```
...

    while (fgets (buffer, sizeof buffer, stdin)) {
        register PE = oct2prim (buffer, strlen (buffer) + 1);

        if (PDataRequest (sd, &pe, 1, pi) == NOTOK)
            error ("P-DATA.REQUEST: %s", PErrString (pa -> pa_reason));

        pe_free (pe);
    }

    if (PRelRequest (sd, NULLPEP, 0, NOTOK, pr, pi) == NOTOK)
        error ("P-RELEASE.REQUEST: %s", PErrString (pa -> pa_reason));

    if (!pr -> pr_affirmative) {
        (void) PUAbortRequest (sd, NULLPEP, 0, pi);
        error ("release rejected by sink");
    }
    PRFREE (pr);

    exit (0);
}
```

Figure 2.3: Data Transfer for the PSAP source entity

2.8 Lightweight Presentation Protocol

To run OSI applications using the lighweight presentation protocol defined in RFC1085, load the program with `-lpsap2-lpp` or `-isode-lpp`.

This is a complete implementation of RFC1085. The following functions are available:

```
PInit
PConnResponse
PAsynConnRequest
PAsynRetryRequest
PDataRequest
PReadRequest
```

```
PUAbortRequest
PRelRequest
PRelResponse
PSetIndications
PSelectMask
PErrSTring
```

Note that when RFC1085 is used as a presentation backing, only a subset of the presentation services are available:

P-CONNECT P-DATA P-U-ABORT P-P-ABORT

The *lppd*(8c) daemon is used to listen for incoming connections and dispatch the appropriate daemon. This daemon will listen only for connections using the TCP backing. For connections using the UDP backing, the responder program must listen itself. This is trivally accomplished using the `isodeserver` routine described in Section 2.5 on page 44.

2.9 For Further Reading

The ISO specification for session services is defined in [ISO88a], while the corresponding protocol definition is [ISO88b].

Chapter 3

Session Services

The *libssap*(3n) library implements the session service.

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 necessary 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 `ssap` 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 *libssap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

3.1 Warning

Please read the following important message.

> **NOTE:** Readers of this chapter should have an intimate understanding of the OSI session service. It is not the intent of this chapter to present a tutorial on these services, so novice users will suffer greatly if they choose to read further.

3.2 Addresses

Addresses at the session service access point are represented by the `SSAPaddr` structure.

```
struct SSAPaddr {
    struct TSAPaddr sa_addr;

#define SSSIZE  64
    int     sa_selectlen;
    char    sa_selector[SSSIZE];
};
#define NULLSA  ((struct SSAPaddr *) 0)
```

This structure contains two elements:

`sa_addr`: the transport address, as described in Section 4.1 on page 101; and,

`sa_selector/sa_selectlen`: the session selector.

In Figure 3.1, an example of how one constructs the SSAP address for the Presentation provider on host `RemoteHost` is presented. The routine `is2saddr` takes a host and service, and then consults the *isoentities*(5) file described in Chapter 7 of *Volume One* to construct a session address.

```
struct SSAPaddr *is2saddr (host, service, is)
char   *host,
       *service;
struct isoservent *is;
```

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

...

register struct SSAPaddr *sa;
register struct isoservent *is;

...

if ((is = getisoserventbyname ("presentation", "ssap")) == NULL)
    error ("ssap/presentation");

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

if ((sa = is2saddr (RemoteHost, NULLCP, is)) == NULLSA)
    error ("address translation failed");

...
```

Figure 3.1: Constructing the SSAP address for the Presentation provider

3.2.1 Calling Address

Certain users of the session service (e.g., the reliable transfer service) need to know the name of the local host when they initiate a connection. The routine `SLocalHostName` has been provided for this reason.

```
char   *SLocalHostName ()
```

3.2.2 Address Encodings

It may be useful to encode a session address for viewing. Although a consensus for a standard way of doing this has not yet been reached, the routines `saddr2str` and `str2saddr` may be used in the interim.

```
char   *saddr2str (sa)
struct SSAPaddr *sa;
```

The parameter to this procedure is:

`sa:` the session address.

If `saddr2str` fails, it returns the manifest constant `NULLCP`.

The routine `str2saddr` takes an ascii string encoding and returns a session address.

