ISODE CONSORTIUM

ISODE Volume 16

Programmer's Guide

Transport Services

This documentation and the software are based on contributions from multiple sources. Acknowledgements for contribution are given in *ISODE Volume 1: Overview*.

This manual is associated with Version 1.1v3 of February 1994, and was last updated in February 1994.

Contents

List of Tables

List of Figures

Preface

This guide is published in support of the ISODE Consortium Release 1 (IC R1). It is intended for programmers working at the level of the transport and network servicse in an OSI environment.

It is assumed that readers of this guide are already aware of the contents of *ISODE Volume 1: Overview* and *ISODE Volume 9: Programmer's Guide: General Services*. Other guides to other IC R1 products and their uses are listed under Related Publications later in this Preface.

You may also find some material of interest in *ISODE Volume 3: Administrator's Guide: General Services*. This includes a description of the use of the ISODE transport switch that facilitates the provision of end-to-end service under conditions where communications involve different communities using different combinations of transport protocols and network services.

What this Guide Covers

This guide describes the ISODE TSAP (Transport Services Access Point) module. It includes sections on the establishment and release of connections, the transfer of data and the handling of addressing, listening, and other aspects of the service provided.

Software and Documentation Versions

IC R1 is the first major release of ISODE software made by the ISODE Consortium. It is derived, with revisions and additions, from the openly available ISODE 8.0 release.

The release is accompanied by a comprehensive documentation set comprising the following volumes:

Volume	Title
01	*ISODE Volume 1: Overview*
02	*ISODE Volume 2: Administrator's Guide: Installation*
03	*ISODE Volume 3: Administrator's Guide: General Services*
04	*ISODE Volume 4: Administrator's Guide: Message Handling Services*
05	*ISODE Volume 5: Administrator's Guide: Directory Services*
07	*ISODE Volume 7: User's Guide: Message Handling Services*
08	*ISODE Volume 8: User's Guide: Directory Services*
09	*ISODE Volume 9: Programmer's Guide: General Services*
10	*ISODE Volume 10: Programmer's Guide: ASEs*
12	*ISODE Volume 12: Programmer's Guide: X.500 DUA*
13	*ISODE Volume 13: Programmer's Guide: Layer Interfaces*
14	*ISODE Volume 14: Programmer's Guide: X.400 MTA*
15	*ISODE Volume 15: Programmer's Guide: X.400 UA*
16	*ISODE Volume 16: Programmer's Guide: Transport Services*
17	*ISODE Volume 17: Programmer's Guide: ASN.1 Library*
18	*ISODE Volume 18: Programmer's Guide: ASN.1 Tools*

IC R1 is released as number of separate modules. Each module has its own update number which is incremented independently from other modules whenever an update is made to it. The version numbers of the various components may therefore differ from one another.

As of February 1994, the latest versions of the modules and their associated documentation are as follows:

Module	Version	Volume	Last Document Update
IC Base	R1.1	01	June 1993
		02	February 1994
		03	February 1994
		09	February 1994
		10	February 1994
		13	February 1994
		16	February 1994
		17	February 1993
		18	February 1993
IC DSA	R1.1	05	February 1993
IC DUA	R1.1	08	February 1993
		12	February 1993
IC MTA	R1.1	04	February 1994
	R1.1	07	February 1994
		14	February 1993
		15	February 1993
IC SNMP	R1.1	03	February 1994
Contributed	R1.1	-	-

The ISODE Consortium

The ISODE Consortium is a not-for-profit cooperative enterprise. Its mission is to promote and evolve the ISODE implementation of OSI. This is a major package of OSI applications, including source code and documentation, with a worldwide record of operational success.

All individuals and organisations handling ISODE source code are encouraged to join the ISODE Consortium. For more information about the ISODE Consortium, contact:

Headquarters

Address: ISODE Consortium
The Dome
The Square
Richmond
TW9 1DT
UK

Phone: +44 81 332 9091
Fax: +44 81 332 9019

Email

Internet (RFC822)ic-info@isode.com

MHS (X.400): S=ic-info; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI;

If this is not possible, you may also reach the ISODE Consortium at:

US Office

Address: ISODE Consortium
PO Box 200195
Austin
TX 78720
USA

Phone: +1 (512) 338 3340
Fax: +1 (512) 338 3600

Availability

IC R1 is a source release, and is available in the following ways:

1. To organisational members of the ISODE Consortium as a benefit of membership.
2. To educational and non-commercial research organisations, under zero-cost licence from the ISODE Consortium or under licence from a Research Consortium member of the ISODE Consortium.
3. To any organisation as a 30 day evaluation copy from the ISODE Consortium.

Acquisition and use of IC R1 and related materials for any purpose requires a written licence agreement from ISODE Consortium Limited, or a written licence from an organisation licenced by ISODE Consortium Limited to grant such a licence.

Binary products based on IC R1 are available on a wide range of platforms from ISODE Consortium members. Information on these products and where to obtain them is available from the ISODE Consortium.

Bug Reporting

Bug reports on software releases made by the ISODE Consortium are welcomed. These may be sent to the Consortium by any means, but electronic mail to one of the bug reporting addresses is preferred. Please send proposed fixes with the reports if possible. For non-members of the ISODE Consortium, any reports will be acknowledged, but no further action is guaranteed. Any changes resulting from bug reports may be included in future releases.

The bug reporting addresses are:

- `bug-isode@isode.com` for general ISODE reports.
- `bug-quipu@isode.com` for reports on QUIPU (Directory).
- `bug-pp@isode.com` for reports on PP (MHS).
- `bug-snmp@isode.com` for reports on SNMP.
- `bug-doc@isode.com` for reports on the documentation.

