```
...

static int  ros_invoke (sd, rox)
int     sd;
register struct RoSAPinvoke *rox;
{
    int     result;
    register struct dispatch   *ds;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;                          10
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    for (ds = dispatches; ds -> ds_operation != APDU_UNKNOWN; ds++)
        if (ds -> ds_operation == rox -> rox_op)
            break;

    if (ds -> ds_operation == APDU_UNKNOWN) {
        if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_UNRECOG,
                    ROS_NOPRIO, roi) == NOTOK)
            error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));   20
        goto out;
    }

    if (rox -> rox_nolinked == 0) {
        if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_LINKED,
                    ROS_NOPRIO, roi) == NOTOK)
            error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));
        goto out;
    }
                                                                           30
    switch (result = (*ds -> ds_vector) (rox)) {
        case ERROR_UNKNOWN:
            if (RoResultRequest (sd, rox -> rox_id, rox -> rox_op,
                        rox -> rox_args, ROS_NOPRIO, roi) == NOTOK)
                error ("RO-RESULT.REQUEST: %s",
                        RoErrString (rop -> rop_reason));
            break;

        default:
            if (RoErrorRequest (sd, rox -> rox_id, result, rox -> rox_args,    40
                        ROS_NOPRIO, roi) == NOTOK)
                error ("RO-ERROR.REQUEST: %s",
                        RoErrString (rop -> rop_reason));
            break;

        case ERROR_MISTYPED:
            if (RoURejectRequest (sd, &rox -> rox_id, ROS_IP_MISTYPED,
                        ROS_NOPRIO, roi) == NOTOK)
                error ("RO-U-REJECT.REQUEST: %s",
                        RoErrString (rop -> rop_reason));                  50
            break;
    }

out: ;
    ROXFREE (rox);
}

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

```
...

static int  ros_result (sd, ror)
int     sd;
register struct RoSAPresult *ror;
{
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;
                                                                        10
    if (RoURejectRequest (sd, &ror -> ror_id, ROS_RRP_UNRECOG, ROS_NOPRIO, roi)
            == NOTOK)
        error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));

    RORFREE (ror);
}


static int  ros_error (sd, roe)
int     sd;                                                             20
register struct RoSAPerror *roe;
{
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (RoURejectRequest (sd, &roe -> roe_id, ROS_REP_UNRECOG, ROS_NOPRIO, roi)
            == NOTOK)
        error ("RO-U-REJECT.REQUEST: %s", RoErrString (rop -> rop_reason));
                                                                        30
    ROEFREE (roe);
}


static int  ros_ureject (sd, rou)
int     sd;
register struct RoSAPureject *rou;
{
/* handle rejection here... */
}                                                                       40


static int  ros_preject (sd, rop)
int     sd;
register struct RoSAPpreject *rop;
{
    if (ROS_FATAL (rop -> rop_reason))
        error ("RO-REJECT-P.INDICATION: %s", RoErrString (rop -> rop_reason));

/* handle temporary failure here... */                                  50
}

...
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

```
...

static int  ros_finish (sd, acf)
int     sd;
struct AcSAPfinish *acf;
{
    struct AcSAPindication  acis;
    register struct AcSAPabort *aca = &acis.aci_abort;

    ACFFREE (acf);                                                          10

    if (AcRelResponse (sd, ACS_ACCEPT, ACR_NORMAL, NULLPEP, 0, &acis) == NOTOK)
        error ("A-RELEASE.RESPONSE: %s", AcErrString (aca -> aca_reason));

    error ("association released");
}
```

Figure 3.2: The request/reply loop for the generic ROS responder (continued)

If the event was caused by a request to invoke an operation, then `ros_invoke` is called. This routine consults a dispatch table to find the function which will execute the operation. Based on the return value of the function, either the result is returned, an error is returned, or the request to perform the operation is rejected.

