- Notes if the responder initially owns the token (the `ST_RESP_VALUE` case); or,
- Gives ownership of the token to the initiator if the choice is at the responder's discretion (the `ST_CALL_VALUE` case).

3.3.2 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 local system implements the desired TSAP entity. In the case of the session service, the *tsapd* program contains the bootstrap for the session provider. The daemon will again consult the ISO services database to determine which program on the system implements the desired SSAP entity.

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 SSAP state contained in the magic arguments. This is done by calling the routine `SInit`, which on a successful return, is equivalent to a S-CONNECT.INDICATION from the session service provider.

```
int     SInit (vecp, vec, ss, si)
int     vecp;
char  **vec;
struct SSAPstart *ss;
struct SSAPindication *si;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

`ss`: a pointer to a `SSAPstart` 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.

```
#include <ssap.h>

...

int     owned = 0;
int     required;               /* initialized from connection negotation */
int     settings;               /*  .. */

...
                                                                        10
#define dotoken(requires,shift,bit,type) \
{ \
   if (requirements & requires) \
        switch (settings & (ST_MASK << shift)) { \
           case ST_CALL_VALUE << shift: \
                settings &= ~(ST_MASK << shift); \
                settings |= ST_INIT_VALUE << shift; \
                break; \
 \
           case ST_INIT_VALUE: \                                        20
                break; \
 \
           case ST_RESP_VALUE: \
                owned |= bit; \
                break; \
 \
           default: \
                error ("initial %s token setting", type); \
        } \
}                                                                       30

   dotokens ();

#undef  dotoken
```

Figure 3.2: Determining the Initial Token Settings

If `SInit` is successful, it returns information in the `ss` parameter, which is a pointer to a `SSAPstart` structure.

```
struct SSAPstart {
    int     ss_sd;

    struct SSAPref  ss_connect;

    struct SSAPaddr ss_calling;
    struct SSAPaddr ss_called;

    int     ss_requirements;
    int     ss_settings;
    long    ss_isn;

    int     ss_ssdusize;

    struct QOStype ss_qos;

#define SS_SIZE         512
    int     ss_cc;
    char    ss_data[SS_SIZE];
};
```

The elements of this structure are:

`ss_sd`: the session-descriptor to be used to reference this connection;

`ss_connect`: the connection identifier (a.k.a. SSAP reference) used by the initiator;

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

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

`ss_requirements`: the proposed session requirements;

`ss_settings`: the initial token settings;

`ss_isn`: the initial serial-number;

`ss_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`ss_qos`: the quality of service on the connection (see Section 4.6.2); and,

`ss_data/ss_cc`: any initial data (and the length of that data).

The `ss_connect` element is a `SSAPref` structure, which is passed transparently by the session service.

```
struct SSAPref {
#define SREF_USER_SIZE 64
    u_char sr_ulen;
    char    sr_udata[SREF_USER_SIZE];

#define SREF_COMM_SIZE 64
    u_char  sr_clen;
    char    sr_cdata[SREF_COMM_SIZE];

#define SREF_ADDT_SIZE 4
    u_char  sr_alen;
    char    sr_adata[SREF_ADDT_SIZE];

    u_char  sr_vlen;
    char    sr_vdata[SREF_USER_SIZE];
};
```

The elements of this structure are:

`sr_udata/sr_ulen`: the user reference (and length of that reference, which may not exceed `SREF_USER_SIZE` octets);

`sr_cdata/sr_clen`: the common reference (and length of that reference, which may not exceed `SREF_COMM_SIZE` octets);

`sr_adata/sr_adata`: the additional reference (and length of that reference, which may not exceed `SREF_ADDT_SIZE` octets); and,

`sr_vdata/sr_vlen`: a second user reference (and length of that reference, which may not exceed `SREF_USER_SIZE` octets), which is used only when starting or resuming an activity.

If the call to the SInit routine is not successful, then a S-P-ABORT.INDICATION event is simulated, and the relevant information is returned in a SSAPindication structure.

```
struct SSAPindication {
    int     si_type;
#define SI_DATA         0x00
#define SI_TOKEN        0x01
#define SI_SYNC         0x02
#define SI_ACTIVITY     0x03
#define SI_REPORT       0x04
#define SI_FINISH       0x05
#define SI_ABORT        0x06

    union {
        struct SSAPdata si_un_data;
        struct SSAPtoken si_un_token;
        struct SSAPsync si_un_sync;
        struct SSAPactivity si_un_activity;
        struct SSAPreport si_un_report;
        struct SSAPfinish si_un_finish;
        struct SSAPabort si_un_abort;
    }   si_un;
#define si_data         si_un.si_un_data
#define si_token        si_un.si_un_token
#define si_sync         si_un.si_un_sync
#define si_activity     si_un.si_un_activity
#define si_report       si_un.si_un_report
#define si_finish       si_un.si_un_finish
#define si_abort        si_un.si_un_abort
};
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Hence, on a failure return, one first coerces a pointer to the SSAPabort structure contained therein, and then consults the elements of that structure.

```
struct SSAPabort {
    int     sa_peer;
```

```
    int     sa_reason;

#define SA_SIZE         512
    int     sa_cc;
    char    sa_data[SA_SIZE];
};
```

The elements of a `SSAPabort` structure are:

`sa_peer`: if set, indicates that a user-initiated abort occurred (a S-U-ABORT.INDICATION event); if not set, indicates that a provider-initiated abort occurred (a S-P-ABORT.INDICATION event);

`sa_reason`: the reason for the provider-initiated abort (codes are listed in Table 3.4), meaningless if the abort is user-initiated; and,

`sa_data`/`sa_cc`: any abort data (and the length of that data) from the peer (if `sa_peer` is set) or a diagnostic string from the provider (if `sa_peer` is not set).

> **NOTE:** Although both [ISO87b] and [CCI84a] both require a maximum length of 9 octets for a user-initiated abort, the current implementation permits up to 512 octets to be used. Without this freedom, higher-layer protocols which use presentation encoding mechanisms would be unable to successfully use the session abort facility.

After examining the information returned by `SInit` on a successful call (and possibly after examining the argument vector), the responder should either accept or reject the connection. For either response, the responder should use the `SConnResponse` routine (which corresponds to the S-CONNECT.RESPONSE action).

```
int     SConnResponse (sd, ref, called, result, requirements,
                settings, isn, data, cc, si)
int     sd;
struct SSAPref *ref;
struct SSAPaddr *called;
```

Provider-initiated Abort (fatal)	
SC_SSAPID	SSAP identifier unknown
SC_SSUSER	SS-user not attached to SSAP
SC_CONGEST	Congestion at SSAP
SC_VERSION	Proposed protocol versions supported
SC_ADDRESS	Address unknown
SC_REFUSED	Connect request refused on this network connection
SC_TRANSPORT	Transport disconnect
SC_ABORT	Provider-initiated abort
SC_PROTOCOL	Protocol error
User-initiated Abort (fatal)	
SC_NOTSPECIFIED	Reason not specifed
SC_CONGESTION	Temporary congestion
SC_REJECTED	Rejected
Interface Errors (non-fatal)	
SC_PARAMETER	Invalid parameter
SC_OPERATION	Invalid operation
SC_TIMER	Timer expired
SC_WAITING	Indications waiting

Table 3.4: SSAP Failure Codes

```
int     result,
        requirements,
        settings,
        cc;
long isn;
char   *data;
struct SSAPindication *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`ref:` the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

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

`result:` the acceptance indicator (either `SC_ACCEPT` if the connection is to be accepted, or one of the user-initiated abort error codes listed in Table 3.4 on page 55);

`requirements:` the responder's session requirements (this may not include any requirements not listed in the initiator's session requirements);

`settings:` the initial token settings (for each token, if the initiator specified `ST_CALL_VALUE`, then the responder should specify either `ST_INIT_VALUE` or `ST_RESP_VALUE`; instead, if the initiator specified `ST_INIT_VALUE`, and the responder wants the token, it can specify the value `ST_RSVD_VALUE`. This is interpreted by the service provider as a "tokens please" request);

`isn:` the initial serial-number;

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

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

If the call to SConnResponse is successful, and if the result parameter is set to SC_ACCEPT, then connection establishment has completed and the users of the session service now operate as symmetric peers. If the call is successful, but the result parameter is not SC_ACCEPT, then the connection has been rejected and the responder may exit. Otherwise, if the call fails and the reason is not an interface error (see Table 3.4 on page 55), then the connection is closed.

Note that when the responder rejects the connection, it need not supply meaningful values for the the requirements, settings, and isn parameters. The data/cc parameters are also optional, but it is recommended that the responder return some diagnostic information.

3.3.3 Client Initialization

A program wishing to connect to another user of session services calls the SConnRequest routine, which corresponds to the S-CONNECT.REQUEST action.

```
int     SConnRequest (ref, calling, called, requirements,
                settings, isn, data, cc, qos, sc, si)
struct SSAPref *ref;
struct SSAPaddr *calling,
                *called;
int     requirements,
        settings,
        cc;
long isn;
char   *data;
struct QOStype *qos;
struct SSAPconnect *sc;
struct SSAPindication *si;
```

The parameters to this procedure are:

ref: the connection identifier used by the initiator (consult page 52 for a description of the SSAPref structure);

calling: the SSAP address of the responder;

called: the SSAP address of the initiator;

`requirements`: the session requirements;

`settings`: the initial token settings;

`isn`: the initial serial-number;

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

`qos`: the quality of service on the connection (see Section 4.6.2);

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

`si`: a pointer to a `SSAPindication` structure, which is updated only if either:

- the call fails; or,
- the call succeeds, but the value of the `sc_result` element of the `sc` parameter is not `SC_ACCEPT` (see below).

If the call to `SConnRequest` is successful (a successful return corresponds to a S-CONNECT.CONFIRMATION event), then information is returned in the `sc` parameter, which is a pointer to a `SSAPconnect` structure.

```
struct SSAPconnect {
    int     sc_sd;

    struct SSAPref  sc_connect;

    struct SSAPaddr sc_responding;

    int     sc_result;

    int     sc_requirements;
    int     sc_settings;
    int     sc_please;
    lnog    sc_isn;

    int     sc_ssdusize;
```