* selection */
/*
* Set the host structure to reference the "listen" service on
* machine "gandalf"
*/
nd_hostserv.h_host = "gandalf";
```

```
nd_hostserv.h_serv = "listen";
/*
*  Initialize the transport service selection mechanism.
*/
if ((handlep = setnetpath()) == NULL) {
nc_perror("setnetpath failed");
exit(1);
}
/*
* Check the transport services.
*/
while ((netconfigp = getnetpath(handlep)) != NULL) {
/*
*   Print out the information associated with the transport
*   services described in the "netconfig" structure.
*/
printf("\nTransport service name: %s\n",   netconfigp->nc_netid);
printf("Transport protocol family: %s\n",
netconfigp->nc_protofmly);
printf("The transport service special file: %s\n",
netconfigp->nc_device);
printf("Transport service semantics: ");
switch (netconfigp->nc_semantics) {
case NC_TPI_COTS:
printf("virtual connection\n");
break;
case NC_TPI_COTS_ORD:
printf("virtual connection with orderly release\n");
break;
case NC_TPI_CLTS:
printf("datagram\n");
break;
}
/*
* Get the addresses for service "listen" on machine
* "gandalf" over the transport service specified in the
* netconfigp structure.
*/
```

```
if (netdir_getbyname(netconfigp, &nd_hostserv,
&nd_addrlistp) != 0) {
printf("Cannot determine the address for the service\n");
netdir_perror("netdir_getbyname failed");
continue;
}
printf("There are <%d> addresses for the date service on
gandalf:\n",
nd_addrlistp->n_cnt);
/*
* Print out all addresses for service "listen" on machine
* "gandalf" on the current transport service.
*/
netbufp = nd_addrlistp->n_addrs;
for (i = 0; i < nd_addrlistp->n_cnt; i++) {
printf("%s\n", taddr2uaddr(netconfigp, netbufp));
netbufp++;
}
}
endnetconfig(handlep);
```

6 XDR/RPC protocol specification

6.1 Introduction to XDR

XDR is a standard for the description and encoding of data. The XDR protocol is useful for transferring data between different computer architectures. XDR fits into the ISO presentation layer and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between the two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats and can only be used to describe data; it is not a programming language. This language makes it possible to describe intricate data formats in a concise manner. The XDR language is similar to the C language. Protocols such as RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data.

6.1.1 Basic block size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. The n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four.

Choosing the XDR block size requires a tradeoff. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that are not aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too large. Four was chosen as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

The same data should encode into the same thing on all machines, so that encoded data can be significantly compared or checksummed. Forcing the padded bytes to be zero ensures this.

This chapter uses graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more additional bytes where required:

A block

```
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)>----------->|
```

6.1.2 Organization of technical information

XDR

The section entitled "XDR data type declarations" in this chapter describes the basic data types that can be represented with XDR. "Other XDR declarations" describes constants, type definitions and "optional data" (an alternative way of representing specific types of unions). "The XDR language specification" provides a formal definition of the XDR language. "An example of an XDR data description" shows how XDR can be used to describe a file.

RPC

The section entitled "RPC protocol specification" specifies the message protocol used by the RPC package. "RPC protocol requirements" contains information on the functionality allowed for by the RPC protocol and the additional functions the RPC package provides. "The RPC message protocol" defines the RPC message protocol using XDR language expressions. "Authentication protocols" describes authentication functions supported by the RPC package. "Record marking with RPC" describes how RPC messages are distinguished

from each other when they are transmitted using a bytestream protocol such as TCP/IP. "The RPC language" contains an example of an RPC service, followed by a formal definition of the RPC language. "The rpcbind protocol" describes the interface to the *rpcbind* service.

6.2 XDR data type declarations

Each of the sections that follow:

- describe a data type defined in the XDR standard
- show how that data type is declared in the language
- include a graphic illustration of the encoding

For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variable-length sequences of data and square brackets ([and]) denote fixed-length sequences of data. *n*, *m* and *r* denote integers. For the full language specification and more formal definitions of terms such as *identifier* and *declaration*, refer to "The XDR language specification".

For some data types, more specific examples are included. A more extensive example of a data description is in the section entitled "An example of an XDR data description".

6.2.1 Integer

Description

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648, 2147483647]. The integer is represented in two's complement notation; the most and least significant bytes are 0 and 3, respectively.

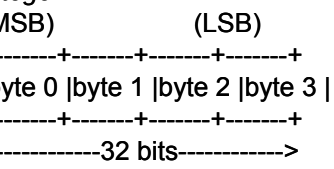
Declaration

Integers are declared as follows:

int *identifier*;

Encoding

Integer



6.2.2 Unsigned Integer

Description

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0, 4294967295]. The integer is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively.

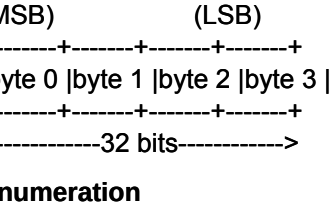
Declaration

An unsigned integer is declared as follows:

unsigned int *identifier*;

Encoding

Unsigned integer



6.2.3 Enumeration

Description

Enumerations have the same representation as signed integers and are handy for describing subsets of the integers.

Declaration

Enumerated data is declared as follows:

enum { *name-identifier* = *constant*, ... } *identifier*;

For example, an enumerated type could represent the three colors red, yellow, and blue as follows:

enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;

It is an error to assign to an enum an integer that has not been assigned in the enum declaration.

Encoding

See the Section "Integer" above.

6.2.4 Boolean

Description

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are integers of value 0 or 1.

Declaration

Booleans are declared as follows:

bool *identifier*;

This is equivalent to:

enum { FALSE = 0, TRUE = 1 } identifier;

Encoding

See the Section "Integer" above.

6.2.5 Hyper integer and unsigned hyper integer

Description

The standard also defines 64-bit (8-byte) numbers called hyper int and unsigned hyper int whose representations are the obvious extensions of integer and unsigned integer, defined above. They are represented in two's complement notation; the most and least significant bytes are 0 and 7, respectively.

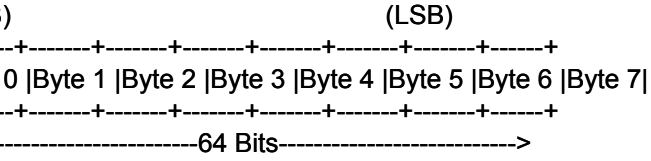
Declaration

Hyper integers are declared as follows:

hyper int *identifier*;
unsigned hyper int *identifier*;

Encoding

Hyper integer



6.2.6 Floating-point

Description

The standard defines the floating-point data type float (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [1]. The following three fields describe the single-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. Eight bits are devoted to this field. The exponent is biased by 127.

F: The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating-point number is described by:

(-1)**S * 2**(E-Bias) * 1.F

Declaration

Single-precision floating-point data is declared as follows:

float *identifier*;

Encoding

Single-precision floating-point

```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
S|   E   |          F          |
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

Just as the most and least significant bytes of an integer are 0 and 3, the most and least significant bits of a single-precision floating-point number are 0 and □31.

The beginning bit (and most/least significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 9, respectively.

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the NaN (not a number) is system dependent and should not be used externally.

6.2.7 Double-precision floating-point

Description

The standard defines the encoding for the double-precision floating-point data type double (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers. The standard defines the following three fields, which describe the double-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

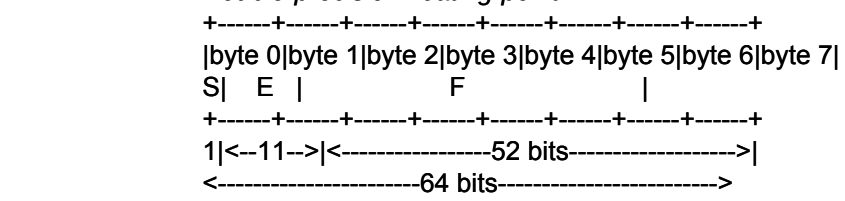
(-1)**S * 2**(E-1023) * 1.F

Declaration

double *identifier:*

Encoding

Double-precision floating-point



Just as the most and least significant bytes of an integer are 0 and 3, the most and least significant bits of a double-precision floating-point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively.

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the NaN (not a number) is system dependent and should not be used externally.

6.2.8 Fixed-length opaque data

Description

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called opaque.

Declaration

opaque data is declared as follows:

opaque identifier[*n*];

where the constant *n* is the (static) number of bytes necessary to contain the opaque data.

Encoding

The *n* bytes are followed by enough (0 to 3) residual zero bytes, *r*, to make the total byte count of the opaque object a multiple of four.

Fixed-length opaque

```
0        1     ...
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)------------>|
```

6.2.9 Variable-length opaque data

Description

The standard also provides for variable-length (counted) opaque data, defined as a sequence of *n* (numbered 0 through *n*-1) arbitrary bytes to be the number *n* encoded as an unsigned integer (as described below), and followed by the *n* bytes of the sequence.

Byte *b* of the sequence always precedes byte *b*+1 of the sequence, and byte 0 of the sequence always follows the length field. The *n* bytes are followed by enough (0 to 3) residual zero bytes, *r*, to make the total byte count a multiple of four.

Declaration

Variable-length opaque data is declared in the following way:

opaque *identifier*<*m*>;

or

opaque *identifier*<>;

The constant *m* denotes an upper bound of the number of bytes that the sequence may contain. If *m* is not specified, as in the second declaration, it is assumed to be (2**32)-1, the maximum length. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

opaque filedata<8192>;

Encoding

Variable-length opaque

```
0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
|<----n+r (where (n+r) mod 4 = 0)---->|
```

It is an error to encode a length greater than the maximum described in the specification.

6.2.10 String

Description

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte b of the string always precedes byte b+1 of the string, and byte 0 of the string always follows the string's length. The n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four.

Declaration

Counted byte strings are declared as follows:

string *object*<*m*>;

or

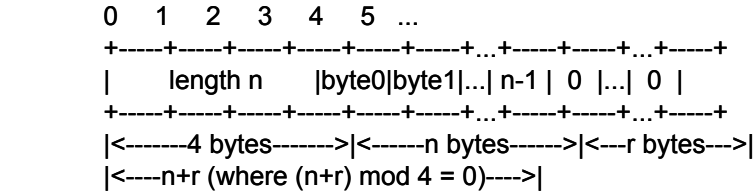
string *object*<>;

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32)-1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

string filename<255>;

Encoding

String



It is an error to encode a length greater than the maximum described in the specification.

6.2.11 Fixed-length array

Description

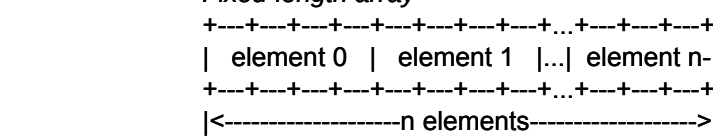
Fixed-length arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type string, yet each element will vary in its length.

Declaration

Declarations for fixed-length arrays of homogeneous elements are in the following form:

type-name identifier[n];

Fixed-length array



6.2.12 Variable-length array

Description

An array of variable length is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element *n*-1.

Declaration

The declaration for variable-length arrays follows this form:

type-name identifier<m>;

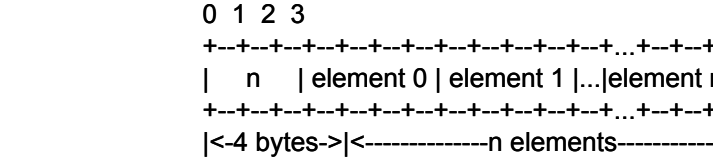
or

type-name identifier<>;

The constant m specifies the maximum acceptable element count of an array. Note that if *m* is not specified, as is the case in the second declaration format above, it is assumed to be (2**32)-1.

Encoding

Counted array



It is an error to encode a value of *n* that is greater than the maximum described in the specification.

6.2.13 Structure

Description

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

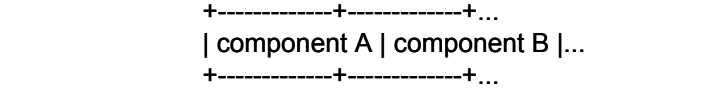
Declaration

Structures are declared as follows:

struct {
component-declaration-A;
component-declaration-B;
...
} *identifier*;

Encoding

Structure



6.2.14 Discriminated union

Description

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either int, unsigned int, or an enumerated type, such as bool. The component types are called *arms* of the union, and are preceded by the value of the discriminant which implies their encoding.

Declaration

Discriminated unions are declared as follows:

union switch (*discriminant-declaration*) {
case *discriminant-value-A*:
arm-declaration-A;
case *discriminant-value-B*:
arm-declaration-B;
...
default:
default-declaration;
} *identifier*;

Each case keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

Encoding

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated union

```
0   1   2   3
+---+---+---+---+---+---+---+---+
|  discriminant |  implied arm  |
+---+---+---+---+---+---+---+---+
|<---4 bytes--->|
```

6.2.15 Void

Description

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not.

Declaration

The declaration is simply as follows:

void;

Encoding

Voids are illustrated as follows:

```
++
||
++
--><-- 0 bytes
```

6.3 Other XDR declarations

6.3.1 Constant

The declaration for a constant follows this form:

const *name-identifier* = *n*;

const is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used.

The following example defines a symbolic constant DOZEN, equal to 12.

const DOZEN = 12;

6.3.2 typedef

typedef does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

typedef *declaration*;

The new type name is actually the variable name in the declaration part of typedef.

The following example defines a new type called eggbox using an existing type called egg and the symbolic constant DOZEN:

typedef egg eggbox[DOZEN];

Variables declared using the new type name have the same type as the new type name would have in the typedef, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable fresheggs:

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

typedef <<*struct*, *union*, *or enum definition*>> *identifier*;

may be converted to the alternative form by removing the typedef part and placing the identifier after the struct, enum, or union keyword, instead of at the end. For example, here are the two ways to define the type bool:

```
typedef enum {          /* using typedef */
FALSE = 0,
TRUE = 1
} bool;
enum bool {             /* preferred alternative *
FALSE = 0,
TRUE = 1
};
```

This syntax is preferred because one does not have to go to the end of a declaration to learn the name of the new type.

6.3.3 Optional-data

Optional-data is a form of union. Because it occurs frequently, it has been given its own declaration syntax. It is declared as follows:

*type-name *identifier*;

This is equivalent to the following union:

```
union switch (bool opted) {
case TRUE:
type-name element;
case FALSE:
void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, because the boolean opted can be interpreted as the length of the array:

type-name identifier<1>;

Optional-data is useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type stringlist that encodes lists of arbitrary length strings:

```
struct *stringlist {
string item<>;
stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
case TRUE:
struct {
string item<>;
stringlist next;
} element;
case FALSE:
void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
string item<>;
stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. The syntax is the same as that of the C language for pointers.

6.4 The XDR language specification

6.4.1 Notational conventions

This specification uses a modified Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

- The characters |, (,), [,], and * are special.
- Terminal symbols are strings of any characters in a constant-width font.
- Non-terminal symbols are strings of non-special *italic* characters.
- Alternative items are separated by a vertical bar (|).
- Optional items are enclosed in brackets.
- Items are grouped together by enclosing them in parentheses.
- A * following an item means 0 or more occurrences of the item.

For example, consider the following pattern:

a very (, very)* [cold and] rainy (day | night)

An infinite number of strings match this pattern. A few of them are:

a very rainy day
a very, very rainy day
a very cold and rainy day
a very, very, very cold and rainy night

6.4.2 Lexical notes

- Comments begin with /* and end with */.
- White space serves to separate items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits or underbars (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign (-).

6.4.3 Syntax information

declaration:
type-specifier identifier
| *type-specifier identifier* [*value*]
| *type-specifier identifier* < [*value*] >
| opaque *identifier* [*value*]
| opaque *identifier* < [*value*] >
| string *identifier* < [*value*] >
| *type-specifier* * *identifier*
| void
value:
constant
| *identifier*
type-specifier:
[unsigned] int
| [unsigned] hyper
| float
| double
| bool
| *enum-type-spec*
| *struct-type-spec*
| *union-type-spec*

| *identifier*
enum-type-spec:
enum enum-body
enum-body:
{
(*identifier* = *value*)
(, *identifier* = *value*)*
}
struct-type-spec:
struct struct-body
struct-body:
{
(*declaration* ;)
(*declaration* ;)*
}
union-type-spec:
union union-body
union-body:
switch (*declaration*) {
(case *value* : *declaration* ;)
(case *value* : *declaration* ;)*
[default : *declaration* ;]
}

constant-def:
const *identifier* = *constant* ;
type-def:
typedef *declaration* ;
| enum *identifier enum-body* ;
| struct *identifier struct-body* ;
| union *identifier union-body* ;
definition:
type-def
| *constant-def*
specification:
definition *

Syntax notes

The following are keywords and cannot be used as identifiers:

bool	const	enum	int	struct	union
case	default	float	opaque	switch	unsigned
char	double	hyper	string	typedef	void

- Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a const definition.
- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
- Similarly, variable names must be unique.
- The discriminant of a union must be of a type that evaluates to an integer. That is, int, unsigned int, bool, an enum type or any typedefed type that leads to one of these. Also, the case values must be legal discriminant values. Finally, a case value may not be specified more than once within the scope of a union declaration.

6.5 An example of an XDR data description

Here is a short XDR data description of an object called file, which might be used to transfer files from one machine to another:

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file */
const MAXNAMELEN = 255;     /* max length of a file name */
/* Types of files: */
enum filekind {
TEXT = 0,       /* ascii data */
DATA = 1,       /* raw data   */
EXEC = 2        /* executable */
};
/* File information, per kind of file: */
union filetype switch (filekind kind) {
case TEXT:
void;                       /* no extra information */
case DATA:
string creator<MAXNAMELEN>;     /* data creator */
case EXEC:
string interpretor<MAXNAMELEN>; /* program interpretor  */
};
/* A complete file: */
struct file {
string filename<MAXNAMELEN>; /* name of file */
filetype type;              /* info about file */
string owner<MAXUSERNAME>;   /* owner of file */
opaque data<MAXFILELEN>;     /* file data */
};
```

Suppose now that there is a user named john who wants to store his lisp program sillyprog that contains just the data (quit). His file would be encoded as follows:

Offset	Hex bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpreter = 4
24	6c 69 73 70	lisp	Interpreter characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

6.6 RPC protocol specification

This chapter specifies the message protocol used by the RPC package. The message protocol is specified in the XDR language (External Data Representation). This chapter assumes the reader is familiar with this (see the section entitled "The XDR language specification").

6.6.1 General attributes of the protocol

The RPC model

The RPC model is similar to a local procedure call. The local caller places arguments to a procedure in some well-specified location. It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from a well-specified location, and the caller continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes. One is the caller's process, the other is a server's process. Conceptually, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this description, only one of the two processes is active at any given time. However, this need not be the case. The RPC protocol makes no restrictions on the concurrency model implemented. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming requests, so that the server can be free to receive other requests.

Transport services and semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not attempt to ensure transport reliability. In this regard, the application must be aware of the type of transport protocol underneath RPC. If the RPC service knows it is running on top of a reliable transport such as TCP/IP, then most of the work is already done for it. On the other hand, if RPC is running on top of an unreliable transport such as UDP/IP, the service must devise its own retransmission and time-out policy. RPC does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short timeouts, the only thing it can infer if it receives no reply is that the procedure was executed either not at all or several times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction ID is by the RPC client for matching replies to requests. A client uses the previously used transaction ID when retransmitting a request. The server can remember this ID after accepting a request in order to reject further requests with the same ID. The server is not allowed to evaluate this ID in any other way.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

Binding and rendezvous independence

The act of binding a client to a service is **not** part of the remote procedure call specification. This important

and necessary function is left up to some higher-level software. (This software may use RPCs; see the section entitled "The rpcbind protocol".)

Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be implemented on the basis of this concept. Various authentication protocols can be supported. A field in the RPC header - the authentication type - specifies the protocol being used. More information on authentication protocols can be found in the section entitled "Authentication protocols".

6.7 RPC protocol requirements

The RPC protocol provides for the following:

- Unique specification of a procedure to be called
- Provisions for matching response messages to request messages
- Provisions for authenticating the caller to service and vice versa

In addition, the RPC package allows the following error situations to be detected:

- The partners are using different RPC protocols.
- The partner does not support the desired service version.
- Protocol errors (such as misspecification of a procedure's parameters)
- Reasons why remote authentication failed

6.7.1 Programs and procedures

The RPC call message has three unsigned fields:

- the RPC program number
- the version number of the RPC program
- the number of the remote procedure

The three fields uniquely identify the procedure to be called.

Program numbers are administered by a central authority (see the Section "Program number assignment").

The first implementation of a program will most likely have version number 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies the version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is "read" and procedure number 12 is "write".

Just as remote program protocols may change over several versions, the RPC message protocol itself may change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the version of RPC described here.

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
- The server cannot decode the parameters of the remote procedure. (Again, this is usually caused by a disagreement about the protocol between client and server.)

6.7.2 Authentication

Provisions for authentication of caller to server and vice versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
AUTH_NONE   = 0,
AUTH_SYS    = 1,
AUTH_SHORT  = 2,
/* and more to be defined */
};
struct opaque_auth {
enum_t   oa_flavor;    /* style of credentials */
caddr_t   oa_base;     /* address of more auth stuff */
u_int   oa_length;     /* not to exceed MAX_AUTH_BYTES */
};
```

Each *opaque_auth* structure contains an *auth_flavor* as an authentication type, followed by the actual authentication as opaque data. The interpretation and semantics of the authentication data are specified by individual, independent protocol specifications. (See "Authentication protocols" for definitions of the various authentication protocols.)

If authentication parameters are rejected, the response message contains information stating why they are rejected.

6.7.3 Program number assignment

Program numbers are given out in groups of 0x20000000 according to the following chart:

Program numbers	Description
0 - 1fffffff	Defined by Sun
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

Sun Microsystems administers the first group of numbers, which should be identical for all UNIX® System V customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.

The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs.

The third group is reserved for applications that generate program numbers dynamically.

The final groups are reserved for future use, and should not be used.

To register a protocol specification, send a request by E-mail to

rpc@sun.com

or write to:

RPC Administrator

Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Please include a compilable *rpcgen .x* describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard RPC services can be found in the include files in */usr/include/rpcsvc*. These services, however, constitute only a small subset of those that have been registered.

6.7.4 Other uses of the RPC protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. In the RPC package, these are batching and RPC broadcasts.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP/IP) for its transport. In batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually finished by a non-batch RPC call. This is followed by positive acknowledgment.

RPC broadcasts

In RPC-based broadcasts, the client sends a broadcast packet to the network and waits for numerous replies. RPC broadcasts use non-reliable, packet-based protocols (like UDP/IP) for transport purposes. Servers that support broadcast protocols only respond when the call is successfully processed. RPC broadcasts use the *rpcbind* service to implement semantics. See "The rpcbind protocol" for more information.

6.8 The RPC message protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style.

```
enum msg_type {
CALL  = 0,
REPLY = 1
};
/*
* A reply to a call message can take on two forms:
* The message was either accepted or rejected.
*/
enum reply_stat {
MSG_ACCEPTED = 0,
MSG_DENIED   = 1
};
/*
* Given that a call message was accepted, the following is the
* status of an attempt to call a remote procedure.
*/
enum accept_stat {
SUCCESS       = 0, /* RPC executed successfully       */
PROG_UNAVAIL  = 1, /* remote hasn't exported program  */
PROG_MISMATCH = 2, /* remote can't support version #  */
PROC_UNAVAIL  = 3, /* program can't support procedure */
GARBAGE_ARGS  = 4  /* procedure can't decode params   */
};
/*
```

```
* Reasons why a call message was rejected:
*/
enum reject_stat {
RPC_MISMATCH = 0, /* RPC version number != 2          */
AUTH_ERROR = 1    /* remote can't authenticate caller */
};
/*
* Why authentication failed:
*/
enum auth_stat {
AUTH_BADCRED      = 1,  /* bad credentials */
AUTH_REJECTEDCRED = 2,  /* client must begin new session */
AUTH_BADVERF      = 3,  /* bad verifier */
AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed  */
AUTH_TOOWEAK      = 5   /* rejected for security reasons */
};
/*
* The RPC message:
* All messages start with a transaction identifier, xid,
* followed by a two-armed discriminated union.  The union's
* discriminant is a msg_type which switches to one of the two
* types of the message.  The xid of a REPLY message always
* matches that of the initiating CALL message.  NB: The xid
* field is only used for clients matching reply messages with
* call messages or for servers detecting retransmissions; the
* service side cannot treat this id as any type of sequence
* number.
*/
```

```
struct rpc_msg {
unsigned int xid;
union switch (msg_type mtype) {
case CALL:
call_body cbody;
case REPLY:
reply_body rbody;
} body;
};
/*
* Body of an RPC request call:
* In version 2 of the RPC protocol specification, rpcvers must
* be equal to 2.  The fields prog, vers, and proc specify the
* remote program, its version number, and the procedure within
* the remote program to be called.  After these fields are two
* authentication parameters: cred (authentication credentials)
* and verf (authentication verifier).  The two authentication
* parameters are followed by the parameters to the remote
* procedure, which are specified by the specific program
* protocol.
*/
struct call_body {
unsigned int rpcvers;  /* must be equal to two (2) */
unsigned int prog;
unsigned int vers;
unsigned int proc;
opaque_auth cred;
opaque_auth verf;
/* procedure specific parameters start here */
};
/*
* Body of a reply to an RPC request:
* The call message was either accepted or rejected.
*/
union reply_body switch (reply_stat stat) {
case MSG_ACCEPTED:
accepted_reply areply;
case MSG_DENIED:
rejected_reply rreply;
} reply;
```

```
/*
* Reply to an RPC request that was accepted by the server:
* there could be an error even though the request was accepted.
* The first field is an authentication verifier that the server
* generates in order to validate itself to the caller.  It is
* followed by a union whose discriminant is an enum
* accept_stat.  The SUCCESS arm of the union is protocol
* specific.  The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP
* arms of the union are "void". The PROG_MISMATCH arm specifies
* the lowest and highest version numbers of the remote program
* supported by the server.
*/
struct accepted_reply {
opaque_auth verf;
union switch (accept_stat stat) {
case SUCCESS:
opaque results[0];
/* procedure-specific results start here */
case PROG_MISMATCH:
struct {
unsigned int low;
unsigned int high;
} mismatch_info;
default:
/*
* Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL,
* and GARBAGE_ARGS.
*/
void;
} reply_data;
};
/*
* Reply to an RPC request that was rejected by the server:
* The request can be rejected for two reasons: either the
* server is not running a compatible version of the RPC
* protocol (RPC_MISMATCH), or the server refuses to
* authenticate the caller (AUTH_ERROR).  In case of an RPC
* version mismatch, the server returns the lowest and highest
* supported RPC version numbers.  In case of refused
* authentication, failure status is returned.
```

```
*/
union rejected_reply switch (reject_stat stat) {
case RPC_MISMATCH:
struct {
unsigned int low;
unsigned int high;
} mismatch_info;
case AUTH_ERROR:
auth_stat stat;
};
```

6.9 Authentication protocols

This section defines some "flavors" of authentication that have already been implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

6.9.1 AUTH_NONE authentication

Calls are often made where the caller does not authenticate itself and the server does not care who the caller is. In these cases, the type (of the discriminant of the *opaque_auth* union) of the RPC message's credentials, verifier, and response verifier is AUTH_NONE. The bytes of the *body* field in the *opaque_auth* structure are undefined. It is recommended that the *body* length be zero when AUTH_NONE authentication is used.

6.9.2 AUTH_SYS authentication

The caller of a remote procedure may wish to identify itself using traditional System V process permissions authentication. The *flavor* of the *opaque_auth* of such an RPC call message is AUTH_SYS. The bytes of the body encode the following structure:

```
struct auth_sysparms {
unsigned int stamp;
string machinename<255>;
uid_t uid;
gid_t int gid;
gid_t int gids<10>;
};
```

stamp	is an arbitrary value that the client machine may generate
machinename	is the name of the client machine (like krypton).
uid	is the caller's actual ID.
gid	is the caller's actual group ID.
gids	is a list of groups to which the caller belongs.

The *flavor* of the verifier accompanying the credentials should be AUTH_NONE (defined above).

The AUTH_SHORT verifier

When using AUTH_SYS authentication, the type of the response verifier received in the reply message from the server may be AUTH_NONE or AUTH_SHORT. In the case of AUTH_SHORT, it consists of a *short_hand_verf* structure. This *opaque* structure may now be passed to the server instead of the original AUTH_SYS credentials. The server keeps a cache that maps shorthand opaque structures (passed back by way of an AUTH_SHORT style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials. The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected owing to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_SYS credentials again.

6.10 Record marking with RPC

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another to detect and possibly recover from user protocol errors. This is called record marking (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment consists of a four-byte header followed by 0 to (2**31)-1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest.

The header encodes two values

- a boolean that specifies whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment)
- a 31-bit unsigned binary value that is the length in bytes of the fragment's data.

The boolean value is the highest-order bit of the header; the length is the 31 low-order bits.

This record specification is not in XDR standard form.

6.11 The RPC language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language.

6.11.1 An example service described in the RPC language

Here is an example of the specification of a simple ping program.

```
/*
* Simple ping program
*/
program PING_PROG {
/* Latest and greatest version */
version PING_VERS_PINGBACK {
void
PINGPROC_NULL(void) = 0;
/*
* Ping the caller, return the round-trip time
* (in microseconds).  Returns -1 if the operation
* timed out.
*/
int
PINGPROC_PINGBACK(void) = 1;
} = 2;
/*
* Original version
```

```
*/
version PING_VERS_ORIG {
void
PINGPROC_NULL(void) = 0;
} = 1;
} = 1;
} = 200000;
const PING_VERS = 2;      /* latest version */
```

The first version described is PING_VERS_PINGBACK with two procedures, PINGPROC_NULL and PINGPROC_PINGBACK. PINGPROC_NULL takes no arguments and returns no results, but it is useful for such things as computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the same semantics, and never require authentication.

The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it returns the amount of time (in microseconds) that the operation used.

The next version, PING_VERS_ORIG, is the original version of the protocol and it does not contain the PINGPROC_PINGBACK procedure. It is useful for compatibility with old client programs.

6.11.2 The RPC language specification

The RPC language is identical to the XDR language, except for the added definitions described below.

program-definition:
program program-ident {
version-list
} = value
version-list: version ;
version ; version-list
version:
version version-ident {
procedure-list
} = value
procedure-list:
procedure ;
procedure ; procedure-list
procedure:
type-ident procedure-ident (type-ident) = value

6.11.3 Syntax notes

The following keywords are added and cannot be used as identifiers: program and version.

A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.

A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.

Program identifiers are in the same name space as constant and type identifiers.

Only unsigned constants can be assigned to programs, versions and procedures.

6.12 The rpcbind protocol

rpcbind maps RPC program and version numbers to universal addresses.

rpcbind is accessible at a well-known universal address, and other programs register their dynamically allocated transport addresses with it. It then makes those addresses publicly available. Universal addresses are string representations of the transport address.

rpcbind also aids in broadcast RPC. There is no fixed relationship between the addresses that a given RPC program will have on different machines, so there is no way to broadcast directly to all these programs.

rpcbind, however, has a universal address. So, to broadcast to a given program, the client actually sends its message to the *rpcbind* process on the machine it wishes to reach.

rpcbind picks up the broadcast and calls the local service specified by the client. When *rpcbind* gets a reply from the local service, it passes it on to the client.

6.12.1 The rpcbind protocol specification (in the RPC language)

The Version 3 protocol (RFC 1833) is explained below. However, the Version 2 protocol (RFC 1050) can also be used.

```
/*
* rpcb_prot.x
* RPCBIND protocol in rpc language
*/
/*
* A mapping of (program, version, network ID) to universal
* address
*/
struct rpcb {
u_long r_prog;       /* program number */
u_long r_vers;       /* version number */
string r_netid<>;    /* transport service name */
string r_addr<>;     /* universal address */
string r_owner<>;    /* owner of service */
};
/*
* A list of mappings
*/
struct rpcblist {
rpcb rpcb_map;
struct rpcblist *rpcb_next;
};
/*
* Arguments of remote calls
*/
struct rpcb_rmtcallargs {
u_long prog;             /* program number */
u_long vers;             /* version number */
u_long proc;             /* procedure number */
opaque args_ptr<>;       /* argument */
};
/*
* Results of the remote call
*/
struct rpcb_rmtcallres {
string addr_ptr<>;       /* remote universal address */
opaque results_ptr<>;    /* result */
};
/*
* rpcbind procedures
*/
program RPCBPROG {
version RPCBVERS {
void
RPCBPROC_NULL(void) = 0;
```

```
bool
RPCBPROC_SET(rpcb) = 1;
bool
RPCBPROC_UNSET(rpcb) = 2;
string
RPCBPROC_GETADDR(rpcb) = 3;
rpcblist
RPCBPROC_DUMP(void) = 4;
rpcb_rmtcallres
RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;
unsigned int
RPCBPROC_GETTIME(void) = 6;
struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;
string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;
} = 3;
} = 100000;
```

6.12.2 rpcbind protocol explanations

rpcbind is contacted by way of an assigned address specific to the transport being used. For TCP and UDP, for example, it is port number 111. Each transport has such an assigned well known address. The following is a description of each of the procedures supported by *rpcbind*.

The RPCBPROC_NULL procedure

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

The RPCBPROC_SET procedure

When an RPC service first becomes available on a machine, it registers itself with the *rpcbind* program running on the same machine. The program passes its program number *prog*, version number *vers*, transport service name *netid*, and the universal address *uaddr* on which it awaits calls.

The procedure returns a boolean response whose value is TRUE if the procedure successfully established the mapping and FALSE otherwise. The procedure rejects registration if registration already exists for *prog*, *vers* or *netid*.

Note that neither *netid* nor *uaddr* can be NULL, and that *netid* should be a valid network identifier on the machine making the call.

For security reasons this procedure may only be executed using a loopback transport.

The RPCBPROC_UNSET procedure

When a program terminates, it should unregister itself with the *rpcbind* program on the same machine. The parameters and results have meanings identical to those of RPCBPROC_SET. The mapping of *prog*, *vers* and *netid* to *uaddr* is deleted.

If *netid* is NULL, all mappings specified by the tuple (*prog*, *vers*, *) and the corresponding universal addresses are deleted.

For security reasons this procedure may only be executed using a loopback transport.

The RPCBPROC_GETADDR procedure

If the program number *prog*, version number *vers*, and transport service name *netid* are passed to this procedure, it returns the universal address on which the program is awaiting call requests.

The *netid* field of the argument is ignored and the *netid* is inferred from the *netid* of the transport service on which the request came in.

The RPCBPROC_DUMP procedure

This procedure lists all entries in *rpcbind*'s database. The procedure takes no parameters and returns a list of program, version, netid, and universal addresses.

The RPCBPROC_CALLIT procedure

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's universal address. It is intended for supporting broadcasts to arbitrary remote programs via *rpcbind*'s universal address.

The parameters *prog*, *vers*, *proc*, and the *args_ptr* are the program number, version number, procedure number, and parameters of the remote procedure.

This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.

The procedure returns the remote program's universal address, and the results of the remote procedure.

The RPCBPROC_GETTIME procedure

This procedure returns the local time on its own machine.

The RPCBPROC_UADDR2TADDR procedure

This procedure converts universal addresses to transport (*netbuf*) addresses. RPCBPROC_UADDR2TADDR is equivalent to *uaddr2taddr()*. See the description of *netdir(3N)* in the "Networking Reference Manual".

Only processes that can not link to the name-to-address library modules should use RPCBPROC_UADDR2TADDR.

The RPCBPROC_TADDR2UADDR procedure

This procedure converts transport (*netbuf*) addresses to universal addresses. RPCBPROC_TADDR2UADDR is equivalent to *taddr2uaddr()*. See the description of *netdir(3N)* in the "Networking Reference Manual".

Only processes that can not link to the name-to-address library modules should use RPCBPROC_TADDR2UADDR.

Glossary

Abortive release
An abrupt termination of a transport connection, which may result in the loss of data.

Asynchronous execution
The mode of execution in which transport interface routines will never block while waiting for specific asynchronous events to occur, but instead will return immediately if the event is not pending.

Authentication data
The data by means of which a client identifies itself to a server and vice versa. Which data this is depends on the authentication type.The authentication of a client by a server consists of credentials and a verifier. The verifier checks that the credentials are correct. No verification is provided for in the default Reliant UNIX authentication types.

Authentication type
Defines the structure of authentication data. Reliant UNIX supports the authentication types AUTH_NONE (no authentication), AUTH_SYS (UNIX-specific authentication by user ID and group ID, etc.) and AUTH_SHORT (short authentication).

Binding
The assignment of a transport service-specific address to a transport endpoint.

Bottom level
The lowest of the four lower RPC levels. Programs written on this level can control a large number of transport service-specific details.

Client
An active communication partner that requests the services of a server.

Client handle
Transport endpoint of an RPC client.

Connection establishment
The phase in connection-oriented mode that enables two transport users to create a transport connection between them.

Connection-oriented mode
A connection-oriented mode of transfer in which data are passed from one user to another over an established connection in a reliable, sequenced manner.

Connection release
The phase in connection-oriented mode that terminates a previously established transport connection between two users.

Connectionless mode
A mode of transfer in which data are passed from one user to another in self-contained units (datagrams) with no logical relationship required among multiple units.

Data transfer
The phase in connection-oriented mode or connectionless mode that supports the transfer of data between two transport users.

Datagram
A unit of data transferred between two users of the connectionless service.

Decoding
The conversion of data from XDR format to a machine-specific representation.

Dispatch routine

Every RPC service has a dispatch routine that is called by the RPC library when a request for a service is received. The essential task of this routine is to call the requested remote procedure.

Expedited data

Data that are considered urgent. The specific semantics of expedited data are defined by the transport protocol that provides the transport service.

Expert level

The second lowest of the four lower RPC levels. Programs written on this level control client and server characteristics. Interface to rpcbind; the assignment of the services can be manipulated.

Encoding
The conversion of data from a machine-specific format to XDR format.

Intermediate level
The second highest of the four lower RPC levels. Programs written on this level specify the transport service they want to use.

Orderly connection release
A procedure for terminating a connection in an orderly manner without loss of data.

Peer user
The user with whom a given user communicates by means of the transport interface.

Ping
A procedure call of an RPC program. A ping is used to verify the existence and accessibility of a program via the network. It is also possible to time network communication with ping.

Port monitor
A daemon process that monitors ports and calls the appropriate service when a message arrives at a port.

Pseudo-RPC
Test routines.

RPC
Remote procedure call: a client/server implementation in which the client calls the services of the server as remote procedures analogously to local procedures.

Server
A passive communication partner that offers services to clients and responds to their requests.

Server handle
The transport endpoint of an RPC server.

Service request
A request by a client to a server to execute an action.

Socket
Endpoint of communication between two processes. Also contains the data structure used to implement socket abstraction and the system call used to create a socket.

STREAMS
A UNIX kernel mechanism that represents the framework for network services and data communication. STREAMS defines interface standards for the input and output of characters in the UNIX kernel and between the UNIX kernel and the user level. This mechanism consists of internal functions, supporting routines, parts of the UNIX kernel and a number of structures.

Synchronous execution
The mode of execution in which transport interface routines may block while waiting for specific asynchronous events to occur.

Transport, transport provider, transport service, transport layer
Synonymous terms for the components of the operating system that execute end-to-end communication between two peers via a transport protocol.

Transport address
The address of a transport endpoint in a transport service-dependent form.

Transport endpoint
The data structure via which an application communicates with a peer.

Transport interface
The library routines and state transition rules that support the services of a transport protocol.

Transport type
A criterion on the basis of which transport service selection can be made. For example, datagram_v stands for all "visible" connectionless transport services.

Transport user
The user-level application or protocol that accesses the services of the transport interface.

Universal address
The address of a service in a transport service-independent format.

Verifier
Authentication of a server to the client. This is not necessarily a verifier in the narrower sense. It may, for example, be a short name that the client uses in subsequent RPCs. The authentication type of a verifier is decisive in determining how it is interpreted.

XDR
External Data Representation: a machine-independent format for data representation used by RPC clients and servers to exchange their data.

Zombie
A process that has executed the exit system call and no longer exists but whose end status has not yet been received by the parent process (or init).

Related publications

[1] **C/C++**
User Guide and Reference Manual

Target Group
C++ programmers working under Reliant UNIX.

Contents
Description of features and use of the C++ Translator and the library functions specific to C++ in the libraries libcomplex and libC.

[2] **Integrated Software Development Guide**
User Guide

Target Group
Programmers of application software and users who wish to combine application software into packages.

Contents
This manual describes commands, functions and programs from the viewpoint of application programming in the environment and creation of application and driver software packages.

[3] **Programmer's Reference Manual (Reliant UNIX 5.45)**
Programmer's Reference Manual

Target Group
C programmers working under Reliant UNIX with C-DS.

Contents
Description of the commands for program development, library functions and system calls, and a description of a number of header files and C-specific file formats.

[4] **Programmer's Guide (Reliant UNIX 5.44)**
System Interfaces and Tools for Application Programming
User Guide

Target Group
Programmers.

Contents
Description of how to develop and integrate applications under Reliant UNIX. The packaging tools are also described.

[5] **Network Administration (Reliant UNIX 5.45)**
System Administrator's Guide

Target Group
System Administrators

Contents
This manual describes the network administration tasks that arise from the use of the TCP/IP software on Reliant UNIX 5.45 and the Basic Network Utilities (BNU).

[6] **Networking Reference Manual (Reliant UNIX 5.45)**
Description

Target Group
System Administrators and Programmers

Contents
Commands, C functions, device drivers, utilities, and system files that are required to configure and administer computer networks and to write applications that use communication in computer networks.

Ordering manuals

The publications are ordered through your local Siemens office.

Publications for more Information

[7] **Power Programming with RPC,**
Bloomer, John (1992), O'Reilly, Sebastopol, CA.

[8] **Internetworking with TCP/IP,**
Volume III: Client-Server Programming and Applications, BSD Version

Volume IV: Design, Implementation, and Internals; Client-Server Programming and Applications, AT&T TLI Version,

Comer, Douglas E. (1993); Stevens, David L., Prentice Hall, Englewood Cliffs, New Jersey.

[9] **UNIX Network Programming,**
Stevens, Richard W. (1990), Prentice Hall, Englewood Cliffs, New Jersey.

[10] **TCP/IP Illustrated,**
Volume 1: The Protocols,

Volume 2: The Implementation,

Volume 3: T/TCP Illustrated,

Stevens, Richard W., Addison Wesley, Reading, Massachusetts.

[11] **Networking Programming, Volume 1,**
Stevens, Richard W., Prentice Hall

[12] **UNIX System V Network Programming,**
Rago, Stephen A., Addison Wesley

[13] **Gigabit Ethernet,**
Seifert, Rich, Addison Wesley