The MHS (X.400) equivalents of these addresses are:

- `S=bug-isode; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-quipu; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-pp; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-snmp; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`
- `S=bug-doc; O=ISODE Consortium; P=ISODE; A=MAILNET; C=FI`

Support

Basic support of ISODE Consortium releases is available to all consortium members as a benefit of membership. This includes upgrades and access to a database of bug reports and fixes. The ISODE Consortium also offers some additional support to companies building products specifically based on ISODE Consortium releases.

A range of ISODE-related support services are offered by ISODE Consortium members. A list of such services will also be available from the ISODE Consortium.

Chapter 1

Transport Services

The IC R1 *libisode*(3n) library contains a set of routines which implement the transport services access point (TSAP).

As with most models of OSI services, the underlying assumption is one of a symmetric, asynchronous environment. That is, although peers exist at a given layer, one does not necessarily view a peer as either a client or a server. Further, the service provider may generate events for the service user without the latter entity triggering the actions which led to the event. For example, in a synchronous environment, an indication that data has arrived usually occurs only when the service user asks the service provider to read data; in an asynchronous environment, the service provider may interrupt the service user at any time to announce the arrival of data.

The `tsap` module in this release initially uses a client/server paradigm to start communications. Once the connection is established, a symmetric view is taken. In addition, initially the interface is synchronous; however once the connection is established, an asynchronous interface may be selected.

All of the routines implementing transport services are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

1.1 Addresses

Addresses at the transport service access point are represented by the `TSAPaddr` structure.

```
struct TSAPaddr {
#define NTADDR  4
    struct NSAPaddr ta_addrs[NTADDR];
    int    ta_naddr;

#define TSSIZE  64
    int    ta_selectlen;
    char   ta_selector[TSSIZE];
};
#define NULLTA  ((struct TSAPaddr *) 0)
```

This structure contains two elements:

`ta_addrs`/`ta_nadr`: a list of network addresses, as described in the Section 1.6.1 on page 17;

```
#include <isode/tsap.h>
#include <isode/isoservent.h>

...

register struct TSAPaddr *ta;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("session", "tsap")) == NULL)
  error ("tsap/session");

/* RemoteHost is the host we're interested in,
  e.g., "gremlin.nrtc.northrop.com" */

if ((ta = is2taddr (RemoteHost, NULLCP, is)) == NULLTA)
  error ("address translation failed");

...
```

Figure 1.1: Constructing the TSAP address for the Session provider

`ta_selector`/`ta_selectlen`: the transport selector.

In Figure 1.1, an example of how one constructs the TSAP address for the session provider on host `RemoteHost` is presented. The routine `is2taddr` takes a host and service, and then consults the *isoentities*(5) file described in *ISODE Volume 10: Programmer's Guide: ASEs* to construct a transport address.

```
struct TSAPaddr *is2taddr (host, service, is)
char   *host,
       *service;
struct isoservent *is;
```

1.1.1 Calling Address

Certain users of the transport service might need to know the name of the local host when they initiate a connection. The routine `TLocalHostName` has been provided for this reason.

```
char   *TLocalHostName ()
```

1.1.2 Address Encodings

It may be useful to encode a transport address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `taddr2str` and `str2taddr` may be used in the interim.

```
char   *taddr2str (ta)
struct TSAPaddr *ta;
```

The parameter to this procedure is:

`ta`: the transport address.

If `taddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2taddr` takes an ascii string encoding and returns a transport address.

```
struct TSAPaddr *str2taddr (str)
char   *str;
```

The parameter to this procedure is:

`str`: the ascii string.

If `str2taddr` fails, it returns the manifest constant `NULLTA`.

Once a connection has been established, the routine `TGetAddresses` may be called to retrieve to the associated addresses. (Normally this information is presented to the application during connection establishment.)

```
struct TSAPaddr *TGetAddresses (sd, initiating,
        responding, td)
int     sd;
struct TSAPaddr *initiating,
                *responding;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`initiating`: the TSAP address of the initiator (to be filled-in);

`responding`: the TSAP address of the responder (to be filled-in); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, the `initiating` and `responding` parameters will be updated accordingly. Otherwise, the parameter `td` contains the reason for the failure.

1.2 Connection Establishment

Until the connection has been fully established, the implementation distinguishes between clients and servers, which are more properly referred to as *initiators* and *responders*, to use the OSI terminology.

1.2.1 Connection Negotiation

From the user's perspective, there are two parameters which are negotiated by the transport providers during connection establishment: *expedited data transfer*, and the *maximum size* of transport service data units.

Expedited Data

If the transfer of expedited data is negotiated, then small amounts of data may be sent out-of-band once the connection has been established. The size of the largest discrete unit to be sent is `TX_SIZE` (which is non-negotiable). This parameter is negotiated downward; that is, both the initiator and responder must agree to the use of expedited data.

Maximum TSDU Size

The transport provider will accept arbitrarily large transport service data units (TSDUs) and transparently fragment and re-assemble them during transit. Hence, the actual TSDU is of unlimited size. However, for efficiency reasons, it may be desirable for the user to send TSDUs which are no larger than a certain threshold. When a connection has been established, the service providers inform the initiator and responder as to what this threshold is.

> **NOTE:** In the current implementation, TSDUs which are no larger than the maximum atomic TSDU size are handled very efficiently. For optimal performance, users of the transport service should strive to avoid sending TSDUs which are larger than this threshold.

1.2.2 Server Initialization

The *tsapd*(8c) daemon, upon accepting a connection from an initiating host, consults the ISO services database to determine which program on the system implements the desired TSAP entity. For efficiency reasons, the *tsapd* program contains the bootstrap for several providers (e.g., session).

Once the program has been ascertained, the daemon runs the program with any arguments listed in the database. In addition, it appends some *magic arguments* to the argument vector. Hence, the very first action performed by the responder is to re-capture the TSAP state contained in these magic arguments. This is done by calling the routine `TInit`, which on a successful return, is equivalent to a T-CONNECT.INDICATION event from the transport service provider.

```
int     TInit (vecp, vec, ts, td)
int     vecp;
char  **vec;
struct TSAPstart *ts;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`ts`: a pointer to a `TSAPstart` structure, which is updated only if the call succeeds; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If `TInit` is successful, it returns information in the `ts` parameter, which is a pointer to a `TSAPstart` structure.

```
struct TSAPstart {
    int     ts_sd;

    struct TSAPaddr ts_calling;
    struct TSAPaddr ts_called;

    int     ts_expedited;

    int     ts_tsdusize;

    struct QOStype ts_qos;

#define TS_SIZE         32
    int     ts_cc;
    char    ts_data[TS_SIZE];
};
```

The elements of this structure are:

`ts_sd`: the transport-descriptor to be used to reference this connection;

`ts_calling`: the address of peer initiating the connection;

`ts_called`: the address of the peer being asked to respond;

`ts_expedited`: whether initiator requests the use of expedited data;

`ts_tsdusize`: the largest atomic TSDU size that can be used on the connection (see the note on page 4);

`ts_qos`: the quality of service on the connection (see Section 1.6.2); and,

`ts_data`/`ts_cc`: any initial data (and the length of that data).

If the call to `TInit` is not successful, then the user is informed that a T-DISCONNECT.INDICATION event has occurred, and the relevant information is returned in a `TSAPdisconnect` structure.

```
struct TSAPdisconnect {
    int     td_reason;

#define TD_SIZE         64
    int     td_cc;
    char    td_data[TD_SIZE];
};
```

The elements of this structure are:

`td_reason`: reason for disconnect (codes are listed in Table 1.1); and,

`td_data`/`td_cc`: any disconnect data (and the length of that data) from the peer.

After examining the information returned by `TInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the connection. If accepting, the `TConnResponse` routine is called (which corresponds to the T-CONNECT.RESPONSE action).

Provider-initiated Disconnect (fatal)

DR_NORMAL	Normal disconnect by session entity
DR_REMOTE	Remote transport entity congested at connect request time
DR_CONNECT	Connection negotiation failed
DR_DUPLICATE	Duplicate source reference detected for the same pair of NSAPs
DR_MISMATCH	Mismatched references
DR_PROTOCOL	Protocol error
DR_OVERFLOW	Reference overflow
DR_REFUSED	Connect request refused on this network connection
DR_LENGTH	Header or parameter length invalid
DR_NETWORK	Network disconnect

User-initiated Disconnect (fatal)

DR_UNKNOWN	Reason not specifed
DR_CONGEST	Congestion at TSAP
DR_SESSION	Session entity not attached to TSAP
DR_ADDRESS	Address unknown

Interface Errors (non-fatal)

DR_PARAMETER	Invalid parameter
DR_OPERATION	Invalid operation
DR_TIMER	Timer expired
DR_WAITING	Indications waiting

Table 1.1: TSAP Failure Codes

```
int     TConnResponse (sd, responding, expedited, data, cc,
                       qos, td)
int     sd;
struct TSAPaddr *responding;
int     expedited,
        cc;
char   *data;
struct QOStype *qos;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`responding`: the TSAP address of the responder (defaulting to the called address, if not present);

`expedited`: whether expedited data is to be permitted (this value must be `0` unless the T-CONNECT.INDICATION event indicated a willingness on the part of the initiator to support expedited data transfer);

data/cc: any initial data (and the length of that data, which may not exceed TC_SIZE octets);

qos: the quality of service on the connection (see Section 1.6.2); and,

td: a pointer to a TSAPdisconnect structure, which is updated only if the call fails.

If the call to TConnResponse is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails and the reason is not an interface error (see Table 1.1 on page 6), then the connection is closed.

If instead, the responder wishes to reject the connection, it should fire the T-DISCONNECT.REQUEST action by calling the TDiscRequest routine.

```
int      TDiscRequest (sd, data, cc, td)
int      sd;
char    *data;
int      cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

sd: the transport-descriptor;

data/cc: any disconnect data (and the length of that data, which may not exceed TD_SIZE octets); and,

td: a pointer to a TSAPdisconnect structure, which is updated only if the call fails.

After a return from this call, the responder may exit.

1.2.3 Client Initialization

A program wishing to connect to another user of transport services calls the TConnRequest routine, which corresponds to the T-CONNECT.REQUEST action.

```
int      TConnRequest (calling, called, expedited, data, cc,
                qos, tc, td)
struct TSAPaddr *calling,
                *called;
int      expedited,
         cc;
char    *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

calling: the TSAP address of the initiator; (need not be present);

called: the TSAP address of the responder;

expedited: whether expedited data is to be permitted;

`data`/`cc`: any initial data (and the length of that data, which may not exceed `TS_SIZE` octets);

`qos`: the quality of service on the connection (see Section 1.6.2);

`tc`: a pointer to a `TSAPconnect` structure, which is updated only if the call succeeds; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call to `TConnRequest` is successful (a successful return corresponds to a T-CONNECT.CONFIRMATION event), then information is returned in the `tc` parameter, which is a pointer to a `TSAPconnect` structure.

```
struct TSAPconnect {
    int     tc_sd;

    struct TSAPaddr tc_responding;

    int     tc_expedited;

    int     tc_tsdusize;

    struct QOStype tc_qos;

#define TC_SIZE         32
    int     tc_cc;
    char    tc_data[TC_SIZE];
};
```

The elements of this structure are:

`tc_sd`: the transport-descriptor to be used to reference this connection;

`tc_responding`: the responding peer's address (which is the same as the `called` address given as a parameter to `TConnRequest`);

`tc_expedited`: whether expedited data will be supported;

`tc_tsdusize`: the largest atomic TSDU size that can be used on the connection (see the note on page 4);

`tc_qos`: the quality of service on the connection (see Section 1.6.2); and,

`tc_data`/`tc_cc`: any initial data (and the length of that data).

If the call to `TConnRequest` is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails then the connection is not established, and the `TSAPdisconnect` structure has been updated.

Asynchronous Connections

Normally `TConnRequest` returns only after a connection has succeeded or failed. This is termed a *synchronous* connection initiation. If the user desires, an *asynchronous* connection may be initiated. The routine `TConnRequest` is really a macro which calls the routine `TAsynConnRequest` with an argument indicating that a connection should be attempted synchronously.

```
int     TAsynConnRequest (calling, called, expedited,
                data, cc, qos, tc, td, async)
struct TSAPaddr *calling,
                *called;
int     expedited,
        cc,
        async;
char   *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The additional parameter to this procedure is:

`async`: whether the connection should be initiated asynchronously.

If the `async` parameter is non-zero, then `TAsynConnRequest` returns one of four values: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection request succeeded; or, either of `CONNECTING_1` or `CONNECTING_2`, which indicates that the connection request is still in progress. In the first two cases, the usual procedures for handling return values from `TConnRequest` are employed (i.e., a `NOTOK` return from `TAsynConnRequest` is equivalent to a `NOTOK` return from `TConnRequest`, and, a `DONE` return from `TAsynConnRequest` is equivalent to a `OK` return from `TConnRequest`). In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `tc_sd` element of the `tc` parameter has been updated; it reflects the transport-descriptor to be used to reference this connection. Note that the `data` parameter is still being referenced by routines within *libisode*(3n) and should not be tampered with until the connection attempt has been completed.

To determine when the connection attempt has been completed, the routine `xselect` (consult *ISODE Volume 10: Programmer's Guide: ASEs*) should be used after calling `TSelectMask`. In order to determine if the connection attempt was successful, the routine `TAsynRetryRequest` is called:

```
int     TAsynRetryRequest (sd, tc, td)
int     sd;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`tc`: a pointer to a `TSAPconnect` structure, which is updated only if the call succeeds; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

Again, one of four values are returned: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection request succeeded; or, either of `CONNECTING_1` or `CONNECTING_2` which indicates that the connection request is still in progress.

In order to make efficient use of the asychronous connection facility, it is necessary to understand a bit of its underlying mechanisms. From a temporal perspective, connection establishment consists of two parts:

1. establishing a reliable end-to-end connection; and,

2. exchanging connection establishment information.

In some cases, the underlying transport mechanisms accomplish both simultaneously (when the end-to-end connection is built, connection establishment information is also exchanged).

Thus, in order to to perform asynchronous connections effectively, use of `TAsynConnRequest` and `TAsynRetry` should reflect this two-step process:

1. Call `TAsynConnRequest` with the `async` parameter taking the value 1. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment process has begun. Remember this value.

2. At some point in the future, call `TSelectMask` to get an argument for `xselect`. Then call `xselect` checking to see if writing is permitted. Then call `xselect` checking for writing if the remembered value was `CONNECTING_1` and for reading if it was `CONNECTING_2`. If either call returns `NOTOK`, then a catastrophic error has occurred.

   Repeat this step as often as necessary until `xselect` says that reading or writing as required is permitted.

3. Call `TAsynRetry`. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment is *still* in progress. Remember this value and go back to the previous step.

Although this seems complicated, implementation of these rules is actually straight-forward. In most cases, your code will do some work unrelated to the connection

Note that this procedure is equally applicable to the higher-layers (session, presentation, and association control) which also provide asynchronous connection facilities. For example, at the application layer, the routines `AcAsynAssocRequest` and `AcAsynRetryRequest` would be used.

1.3 Data Transfer

Once the connection has been established, a transport-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libisode*(3n) library. Further, the last parameter is usually a pointer to a `TSAPdisconnect` structure (as described on page 5). If a call to one of these routines fails, then the structure is updated. Otherwise, if the value of the `td_reason` element is associated with a fatal error, then the connection is closed. That is, a T-DISCONNECT.INDICATION event has occurred. The `DR_FATAL` macro can be used to determine this.

```
int     DR_FATAL (r)
int     r;
```

For protocol purists, the `DR_OFFICIAL` macro can be used to determine if the error is an “official” error as defined by the specification, or an “unofficial” error used by the implementation.

```
int     DR_OFFICIAL (r)
int     r;
```

1.3.1 Sending Data

There are three routines that may be used to send data. A call to the TDataRequest routine is equivalent to a T-DATA.REQUEST action on the part of the user.

```
int     TDataRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

sd: the transport-descriptor;

data/cc: the data to be written (and the length of that data); and,

td: a pointer to a TSAPdisconnect structure, which is updated only if the call fails.

If the call to TDataRequest is successful, then the data has been queued for sending. Otherwise, the td parameter indicates the reason for failure.

The TWriteRequest routine is similar in nature to the TDataRequest routine, but uses a different set of parameters. The invocation is:

```
int     TWriteRequest (sd, uv, td)
int     sd;
struct udvec *uv;
struct TSAPdisconnect  *td;
```

While the parameters are:

sd: the transport-descriptor;

uv: the data to be written, described in a null-terminated array of scatter/gather elements:

```
struct udvec {
    caddr_t uv_base;
    int     uv_len;
};
```

The elements of the structure are:

uv_base: the base of an element; and,

uv_cc: the length of an element.

and,

td: a pointer to a TSAPdisconnect structure, which is updated only if the call fails.

If the call to TWriteRequest is successful, then the data has been queued for sending. Otherwise, the td parameter indicates the reason for failure.

The TWriteRequest2 routine is similar to the TWriteRequest routine, but allows the caller to write a partial TSDU. The invocation is:

```
int     TWriteRequest2 (sd, uv, td, end)
int     sd;
struct udvec *uv;
struct TSAPdisconnect  *td;
int end;
```

The additional parameter is:

`end`: if this is non-zero then this call contains all (or the remaining) data of the TSDU. If this parameter is 0 then further call(s) will be made to complete this TSDU.

A call to the `TExpdRequest` routine is equivalent to a T-EXPEDITED-DATA.REQUEST action on the part of the user.

```
int     TExpdRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`data`/`cc`: the data to be written (and the length of that data, which may not exceed `TX_SIZE` octets); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call to `TExpdRequest` is successful, then the data has been queued for expedited sending. Otherwise, the `td` parameter indicates the reason for failure.

1.3.2 Receiving Data

There is one routine that is used to read data, `TReadRequest`, a call to which is equivalent to waiting for a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event.

```
int     TReadRequest (sd, tx, secs, td)
int     sd;
struct TSAPdata *tx;
int     secs;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`tx`: a pointer to the `TSAPdata` structure to be given the data;

`secs`: the maximum number of seconds to wait for the data (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and, `td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call to `TReadRequest` is successful, then the data has been read into the `tx` parameter.

```
struct TSAPdata {
    int     tx_expedited;

    int     tx_cc;
    struct qbuf tx_qbuf;
};
```

The elements of a `TSAPdata` structure are:

`tx_expedited`: whether the data was received via expedited transfer (i.e., an T-EXPEDITED-DATA.INDICATION event occurred);

`tx_cc`: the total number of octets that was read; and,

`tx_qbuf`: the data that was read, in a buffer-queue form.

```
struct qbuf {
    struct qbuf *qb_forw;
    struct qbuf *qb_back;

    int    qb_len;
    char  *qb_data;

    char   qb_base[1];
};
```

The elements of a `qbuf` structure are:

`qb_forw`/`qb_back`: forward and back pointers;

`qb_data`/`qb_len`: the user data (and the length of that data); and,

`qb_base`: the extensible array containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `TXFREE` macro when no longer referenced. The `TXFREE` macro, which is used for this purpose, behaves as if it was defined as:

```
void    TXFREE (tx)
struct TSAPdata *tx;
```

The macro frees only the data allocated by `TDataRequest`, and not the `TSAPdata` structure itself. Further, `TXFREE` should be called only if the call to the `TDataRequest` routine returned `OK`.

NOTE: Because the `TSAPdata` structure contains a `qbuf` element, care must be taken in initializing and copying variables of this type. The routines in *libisode*(3n) library will correctly initialize these structures when given as parameters. But, users who otherwise manipulate `TSAPdata` structures should take great care.

Otherwise if the call to `TReadRequest` did not succeed, the `td` parameter indicates the reason for failure.

1.3.3 Asynchronous Event Handling

The data transfer events discussed thus far have been synchronous in nature. Some users of the transport service may wish an asynchronous interface. The `TSetIndications` routine is used to change the service associated with a transport-descriptor to or from an asynchronous interface.

```
int     TSetIndications (sd, data, disc, td)
int     sd;
int    (*data) (),
       (*disc) ();
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`data`: the address of an event-handler routine to be invoked when data has arrived (either a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event occurs);

`disc`: the address of an event-handler routine to be invoked when the connection has been closed (a T-DISCONNECT.INDICATION event occurs); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the service is to be made asynchronous, then both `data` and `disc` are specified; otherwise, if the service is to be made synchronous, neither should be specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked.

When an event associated with data arriving occurs, the event-handler routine is invoked with two parameters:

```
(*data) (sd, tx);
int     sd;
struct TSAPdata *tx;
```

The parameters are:

`sd`: the transport-descriptor; and,

`tx`: a pointer to a `TSAPdata` structure containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the `TXFREE` macro (described on page 13) when no longer needed.

Similarly, when an event associated with connection release occurs, the event-handler is also invoked with two parameters:

```
(*disc) (sd, td);
int     sd;
struct TSAPdisconnect *td;
```

The parameters are

`sd`: the transport-descriptor; and,

`td`: a pointer to a `TSAPdisconnect` structure indicating why the connection was released.

Note that the transport-descriptor is no longer valid at the instant the call is made.

NOTE:	The *libisode*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous transport-descriptors should NOT use SIGEMT for other purposes.

1.3.4 Synchronous Event Multiplexing

A user of the transport service may wish to manage multiple transport-descriptors simultaneously; the routine `TSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect`.

```
int     TSelectMask (sd, mask, nfds, td)
int     sd;
fd_set *mask,
int    *nfds;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`mask`: a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds`: a pointer to an integer-valued location meaningful to `xselect`; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the transport-descriptor is ready for reading, `TReadRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `TReadRequest` will return `NOTOK` with error code `DR_TIMER`.

In addition, the routine `TSelectOctets` is provided to return an estimate of how many octets might be returned by the next call to `TReadRequest`:

```
int     TSelectOctets (sd, nbytes, td)
int     sd;
long   *nbytes;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`nbytes`: a pointer to a longword location that, on success, will be updated to contain the number of octets that might be returned; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

1.4 Connection Release

The `TDiscRequest` routine is used to release a connection. Note, that the TSAP makes no guarantee that any queued data will be received before the connection closes.

```
int      TDiscRequest (sd, data, cc, td)
int      sd;
char    *data;
int      cc;
struct TSAPdisconnect  *td;
```

The parameters to the procedure are described on page 7.

1.5 State Saving and Restoration

Some users of the transport service, and in particular the session provider, require the ability to *execve*(2) another process image and have that process use the transport connection. Since the *libisode*(3n) library is not kernel-resident, special provisions are necessary to support this behaviour.

1.5.1 Saving the State

The routine `TSaveState` is used to record the state of the TSAP for a given transport-descriptor.

```
int      TSaveState (sd, vec, td)
int      sd;
char   **vec;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`vec`: a pointer to the first free slot in the argument vector for *execve*(2); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call succeeds, then an extra *magic argument* has been placed in the argument vector. The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

NOTE:	Once a successful call to `TSaveState` is made on a transport descriptor, that descriptor may no longer be referenced until a corresponding call to `TRestoreState` is successful.

1.5.2 Restoring the State

The routine `TRestoreState` is used to re-initialize the state of the TSAP.

```
int     TRestoreState (buffer, ts, td)
char   *buffer;
struct TSAPstart  *ts;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`buffer`: the *magic argument* constructed by `TSaveState`;

`ts`: a pointer to a `TSAPstart` structure, which is updated only if the call succeeds; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If `TRestoreState` is successful, it returns information in a `TSAPstart` structure, as defined on page 5. There is one exception however: in the current implementation the `ts_cc` and `ts_data` elements are undefined on a successful return from `TRestoreState`.

1.6 Broad Scope Parameters

There are two *cookie* parameters where both requirements and implementations can vary greatly: network addresses and quality of service.

1.6.1 Network Addresses

Network addresses can be of very different kinds. In this software distribution, a "unified" format has been adopted in the `NSAPaddr` structure:

```
struct NSAPaddr {
    long    na_stack;
#define NA_NSAP 0
#define NA_TCP  1
#define NA_X25  2

    long    na_community;

    union {
        struct na_nsap {
#define NASIZE  64
            char    na_nsap_address[NASIZE];
            char    na_nsap_addrlen;
        }               un_na_nsap;

        struct na_tcp {
#define NSAP_DOMAINLEN  63
            char    na_tcp_domain[NSAP_DOMAINLEN + 1];
            u_short na_tcp_port;
```

```
                u_short na_tcp_tset;
#define NA_TSET_TCP     0x0001
#define NA_TSET_UDP     0x0002
        }                un_na_tcp;

        struct na_x25 {
#define NSAP_DTELEN      15
            char    na_x25_dte[NSAP_DTELEN + 1];
            char    na_x25_dtelen;

#define NPSIZE  4
            char    na_x25_pid[NPSIZE];
            char    na_x25_pidlen;

#define CUDFSIZE 16
            char    na_x25_cudf[CUDFSIZE];
            char    na_x25_cudflen;

#define FACSIZE 6
            char    na_x25_fac[FACSIZE];
            char    na_x25_faclen;
        }                un_na_x25;
    }               na_un;

#define na_address      na_un.un_na_nsap.na_nsap_address
#define na_addrlen      na_un.un_na_nsap.na_nsap_addrlen

#define na_domain       na_un.un_na_tcp.na_tcp_domain
#define na_port         na_un.un_na_tcp.na_tcp_port

#define na_dte          na_un.un_na_x25.na_x25_dte
#define na_dtelen       na_un.un_na_x25.na_x25_dtelen
#define na_pid          na_un.un_na_x25.na_x25_pid
#define na_pidlen       na_un.un_na_x25.na_x25_pidlen
#define na_cudf         na_un.un_na_x25.na_x25_cudf
#define na_cudflen      na_un.un_na_x25.na_x25_cudflen
#define na_fac          na_un.un_na_x25.na_x25_fac
#define na_faclen       na_un.un_na_x25.na_x25_faclen
};
#define NULLNA  ((struct NSAPaddr *) 0)
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Based on the value of the tag (`na_stack`), a different structure is selected.

For a native OSI CO-mode transport service, the value of the tag is `NA_NSAP`, and the following elements are meaningful:

`na_address`/`na_addrlen`: the network address (and its length), binary-valued.

For emulation of the OSI transport service on top of the TCP, the value of the tag is `NA_TCP`, and the following elements are meaningful:

`na_domain`: the null-terminated domain name (e.g., "gonzo.twg.com") or dotted-quad (e.g., "128.99.0.17");

`na_port`: the TCP-port number offering the service (if zero, the service on port 102 is used); and,

`na_tset`: the set of IP-based transport services available at the address (if zero, the TCP service is used).

For use of a single-subnet X.25, the value of the tag is `NA_X25`, and the following elements are meaningful:

`na_dte`/`na_dtelen`: the X.121 address (and its length), ascii-valued, possibly null-terminated;

`na_pid`/`na_pidlen`: the protocol id (and its length), binary-valued;

`na_cudf`/`na_cudflen`: the call user data (and its length), binary-valued; and,

`na_fac`/`na_faclen`: the negotiated facilities proposed in the call request packet (and its length), binary-valued.

If the value of the tag is not `NA_NSAP`, it may be useful to normalize the address into a "real" OSI address. The routine `na2norm` is used:

```
char   *na2norm (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be normalized.

A new network address is returned from a static area which contains the normalized form.

The routine `na2str` takes a network address and returns a null-terminated ascii string suitable for viewing:

```
char   *na2str (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be printed.

The `na_community` field is an internal number used to distinguish between different OSI communities. Consult *ISODE Volume 3: Administrator's Guide: General Services* for further details.

