likely reason for the call failing is `SC_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 arrival occurs, the event-handler routine is invoked with two parameters:

```
(*data) (sd, sx);
int     sd;
struct SSAPdata *sx;
```

The parameters are:

`sd:` the session-descriptor; and,

`sx:` a pointer to the `SSAPdata` structure containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the `SXFREE` macro (described on page 67) when no longer needed.

When an event associated with token management arrives the event-handler routine is invoked with two parameters:

```
(*tokens) (sd, st);
int       sd;
struct SSAPtoken *st;
```

The parameters are:

`sd:` the session-descriptor; and,

`st:` a pointer to the `SSAPtoken` structure containing the token management information.

When an event associated with synchronization management arrives the event-handler routine is invoked with two parameters:

```
(*sync) (sd, sn);
int       sd;
struct SSAPsync *sn;
```

The parameters are:

`sd:` the session-descriptor; and,

`sn:` a pointer to the `SSAPsync` structure containing the synchronization management information.

When an event associated with activity management arrives the event-handler routine is invoked with two parameters:

```
(*activity) (sd, sv);
int       sd;
struct SSAPactivity *sv;
```

The parameters are:

`sd:` the session-descriptor; and,

`sv:` a pointer to the `SSAPactivity` structure containing the activity management information.

When an event associated with exception reporting arrives the event-handler routine is invoked with two parameters:

```
(*report) (sd, sp);
int       sd;
struct SSAPreport *sp;
```

The parameters are:

`sd:` the session-descriptor; and,

`sp:` a pointer to the `SSAPreport` structure containing the exception report information.

When an event associated with connection termination arrives the event-handler routine is invoked with two parameters:

```
(*finish) (sd, sf);
int       sd;
struct SSAPfinish *sf;
```

The parameters are:

`sd:` the session-descriptor; and,

`sf:` a pointer to the `SSAPfinish` structure containing information concerning the request to terminate the connection.

When an event associated with a user- or provider-initiated abort occurs, the event-handler is invoked with two parameters:

```
(*abort) (sd, sa);
int      sd;
struct SSAPabort *sa;
```

The parameters are:

`sd:` the session-descriptor; and,

`sa:` a pointer to the `SSAPabort` structure indicating why the connection was aborted.

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

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

3.4.9 Synchronous Event Multiplexing

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

```
int     SSelectMask (sd, mask, nfds, si)
int     sd;
fd_set *mask,
int    *nfds;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

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

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

`si`: a pointer to a `SSAPindication` 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 `SC_WAITING`, which indicates that an event is waiting for the user.

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

3.5 Connection Release

The `SRelRequest` routine is used to request the release a connection, and corresponds to a S-RELEASE.REQUEST action. The SSAP attempts to gracefully drain any queued data before closing the connection.

```
int     SRelRequest (sd, data, cc, secs, sr, si)
int     sd;
char   *data;
int     cc;
int     secs;
struct SSAPrelease  *sr;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

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

`secs`: the maximum number of seconds to wait for a response (use the manifest constant `NOTOK` if no time-out is desired);

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

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

If the call to `SRelRequest` is successful, then this corresponds to a S-RELEASE.CONFIRMATION event, and it returns information in the `sr` parameter, which is a pointer to a `SSAPrelease` structure.

```
struct SSAPrelease {
    int     sr_affirmative;

#define SR_SIZE         512
    int     sr_cc;
    char    sr_data[SR_SIZE];
};
```

The elements of this structure are:

`sr_affirmative`: the acceptance indicator; and,

`sr_data`/`sr_cc`: any final data (and the length of that data).

If the call to `SRelRequest` is successful, and the `sr_affirmative` element is set, then the connection has been closed. If the call is successful, but the `sr_affirmative` element is not set (i.e., zero), then the request to close the connection has been rejected by the remote user, and the connection is still open. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Note that if a non-negative value is given to the `secs` parameter and a response is not received within this number of seconds, then the value contained in the `sa_reason` element is `SC_TIMER`. The user can then call the routine `SRelRetryRequest` to continue waiting for a response:

```
int     SRelRetryRequest (sd, secs, sr, si)
int     sd;
```

```
int     secs;
struct SSAPrelease  *sr;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`secs:` the maximum number of seconds to wait for a response (use the manifest constant `NOTOK` if no time-out is desired);

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

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

If the call to `SRelRetryRequest` is successful, and the `sr_affirmative` element is set, then the connection has been closed. If the call is successful, but the `sr_affirmative` element is not set (i.e., zero), then the request to close the connection has been rejected by the remote user, and the connection is still open. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure. As expected, the value `SC_TIMER` indicates that no response was received within the time given by the `secs` parameter.

Upon receiving a S-RELEASE.INDICATION event, the user is required to generate a S-RELEASE.RESPONSE action using the `SRelResponse` routine.

```
int     SRelResponse (sd, status, data, cc, si)
int     sd;
int     status,
        cc;
char   *data;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`status:` the acceptance indicator (either `SC_ACCEPT` if the connection is to be closed, or `SC_REJECTED` otherwise);

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

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

If the call to `SRelResponse` is successful, and if the `result` parameter is set to `SC_ACCEPT`, then the connection has been closed. If the call is successful, but the `result` parameter is not `SC_ACCEPT`, then the connection still remains open. Note that in order to specify a value other tha `SC_ACCEPT` for the `result` parameter to `SRelResponse`, the release token must exist but must not be owned by the user making the call to `SRelResponse`.[3]

3.6 Restrictions on User Data

The *libssap*(3n) contains partial support for the use of unlimited user data for session services. With the exception of the S-DATA and the S-TYPED-DATA services, the initial session service limited the amount of user data that could be present. With the introduction of the unlimited user data addendum[ISO87c, ISO87d] to the session service and protocol, this restriction has been lifted.

During connection establishment, the *libssap*(3n) library will attempt to negotate the use of unlimited user data. If this negotiation fails, then session services which permit user data, other than S-DATA and S-TYPED-DATA, are limited to 512 octets of user data.[4]

If the negotation succeeds, then session services which permit user data, other than S-DATA and S-TYPED-DATA, are limited to 65400 octets of user data, with the exception of the S-CONNECT.REQUEST primitive, which is limited to 10240 octets of user data (the S-CONNECT.RESPONSE primitive is limited to 65528 octets).[5]

There is one further limitation however: although the initiator of a connection can send upto 10240 octets, due to limitations in the UNIX kernel,

[3]Gentle reader, we don't write the standards; we just try to implement them.

[4]Strictly speaking, the S-U-ABORT service permits only 9 octets of user data. This limitation is unreasonable — upto 512 octets will be permitted.

[5]Strictly speaking, the amount should be unlimited. However this full generality is not available at this time.

if the *tsapd*(8c) is dispatching based on session selector, then a responder can accept upto approximately 1536 octets. To avoid this problem, have *tsapd*(8c) dispatch based on transport selector (see Section 10.1 in *Volume One*).

3.7 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `si` parameter given to the routine. The `si_abort` element of the `SSAPindication` structure contains a `SSAPabort` structure detailing the reason for the failure. The `sa_reason` element of this latter structure can be given as a parameter to the routine `SErrString` which returns a null-terminated diagnostic string.

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

3.8 Compiling and Loading

Programs using the *libssap*(3n) library should include `<isode/ssap.h>`. The programs should also be loaded with `-lssap` and `-ltsap` (this latter library contains the routines which implement the transport services used by the session provider).

3.9 An Example

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

There are two parts to the program: initialization and data transfer; release will occur when the standard input has been exhausted. In our example, we assume that the routine `error` results in the process being terminated after printing a diagnostic.

In Figure 3.3, the initialization steps for the source entity, including the outer *C* wrapper, is shown. First, a lookup is done in the ISO services database, and the `SSAPaddr` is initialized. The `SSAPref` is zeroed. Next,

for each token associated with the session requirements, initial ownership of that token is claimed. Finally, the call to `SConnRequest` is made. If the call is successful, a check is made to see if the remote user accepted the connection. If so, the session-descriptor is captured, along with the negotiated requirements and initial token settings.

In Figure 3.4 on page 100, the data transfer loop is realized. The source entity reads a line from the standard input, and then queues that line for sending to the remote side. When an end-of-file occurs on the standard input, the source entity requests release and then gracefully terminates. Although no checking is done in this example, for the calls to `SDataRequest` and `SRelRequest`, on failure a check for the operational error `SC_OPERATION` should be made. For `SDataRequest`, this would occur when the data token was not owned by the user; for `SRelRequest`, this would occur when the release token was not owned by the user.

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


static int requirements = SR_HALFDUPLEX | SR_NEGOTIATED;

static int owned = 0;


                                                                    10
/* ARGSUSED */

main (argc, argv, envp)
int       argc;
char  **argv,
     **envp;
{
   int     sd,
             settings;
   char    buffer[BUFSIZ];                                          20
   register struct SSAPaddr   *sz;
   struct SSAPref  sfs;
   register struct SSAPref *sf = &sfs;
   struct SSAPconnect  scs;
   register struct SSAPconnect *sc = &scs;
   struct SSAPrelease  srs;
   register struct SSAPrelease *sr = &srs;
   struct SSAPindication   sis;
   register struct SSAPindication *si = &sis;
   register struct SSAPabort *sa = &si -> si_abort;                 30
   register struct isoservent *is;

   if ((is = getisoserventbyname ("sink", "ssap")) == NULL)
         error ("ssap/sink:  unknown provider/entity pair");
   if (argc != 2)
         error ("usage:  source \"host\"");

...
```

Figure 3.3: Initializing the SSAP source entity