```
                /* check fds for activity, do other things */
    }
```

Please note that `isodeserver` and `iserver_wait` implement one possible discipline for association management. Many others are possible, depending on the needs of the service being provided. Further, note that while the text above and the example below are expressed in terms of association control (i.e., they make use of `AcInit` and `AcAssocResponse`), this facility will provide similar support at any other layer in the system (e.g., `isodeserver` can be used for transport or session entities).

Also note that as these routines use the `TNetAccept` routines, child process are collected automatically. If you wish to start child processes and wait for their exit status, you should take note of the warnings associated with the `TNetAccept` procedure (see Section 4.7 on page 130 of *Volume Two*).

## 2.6 Restrictions on User Data

To quote the [ISO88a] specification:

> **NOTE:** Use of the services ... may be subject to some constraints from (the) session services until work on providing unlimited length user data field parameters on session primitives is completed.

## 2.7 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `aci` parameter given to the routine. The `aci_abort` element of the `AcSAPindication` structure encodes the reason for the failure. One coerces the pointer to an `AcSAPabort` structure, and consults the `aca_reason` element of this latter structure. This element can be given as a parameter to the routine `AcErrString` which returns a null-terminated diagnostic string.

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

The  macro can be used to determine if the failure is fatal (the association has been lost).

```
int     ACS_FATAL(r)
int     r;
```

For protocol purists, the `ACS_OFFICIAL` macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     ACS_OFFICIAL (r)
int     r;
```

## 2.8 Compiling and Loading

Programs using the *libacsap*(3n) library should include `<isode/acsap.h>`. The programs should also be loaded with `-lisode`.

## 2.9 An Example

One example of the use of the *libacsap*(3n) library is found in Section 3.6.1 on page 75. This example is straight-forward in presenting the interaction of association control and remote operations. Now let's consider how to rewrite the server to use the facilities described above in Section 2.5. Instead of using an asynchronous interface, a synchronous interface will be employed. Only Figure 3.1 from the example in Section 3.6.1 need be changed.

We assume that there are three exception-logging routines: *fatal*, which prints a diagnostic and terminates the process; *error*, which prints a diagnostic and then executes the statement

```
longjmp (toplevel, NOTOK);
```

and, *warn*, which simply prints a diagnostic.

In Figure 2.3, the replacement routines are shown. First, the application-entity title for the service is deteremined. The routine `isodeserver` is then called with its requisite arguments. If the routine fails, the process terminates after printing a diagnostic. Otherwise the process exits. In the case where the *tsapd*(8c) daemon invokes a new instance of the server each time an incoming

connection is received, `isodeserver` will return after that assocation has been released. In the case where all incoming connections are given to a single instance of the server, then `isodeserver` returns only if a fatal error is detected.

The routine `ros_init` is called for each incoming connection. First, the ACSE state is re-captured by calling `AcInit`. If this succeeds, then any command line arguments are parsed. These arguments are present only if the server was invoked by the *tsapd*(8c) daemon. Assuming that the responder is satisfied with the proposed association, the routine then calls `AcAssocResponse` to accept the association. Then, `RoSetService` is called to set the underlying service to be used for remote operations. Finally, the association-descriptor is returned to `isodeserver`.

The routine `ros_work` is called when activity occurs on an association. The routine sets a global return vector using *setjmp* (3) and then calls `RoWaitRequest` to read the next indication. This usually results in the routine `ros_indication` (found in Figure 3.2 on page 77) being called. If the association was not released, then `ros_work` returns `OK`. Otherwise if some error occurred, use of the routine `error` will cause control to return to the `setjmp` call. In this case, `AcUAbortRequest` is called to make sure that the association is (ungracefully) released, then `NOTOK` is returned to `isodeserver`.

The routine `ros_lose` simply logs the failure of `TNetAccept` when called from `isodeserver`.

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


static char *myservice = "service";
static char *mycontext = "context";
                                                                        10
static jmp_buf toplevel;


int     ros_init (), ros_work (), ros_lose ();


main (argc, argv, envp)
int     argc;
char  **argv,
     **envp;                                                            20
{
    AEI     aei;
    struct TSAPdisconnect tds;
    register struct TSAPdisconnect *td = &tds;

    if ((aei = _str2aei (PLocalHostName (), myservice, mycontext, 0,
                         NULLCP, NULLCP)) == NULLAEI)
        fatal ("unable to resolve service:  %s", PY_pepy);

    if (isodeserver (argc, argv, aei, ros_init, ros_work, ros_lose, td)  30
            == NOTOK) {
        if (td -> td_cc > 0)
            fatal ("isodeserver:  [%s] %*.*s", TErrString (td -> td_reason),
                    td -> td_cc, td -> td_cc, td -> td_data);
        else
            fatal ("isodeserver:  [%s]", TErrString (td -> td_reason));
    }

    exit (0);
}                                                                       40

...
```

Figure 2.3: The generic ROS server

```
...

static int  ros_init (vecp, vec)
int         vecp;
char  **vec;
{
    int         sd;
    struct AcSAPstart   acss;
    register struct AcSAPstart *acs = &acss;
    struct AcSAPindication  acis;                                          10
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort   *aca = &aci -> aci_abort;
    register struct PSAPstart *ps = &acs -> acs_start;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (AcInit (vecp, vec, acs, aci) == NOTOK) {
        warn ("initialization fails:  %s", AcErrString (aca -> aca_reason));
        return NOTOK;                                                      20
    }

    sd = acs -> acs_sd;
    ACSFREE (acs);

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

    if (AcAssocResponse (sd, ACS_ACCEPT, ACS_USER_NULL, NULLOID, NULLAEI,
                    &ps -> ps_called, NULLPC, ps -> ps_defctxresult,
                    ps -> ps_prequirements, ps -> ps_srequirements,        30
                    SERIAL_NONE, ps -> ps_settings, &ps -> ps_connect,
                    NULLPEP, 0, aci) == NOTOK) {
        warn ("A-ASSOCIATE.RESPONSE: %s", AcErrString (aca -> aca_reason));
        return NOTOK;
    }

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        fatal ("RoSetService:  %s", RoErrString (rop -> rop_reason));

    return sd;                                                             40
}

...
```

Figure 2.3: The generic ROS server (continued)

```
...

static int  ros_work (fd)
int       fd;
{
   int       result;
   struct AcSAPindication  acis;
   struct RoSAPindication  rois;
   register struct RoSAPindication *roi = &rois;
   register struct RoSAPpreject   *rop = &roi -> roi_preject;                    10

   switch (setjmp (toplevel)) {
         case OK:
            break;

         default:
            (void) AcUAbortRequest (fd, NULLPEP, 0, &acis);
            return NOTOK;
   }
                                                                                 20
   switch (result = RoWaitRequest (fd, OK, roi)) {
         case NOTOK:
            if (rop -> rop_reason == ROS_TIMER)
                  break;
         case OK:
         case DONE:
            ros_indication (fd, roi);
            break;

         default:                                                                30
            fatal (NULLCP, "unknown return from RoWaitRequest=%d", result);
   }

   return OK;
}

...
```

Figure 2.3: The generic ROS server (continued)

```
...

static int  ros_lose (td)
struct TSAPdisconnect *td;
{
    if (td -> td_cc > 0)
        fatal ("TNetAccept:  [%s] %*.*s", TErrString (td -> td_reason),
                td -> td_cc, td -> td_cc, td -> td_data);
    else
        fatal ("TNetAccept:  [%s]", TErrString (td -> td_reason));
}
```

Figure 2.3: The generic ROS server (continued)

## 2.10 For Further Reading

The ISO specification for association control services is defined in [ISO88a]. The corresponding protocol definition is [ISO88b].

# Chapter 3

# Remote Operations

The *librosap*(3n) library implements the Remote Operations Service (ROS). Three service disciplines are implemented: when the ECMA interpretation of the ROS is used, we term this the *basic* service discipline; when the CCITT X.400 interpretation is used, we term this the *advanced* service discipline; and, when the new ISO and CCITT MOTIS interpretation is used, we term this the *complete* service discipline. The advanced discipline is somewhat less restrictive than the basic discipline, at the cost of requiring a more complex implementation on the part of both the provider and the user. The complete discipline, in addition to all of the facilities provided by the advanced discipline, also supports the notion of *linked* operations.

The abstraction provided to applications is that of an *association* for remote operations. An association is a binding between two users: the *initiator* of the association, and the *responder* to the association. Once an association is established, the initiator requests the responder to perform remote operations. The responder in turn attempts these operations, replying with either a result or an error. This process continues until the initiator decides to release the association.

Like most models of OSI services, the underlying assumption is one of an asynchronous environment: 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 `rosap` module in this release presents a synchronous interface. However once the association is established, an asynchronous interface may be selected.

All of the routines in the *librosap*(3n) library are integer-valued. They return the manifest constant `OK` on success, or `NOTOK` otherwise.

## 3.1 Notice

Please read the following important message.

> **NOTE:** The interface described herein is a "raw" interface to the remote operations service. Consult *Volume Four* for a "cooked" interface.

## 3.2 Service Disciplines and Associations

There are three service disciplines for remote operations: *basic*, *remote*, and *complete*. The basic service discipline limits its users by permitting only the initiator to invoke remote operations. Certain applications, e.g., message handling systems, require more freedom than this, along with more reliability.

The Remote Operations Service Element does not establish associations. Consult Section 2.1 to determine which mechanism you should use to manage associations for your application. Since the advanced and complete disciplines are both proper supersets of the basic service discipline, two users of an association can utilize exactly the features of the basic service discipline even though the advanced or complete service discipline has been selected, without a loss of generality.

## 3.3 Remote Operations

Once the association has been established, an association-descriptor is used to reference the association. This is usually the first parameter given to any of the remaining routines in the *librosap*(3n) library. Further, the last parameter is usually a pointer to a `RoSAPindication` structure. If a call to one of these routines fails, then the structure is updated.

```
struct RoSAPindication {
    int     roi_type;
#define ROI_INVOKE      0x00
#define ROI_RESULT      0x01
#define ROI_ERROR       0x02
#define ROI_UREJECT     0x03
#define ROI_PREJECT     0x04
#define ROI_END         0x05
#define ROI_FINISH      0x06

    union {
        struct RoSAPinvoke roi_un_invoke;
        struct RoSAPresult roi_un_result;
        struct RoSAPerror roi_un_error;
        struct RoSAPureject roi_un_ureject;
        struct RoSAPpreject roi_un_preject;
        struct RoSAPend roi_un_end;
        struct AcSAPfinish roi_un_finish;
    }   roi_un;
#define roi_invoke      roi_un.roi_un_invoke
#define roi_result      roi_un.roi_un_result
#define roi_error       roi_un.roi_un_error
#define roi_ureject     roi_un.roi_un_ureject
#define roi_preject     roi_un.roi_un_preject
#define roi_end         roi_un.roi_un_end
#define roi_finish      roi_un.roi_un_finish
};
```

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 `RoSAPpreject` structure contained therein, and then consults the elements of that structure.

```
struct RoSAPpreject {
    int     rop_reason;

    int     rop_id;
    PE      rop_apdu;

#define ROP_SIZE        512
```