1.6.2 Quality of Service

Currently, quality of service is largely uninterpreted by the software. However, the quality of service structure contains those parameters which are supported:

```
struct QOStype {
                              /* transport QOS */
    int    qos_reliability;
#define HIGH_QUALITY   0
#define LOW_QUALITY    1
```

```
                                /* session QOS */
     int    qos_sversion;
     int    qos_extended;
     int    qos_maxtime;
};
#define NULLQOS ((struct QOStype *) 0)
```

The elements of this structure are:

`qos_reliability`: the “reliability” level of the connection, either high- or low-quality;

`qos_sversion`: the session version requested/negotiated on this connection (only applicable above the transport layer, obviously), if the manifest constant `NOTOK` is used when initiating a connection, this indicates that the highest possible version should be negotiated;

`qos_extended`: the extended control parameter for the connection, (if non-zero, extended control is used by the session layer); and,

`qos_maxtime`: after a transport connection is established, the maximum number of seconds to wait for an acknowledgement from the responding SPM (any non-positive number indicates that no timelimit is desired).

1.7 Listen Facility

The transport service routines in the *libisode*(3n) library, support a facility which permits a process to *listen* for certain connections. This can be useful for implementing an application which requires that a single server process handle multiple clients, or for connection recovery. These routines return the manifest constant `NOTOK` on error, and `OK` on success, they also update the `td` parameter given to the routine. This parameter is a pointer to a `TSAPdisconnect` structure.

Although this facility is described in terms of the *libisode*(3n) library, it will function at any other higher-layer in the system (e.g., the listen facility can be used for session or application-entities).

A program starts listening for an particular connection by calling the routine `TNetListen`.

```
int     TNetListen (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: the transport address (0 or more network addresses) to listen on; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, then the program is now listening for incoming connections on that network address. Otherwise, the `td` parameter indicates the reason for failure.

A variant of `TNetListen` is the `TNetUnique` routine, which starts the process listening on a set of unique network (sub)addresses.

```
int     TNetUnique (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: a transport address containing one or more partially filled-in network addresses; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, each network address in the `ta` parameter is fully filled-in, the program is now listening for incoming connections on the resulting transport. Otherwise, the `td` parameter indicates the reason for failure.

To check when a new connection is waiting, or when existing connections have activity on them, the routine `TNetAccept` is used.

```
int     TNetAccept (vecp, vec, nfds, rfds, wfds, efds, secs,
                td)
int    *vecp,
char  **vec;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`vecp`/`vec`: the initialization vector for the new connection.

`nfds`/`rfds`/`wfds`/`efds`: connection-descriptors for use with `xselect`;

`secs`: the maximum number of seconds to wait for activity (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call to `TNetAccept` succeeds then the value of `vecp` should be checked. If `vecp` is greater than zero, a new connection has been accepted, and `TInit` should be called, presumably followed by `TConnResponse`.[1] Regardless of the value of `vecp`, the value of `rfds` and `wfds` should be checked to see which connections have activity pending. For these connections, any reads should probably be done with a `secs` argument indicating a polling operation (i.e., a value of `OK`). Otherwise, if the call to `TNetAccept` fails, then the `td` parameter indicates the reason for failure.

NOTE:	The `TNetAccept` procedure when first called arranges to clean up dead child processes. If the program will run any sub-processes and check their exit status, the automatic collection of zombie process should be disabled, by first calling `TNetAccept` with a timeout of `OK` and then setting the child signal handler to it's default state.

[1] Actually, any service addressable via a transport selector can use this service, e.g., if appropriate, a call to `AcInit`, followed by a call to `AcAssocResponse`, can be made.

The routine `TNetAccept` is actually a macro which invokes a routine called `TNetAcceptAux`:

```
int     TNetAcceptAux (vecp, vec, newfd, ta, nfds, rfds, wfds,
                efds, secs, td)
int    *vecp,
char  **vec;
int     newfd;
struct TSAPaddr *ta;
int     nfds;
fd_set *rfds,
       *wfds,
       *efds;
int     secs;
struct TSAPdisconnect *td;
```

The additional parameters to this procedure are:

`newfd`: a pointer to an integer which will be given the value of the connection-descriptor associated with the new connection; and,

`ta`: a pointer to a `TSAPaddr` structure which will be given the value of the transport address receiving the new connection (the called or listening address).

Prior to exiting, the user should call `TNetClose` to stop listening for connections.

```
int     TNetClose (ta, td)
struct TSAPaddr *ta;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`ta`: the transport address to stop listening on (use the manifest constant `NULLTA` to stop listening on all addresses); and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is successful, then the program has stopped listening for incoming connections on that network address. Otherwise, the `td` parameter indicates the reason for failure.

1.8 Queued (non-blocking) Writes Facility

All "read" operations in the ISODE are inherently non-blocking. Historically, "write" operations have not required this capability. However, some applications (e.g., the QUIPU DSA) require non-blocking writes. The routine `TSetQueuesOK` is used to enable or disable this facility:

```
int     TSetQueuesOK (sd, onoff, td)
int     sd;
int     onoff;
struct TSAPdisconnect *td;
```

The parameters to this procedure are:

`sd`: the transport-descriptor;

`onoff`: a flag indicating whether the facility is to be enabled (non-zero value) or disabled (zero value) for the transport-descriptor; and,

`td`: a pointer to a `TSAPdisconnect` structure, which is updated only if the call fails.

If the call is not successful, the `td` parameter indicates the reason for failure. Otherwise, if the `onoff` parameter has a non-zero value, then any "write" operations which ultimately map to this transport-descriptor will not block the process. This is done by maintaining a queue of write operations and periodically retrying them. In order to schedule retries, the routine `TNetAccept` described earlier should be called frequently. In addition, if some failure occurs during the retry (e.g., the transport connection is disconnected), then `TNetAccept` will mark that descriptor as being ready for reading. When the program interrogates the descriptor, the appropriate error code will be returned. Note that in order for this action to occur, any descriptor which has queued writes enabled must be included in the `rfs` parameter supplied when the `TNetAccept` routine is called.