If the event was caused by a reply to the invocation of an operation (i.e., either results or an error), this is rejected. The basic service discipline prohibits this event from happening to responders; hence, the `ros_result` and `ros_error` routines are used as stubs.

If the event was caused by a rejection of a previous reply, then the `ros_ureject` routine is called to handle this. If the event was caused by the provider initiating a rejection, then the `ros_preject` routine is called to handle this.

Finally, if the event was caused by the association being released, the `ros_finish` routine is called to handle this.

3.6.2 The Generic Client

As the previous example described — in great detail — how users of the remote operations services employ the *librosap*(3n) library, we now present a short example of how a client connects to the server.

In Figure 3.3, the initialization steps for the generic initiator, including

the outer *C* wrapper, is shown. First, the application-entity information and presentation address are looked-up. Second, the application context and protocol control information objects are looked-up (and copied, since `ode2oid` returns a pointer to a static area). Next, the session connection reference is initialized. Then, the call to `AcAssocRequest` is made. The arguments to this routine are the minimal required for remote operations. If the call succeeds, then the client checks to see if the association is successfully established. If so, the association descriptor is captured, and then the routine `invoke`, which is not described in this example, is called. This routine presumably requests remote operations from the responder previously described. Upon the return of the `invoke` routine, the association is (forcibly) released, and the program exits.

```
#include <stdio.h>
#include "generic.h"              /* defines OPERATIONS and ERRORS */
#include <isode/rosap.h>


static char *myservice = "service";

static char *mycontext = "iso service";
static char *mypci = "service pci version m.n";
                                                                        10

main (argc, argv, envp)
int       argc;
char  **argv,
     **envp;
{
   int       sd;
   struct SSAPref sfs;
   register struct SSAPref *sf;
   register struct PSAPaddr *pa;                                        20
   struct AcSAPconnect accs;
   register struct AcSAPconnect   *acc = &accs;
   struct AcSAPrelease acrs;
   register struct AcSAPrelease   *acr = &acrs;
   struct AcSAPindication  acis;
   register struct AcSAPindication *aci = &acis;
   register struct AcSAPabort *aca = &aci -> aci_abort;
   struct RoSAPindication  rois;
   register struct RoSAPpreject   *rop = &rois.roi_preject;
   register AEI aei;                                                    30
   register OID ctx, pci;
   struct PSAPctxlist pcs;
   register struct PSAPctxlist *pc = &pcs;

   if ((aei = _str2aei (argv[1], myservice, mycontext, 0,
                                NULLCP, NULLCP)) == NULLAEI)
         error ("unable to resolve service:  %s", PY_pepy);
   if ((pa = aei2addr (aei)) == NULLPA)
         error ("address translation failed");
   if ((ctx = ode2oid (mycontext)) == NULLOID)                          40
         error ("%s:  unknown object descriptor", mycontext);
   if ((ctx = oid_cpy (ctx)) == NULLOID)
         error ("oid_cpy");
   if ((pci = ode2oid (mypci)) == NULLOID)
         error ("%s:  unknown object descriptor", mypci);
   if ((pci = oid_cpy (pci)) == NULLOID)
         error ("oid_cpy");
   pc -> pc_nctx = 1;
   pc -> pc_ctx[0].pc_id = 1;
   pc -> pc_ctx[0].pc_asn = pci;                                        50
   pc -> pc_ctx[0].pc_atn = NULLOID;

...
```