```
struct SSAPaddr *str2saddr (str)
char   *str;
```

The parameter to this procedure is:

`str:` the ascii string.

If `str2saddr` fails, it returns the manifest constant `NULLSA`.

3.3 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 OSI terminology.

3.3.1 Connection Negotiation

From the user's perspective, there are several parameters which are negotiated by the session providers during connection establishment. Suggestions as to the values of some of these parameters are made by the user.

Maximum SSDU Size

The session provider will accept arbitrarily large session service data units (SSDUs) and transparently fragment and re-assemble them during transit. Hence, the actual SSDU is of unlimited size. However, for efficiency reasons, it may be desirable for the user to send SSDUs 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, SSDUs which are no larger than the maximum atomic SSDU size are handled very efficiently. For optimal performance, users of the session service should strive to avoid sending SSDUs which are larger than this threshold.

Session Requirements

Users may specify the particular services that they will require from the session provider. The particular requirements supported in the current implementation are listed in Table 3.1. These requirements are always negotiated downward. That is, the initiator of the connection (i.e., the "client") indicates the desired session requirements. These are then given to the responder to the connection request (i.e., the "server") who may select any (or all) of the indicated session requirements.[1] This selection is then indicated to the initiator.

[1]There is one exception, the responder may not select both the half- and full-duplex requirements. It must choose one. If the initiator selects both initially, it is indicating that the choice is made at the responder's discretion.

Requirements	
SR_HALFDUPLEX	Half-duplex
SR_DUPLEX	Full-duplex
SR_EXPEDITED	Expedited Data Transfer
SR_MINORSYNC	Minor Synchronize
SR_MAJORSYNC	Major Synchronize
SR_RESYNC	Resynchronize
SR_ACTIVITY	Activity Management
SR_NEGOTIATED	Negotiated Release
SR_CAPABILITY	Capability Data Transfer
SR_EXCEPTIONS	Exception Reporting
SR_TYPEDATA	Typed Data Transfer
Subsets (combinations of the above)	
SR_BASUBSET	Basic Activity Subset
SR_BCSUBSET	Basic Combined Subset
SR_BSSUBSET	Basic Synchronized Subset

Table 3.1: Session Requirements

Token	Name	Availability
ST_RLS_TOKEN	release token	SR_RLS_EXISTS
ST_MAJ_TOKEN	majorsync token	SR_MAJ_EXISTS
ST_ACT_TOKEN	activity token (really majorsync token)	SR_ACT_EXISTS
ST_MIN_TOKEN	minorsync token	SR_MIN_EXISTS
ST_DAT_TOKEN	data token	SR_DAT_EXISTS

Table 3.2: Session Tokens

Session Tokens

Depending on the session requirements selected, several session tokens may be available in order to coordinate the use of session services.

There are two terms commonly used when referring to a session token:

- Availability
  If a token is available, then it exists for use during the session connection. The availability of a token depends on the session requirements selected for the connection.

- Ownership
  Certain session services are may not be requested by a user unless that user owns the token associated with those services.

The particular tokens supported in the current implementation, along with their associated availability information are listed in Table 3.2.

The session requirements are encoded as a single integer (actually, only the low-order 2 octets of an integer). To determine if a token is available, one can use a simple test involving the session requirements:

```
if (requirements & SR_xxx_EXISTS) {
    ...
}
```

For example, to determine if the negotiated release token is available:

```
if (requirements & SR_RLS_EXISTS) {
    ...
}
```

ST_INIT_VALUE	initiator's side
ST_RESP_VALUE	responder's side
ST_CALL_VALUE	responder's choice

Table 3.3: Initial Token Settings

Finally, the macro `dotokens` may be used to execute *C* code for each session token (regardless of availability). This macro acts as if it executes:

```
dotoken (SR_xxx_EXISTS, ST_xxx_SHIFT, ST_xxx_TOKEN, "xxx");
```

for each token. Usually, `dotoken` is a macro which executes some code for each token. An example is provided momentarily.

Initial Token Settings

For each token which is made available during connection negotiation, the choice as to which user initially has the token is left to the discretion of the initiator. The three choices for the initial settings of a token are listed in Table 3.3.

The initial settings for all available tokens are encoded in a single integer (actually, only the low-order 2 octets of an integer). To encode a value:

```
settings &= ~(ST_MASK << ST_yyy_SHIFT);
settings |= ST_xxx_VALUE << ST_yyy_SHIFT;
```

For example, to indicate that the responder of the connection is to initially have the data token:

```
settings &= ~(ST_MASK << ST_DAT_SHIFT);
settings |= ST_RESP_VALUE << ST_DAT_SHIFT;
```

The first statement, which clears the field in `settings` by using `ST_MASK`, is not required if `settings` is initially `0`.

If the initiator indicates that the initial setting of a token is left to the responder's choice, then the responder must decide. In Figure 3.2, an example of the `dotokens` macro is presented. In this example, the responder examines the initial setting for each available token, and: