```
...

    if ((sz = is2saddr (argv[1], NULLCP, is)) == NULLSA)
            error ("address translation failed");
    (void) bzero ((char *) sf, sizeof *sf);

    settings = 0;
#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \                                  10
            settings |= ST_INIT_VALUE << shift; \
}
    dotokens ();
#undef dotoken

    if (SConnRequest (sf, NULLSA, sz, requirements, settings, SERIAL_NONE,
                NULLCP, 0, NULLQOS, sc, si) == NOTOK)
            error ("S-CONNECT.REQUEST: %s", SErrString (sa -> sa_reason));
    if (sc -> sc_result != SC_ACCEPT)
            error ("connection rejected by sink[%s]:  %s",          20
                        SErrString (sa -> sa_reason),
                        SErrString (sc -> sc_result));

    sd = sc -> sc_sd;
    requirements = sc -> sc_requirements;

#define dotoken(requires,shift,bit,type) \
{ \
    if (requirements & requires) \
            if ((sc -> sc_settings & (ST_MASK << shift)) == ST_INIT_VALUE)30\
                owned |= bit; \
}
            dotokens ();
#undef dotoken

...
```

Figure 3.3: Initializing the SSAP source entity (continued)

```
...

    while (fgets (buffer, sizeof buffer, stdin))
        if (SDataRequest (sd, buffer, strlen (buffer) + 1, si) == NOTOK)
            error ("S-DATA.REQUEST: %s", SErrString (sa -> sa_reason));

    if (SRelRequest (sd, NULLCP, 0, NOTOK, sr, si) == NOTOK)
        error ("S-RELEASE.REQUEST: %s", SErrString (sa -> sa_reason));

    if (!sr -> sr_affirmative) {                                        10
        (void) SUAbortRequest (sd, NULLCP, 0, si);
        error ("release rejected by sink");
    }

    exit (0);
}
```

Figure 3.4: Data Transfer for the SSAP source entity

.

3.10 For Further Reading

The ISO specification for session services is defined in [ISO87e], while the complementary CCITT recommendation is defined in [CCI84b]. The corresponding protocol definitions are [ISO87b] and [CCI84a], respectively.

Chapter 4

Transport Services

At the heart of this distribution is the *libtsap*(3n) library. This 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 in the *libtsap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

4.1 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 4.6.1 on page 125;

`ta_selector/ta_selectlen:` the transport selector.

In Figure 4.1, 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 Chapter 7 of *Volume One* to construct a transport address.

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

4.1.1 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 ()
```

```
#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 4.1: Constructing the TSAP address for the Session provider

4.1.2 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.

4.2 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.

4.2.1 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.

4.2.2 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 106);

`ts_qos`: the quality of service on the connection (see Section 4.6.2); 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 4.1); 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).

```
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);