1.9 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `td` parameter given to the routine. The `td_reason` element of the `TSAPdisconnect` structure can be given as an parameter to the routine `TErrString` which returns a null-terminated diagnostic string.

```
char   *TErrString (c)
int     c;
```

1.10 Compiling and Loading

Programs using the transport service routines in the *libisode*(3n) library should include `<isode/tsap.h>`. These programs should also be loaded with `-ltsap` and, for reasons explained momentarily, `-licompat`.

1.11 An Example

Let's consider how one might construct a loopback entity that resides on the TSAP. This entity will use a synchronous interface.

First, we must decide at what address the entity will reside. For simplicity's sake, we'll say that the location is `tsap/echo`, as defined in the *isoservices*(5) database.

Next, we actually code the loopback entity. There are two parts to the program: initialization and data transfer; release will occur whenever data transfer fails. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 1.2, the initialization steps for the loopback entity, including the outer *C* wrapper, is shown. The entity does not examine any of its arguments, but could do so after the call to `TInit`. After examining the arguments, it could decide to reject the connection attempt, by using `TDiscRequest`.

Instead, it uses `TConnResponse` with the exact negotiated parameters with which it was supplied. Hence, if the initiator wanted to use expedited data transfer, the loopback entity responding to the connection would honor that.

In Figure 1.3 on page 27, the data transfer loop is realized. The loopback entity awaits an event from the service provider, which either indicates that data has arrived or that the connection has been closed. If a disconnection occurred (a T-DISCONNECT.INDICATION event is reported), then the reason is checked. If the event did not occur because the initiator performed a T-DISCONNECT.REQUEST indication, then an error is signaled. Otherwise, the inner-loop is terminated and the process will gracefully terminate. If instead data arrived, it is echoed back to the initiator.

```
#include <stdio.h>
#include <isode/tsap.h>
#include <isode/isoservent.h>


/* ARGSUSED */

main (argc, argv, envp)
int     argc;
char  **argv,
      **envp;
{
    int     result,
            sd;
    struct TSAPstart    tss;
    register struct TSAPstart  *ts = &tss;
    struct TSAPdata txs;
    register struct TSAPdata  *tx = &txs;
    struct TSAPdisconnect   tds;
    register struct TSAPdisconnect *td = &tds;

    if (TInit (argc, argv, ts, td) == NOTOK)
        error ("T-CONNECT.INDICATION: %s", TErrString (td -> td_reason));
    sd = ts -> ts_sd;

/* examine argv here, if need be */

    if (TConnResponse (sd, &ts -> ts_called, ts -> ts_expedited,
                ts -> ts_data, ts -> ts_len, NULLQOS, td) == NOTOK)
        error ("T-CONNECT.RESPONSE: %s", TErrString (td -> td_reason));

...
```

Figure 1.2: Initializing the loopback entity

1.12 Compatibility Issues

The *libicompat*(3) library is used as an aid for porting the software from one system to another. This library contains generic service routines, which are in turn composed of the native facilities available on the target host. All of the higher layer `#include` files automatically reference parts of this library as appropriate. Hence, when loading the the portions of the software independently, the loader must be given the `-licompat` flag.

The problem of this approach, of course, is that not all facilities can be precisely emulated. To misquote M.A. Padlipsky[3]:

> *Sometimes when you try to make an apple look like an orange you get back something that smells like a lemon.*

1.13 For Further Reading

The ISO specification for transport services is defined in [2]. The corresponding CCITT recommendation is defined in [1]. The document describing how these services can be implemented on top of the TCP[4] is [5].

```
...

  for (;;) {
        if (TReadRequest (sd, tx, NOTOK, td) == NOTOK) {
          if (td -> td_reason != DR_NORMAL)
                error ("T-READ.REQUEST: %s", TErrString (td -> td_reason));
          break;
        }

        if (tx -> tx_expedited)
          result = TExpdRequest (sd, tx -> tx_base, tx -> tx_cc, td);
        else
          result = TDataRequest (sd, tx -> tx_base, tx -> tx_cc, td);
        if (tx -> tx_base)
          free (tx -> tx_base);

        if (result == NOTOK)
          error ("%s:  %s", tx -> tx_expedited ? "T-EXPEDITED-DATA.REQUEST"
                  : "T-DATA.REQUEST", TErrString (td -> td_reason));

  }

  exit (0);
}
```

Figure 1.3: Data Transfer for the loopback entity

Bibliography

[1] Transport Service Definition for Open Systems Interconnection (OSI) for CCITT Applications. International Telegraph and Telephone Consultative Committee, October 1984. Recommendation X.214.

[2] Information Processing Systems — Open Systems Interconnection — Transport Service Definition. International Organization for Standardization and International Electrotechnical Committee, 1986. International Standard 8072.

[3] Michael A. Padlipksy. Gateways, Architectures, and Heffalumps. In *The Elements of Networking Style and Other Essays and Animadversions on the Art of Intercomputer Networking*, chapter 10, pages 167–176. Prentice-hall, Englewood Cliffs, New Jersey, 1985. Also available as Internet Request for Comments 875.

[4] Jon B. Postel. Transmission Control Protocol. Request for Comments 793, DDN Network Information Center, SRI International, September 1981. See also MIL-STD 1778.

[5] Marshall T. Rose and Dwight E. Cass. ISO Transport Services on top of the TCP. Request for Comments 1006, DDN Network Information Center, SRI International, May 1987.