Figure 3.3: Initializing the generic ROS initiator

3.7 For Further Reading

The ECMA technical report on remote operation services is defined in [ECM85]. The CCITT recommendation describing remote operations, as supported by the reliable transfer service, is [CCI84b]. These both assume the use of old-style associations; the draft CCITT work which assumes the use of new-style associations is defined in [CCI88c] and [CCI88d]. Similarly, the corresponding ISO work is [ISO88e] and [ISO88f].

```
...

    if ((sf = addr2ref (PLocalHostName ())) == NULL) {
        sf = &sfs;
        (void) bzero ((char *) sf, sizeof *sf);
    }

/* read command line arguments here... */

    if (AcAssocRequest (ctx, NULLAEI, NULLAEI, NULLPA, pa, pc, NULLOID,
                        0, ROS_MYREQUIRE, SERIAL_NONE, 0, sf, NULLPEP, 0, NULLQOS,
                        acc, aci)
            == NOTOK)
        error ("A-ASSOCIATE.REQUEST: %s", AcErrString (aca -> aca_reason));

    if (acc -> acc_result != ACS_ACCEPT)
        error ("association rejected:  %s", AcErrString (aca -> aca_reason));

    sd = acc -> acc_sd;
    ACCFREE (acc);

    if (RoSetService (sd, RoPService, &rois) == NOTOK)
        error ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    invoke (sd);                        /* invoke the operations, etc. */

    if (AcRelRequest (sd, ACF_NORMAL, NULLPEP, 0, NOTOK, acr, aci) == NOTOK)
        error ("A-RELEASE.REQUEST: %s", AcErrString (aca -> aca_reason));

    if (!acr -> acr_affirmative) {
        (void) AcUAbortRequest (sd, NULLPEP, 0, aci);
        error ("release rejected by peer:  %d", acr -> acr_reason);
    }

    ACRFREE (acr);

    exit (0);
}
```

Figure 3.3: Initializing the generic ROS initiator (continued)

Chapter 4

Reliable Transfer

The *librtsap*(3n) library implements the reliable transfer service (RTS). The abstraction provided to applications is that of an *association* for reliably transfering data. Most applications use the remote operations facilities, described in the previous chapter, and do not directly use the reliable transfer service.[1] However, for those applications which do not base themselves in remote operations, the reliable transfer interface is used.

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 `rtsap` 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 *librtsap*(3n) library are integer-valued. They

[1] Actually, applications such as message-handling systems explicitly use the reliable transfer service to perform association management, and then optionally utilize the remote operations service for actual communication (otherwise they use reliable transfer directly). Both usages are permitted and encouraged in this implementation.

return the manifest constant `OK` on success, or `NOTOK` otherwise.

4.1 Associations

As briefly mentioned earlier, an association is the binding between two applications. An association is formed when one application, termed the *initiator*, specifies the address of a *responder* and asks the reliable transfer service to establish a connection.

There are three aspects of association management: *association establishment*, *association release*, and, *association abort*. Each of these are now described in turn. All of these facilities rely on the mechanisms described in Section 2.2 on page 17.

4.1.1 Association Establishment

The *librtsap*(3n) library distinguishes between the user which started an assocoation, the *initiator*, and the user which was subsequently bound to the association, the *responder*. We sometimes term these two entities the *client* and the *server*, respectively.

Addresses

Addresses for the reliable transfer service entity consist of two parts: a presentation address (as discussed in Section 2.2 on page 16 of *Volume Two*), and application-entity information (as discussed in Section 2.2.1 on page 17).

In Figure 2.1 on page 21, an example of how these components are determined is given.

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 application context. Once the program has been ascertained, the daemon runs the program with any argument 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 recapcture the RTSE state contained in the magic arguments. This is done by

calling the routine `RtInit`, which on a successful return, is equivalent to a RT-OPEN.INDICATION from the reliable transfer service provider.

```
int     RtInit (vecp, vec, rts, rti)
int     vecp;
char  **vec;
struct RtSAPstart *rts;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`vecp`: the length of the argument vector;

`vec`: the argument vector;

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

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

If `RtInit` is successful, it returns information in the `rts` parameter, which is a pointer to a `RtSAPstart` structure.

```
struct RtSAPstart {
    int     rts_sd;

    int     rts_mode;
#define RTS_MONOLOGUE   0
#define RTS_TWA         1

    int     rts_turn;
#define RTS_INITIATOR   0
#define RTS_RESPONDER   1

    PE      rts_data;

    struct AcSAPstart rts_start;
};
```

The elements of this structure are: