```
    int     rop_cc;
    char    rop_data[ROP_SIZE];
};
```

The elements of a `RoSAPpreject` structure are:

`rop_reason:` the reason for the provider-initiated reject (codes are listed in Table 3.1),

`rop_id/rop_apdu:` if an APDU could not be transferred (`rop_reason` is `ROS_APDU`), then this is the invocation ID of the APDU and (possibly) the APDU itself, respectively; and,

`rop_data/rop_cc:` a diagnostic string from the provider.

Note that the `rop_apdu` element is allocated via *malloc*(3) and should be released using the `ROPFREE` macro when no longer referenced. The `ROPFREE` macro behaves as if it was defined as:

```
void    ROPFREE (rop)
struct RoSAPpreject *rop;
```

The macro frees only the data allocated in the `RoSAPpreject` structure and not the structure itself.

On a failure return, if the `rop_reason` element of the `RoSAPpreject` structure is associated with a fatal error, the the association is released. The `ROS_FATAL` macro can be used to determine this.

```
int     ROS_FATAL (r)
int     r;
```

For protocol purists, the `ROS_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     ROS_OFFICIAL (r)
int     r;
```

Finally, some of these routines also take a `priority` parameter indicating the relative importance of the remote operation. Under the basic service discipline, this parameter is ignored. Under the advanced or complete service discipline, each application decides on its own integer-valued definitions. The manifest constant `ROS_NOPRIO` can be used if the application is unconcerned with the priority.

Provider-Initiated Aborts (fatal)	
ROS_ADDRESS	Address unknown
ROS_REFUSED	Connect request refused on this network connection
ROS_SESSION	Session disconnect
ROS_PROTOCOL	Protocol error
ROS_CONGEST	Congestion at RoSAP
ROS_REMOTE	Remote system problem
ROS_DONE	Assocation done via async handler
ROS_ABORTED	Peer aborted association
ROS_RTS	RTS disconnect
ROS_PRESENTATION	Presentation disconnect
ROS_ACS	ACS disconnect
Provider-Initiated Rejects (non-fatal)	
ROS_GP_UNRECOG	Unrecognized APDU
ROS_GP_MISTYPED	Mistyped APDU
ROS_GP_STRUCT	Badly structured APDU
User-Initiated Rejects (fatal)	
ROS_VALIDATE	Authentication failure
ROS_BUSY	Busy

Table 3.1: RoSAP Failure Codes

User-Initiated Rejects (non-fatal)	
ROS_IP_DUP	Duplicate invocation
ROS_IP_UNRECOG	Unrecognized operation
ROS_IP_MISTYPED	Mistyped argument
ROS_IP_LIMIT	Resource limitation
ROS_IP_RELEASE	Initiator releasing
ROS_IP_UNLINKED	Unrecognized linked ID
ROS_IP_LINKED	Linked response unexpected
ROS_IP_CHILD	Unexpected child operation
ROS_RRP_UNRECOG	Unrecognized invocation
ROS_RRP_UNEXP	Result response unexpected
ROS_RRP_MISTYPED	Mistyped result
ROS_REP_UNRECOG	Unrecognized invocation
ROS_REP_UNEXP	Error response unexpected
ROS_REP_RECERR	Unrecognized error
ROS_REP_UNEXPERR	Unexpected error
ROS_REP_MISTYPED	Mistyped parameter
Interface Errors (non-fatal)	
ROS_PARAMETER	Invalid parameter
ROS_OPERATION	Invalid operation
ROS_TIMER	Timer expired
ROS_WAITING	Indications waiting
ROS_APDU	APDU not transferred
ROS_INTERRUPTED	Stub interrupted

Table 3.1: RoSAP Failure Codes (continued)

3.3.1 Selecting an Underlying Service

As should be obvious from Section 3.2, the *librosap*(3n) library supports the use of three different underlying services, depending on the method used to establish the association. The *librosap*(3n) library is constructed in such a way as to avoid loading the object code for all three underlying services when only one is desired. In order to effect this, it is necessary to give the loader a small hint.

Once an association has been established, the `RoSetService` routine should be called.

```
int     RoSetService (sd, bfunc, roi)
int     sd;
IFP     bfunc;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`bfunc:` a *magic* argument, use:

Value	Underlying Service	Routine

`RoRtService`	Reliable Transfer	`RtOpenResponse`
		`RtOpenRequest`
		`RtBeginResponse`
		`RtBeginRequest`

`RoPService`	Presentation	`AcAssocResponse` `AcAssocRequest`
`RoSService`	Session	`RoBeginResponse` `RoBeginRequest`

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

The call to `RoSetService` should be made after a successful return from any of the routines listed above.

3.3.2 Invoking Operations

The RoInvokeRequest routine is used to request an operation to be performed remotely, and corresponds to a RO-INVOKE.REQUEST action. Under the basic service discipline, this action may be taken by only the initiator of an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoInvokeRequest (sd, op, class, args, invoke,
                linked, priority, roi)
int     sd;
int     op,
        class,
        invoke,
       *linked,
        priority;
PE      args;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`op:` the operation code;

`class:` the operation class (either `ROS_SYNC` for a synchronous operation, or `ROS_ASYNC` for an asynchronous one);

`args:` the arguments for the operation;

`invoke:` the invocation ID of this request;

`linked:` the linked ID of this request (only present if the complete service discipline has been selected, use `NULLIP` otherwise);

`priority:` the priority of this request (use `ROS_NOPRIO` if the priority is undetermined); and,

`roi:` a pointer to a `RoSAPindication` structure, which is always updated on synchronous operations, and only updated if the call fails for asynchronous operations.

If the `class` parameter was `ROS_ASYNC`, then `RoInvokeRequest` returns immediately. Otherwise, after queuing the request, the `RoWaitRequest` routine (described in Section 3.3.4 on page 66) is called implicitly to return the reply of the request. Every attempt will be made to return the corresponding reply to the request. Nevertheless, it is the responsibility of the user to verify the invocation ID which is returned.

The routine `RoIntrRequest` is similar to `RoInvokeRequest` with a `class` argument of `ROS_SYNC`: it invokes an operation and then waits for a response. However, if the user generates an interrupt, usually by typing control-C (`^C`), then `RoIntrRequest` will return immediately by simulating a RO-U-REJECT.INDICATION with reason `ROS_INTERRPUTED` (see section 3.3.4 on page 70). This is useful for users of remote operations which support an "abandon" functionality (e.g., the OSI Directory).

```
int     RoIntrRequest (sd, op, args, invoke, linked,
                priority, roi)
int     sd;
int     op,
        invoke,
       *linked,
        priority;
PE      args;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`op`: the operation code;

`args`: the arguments for the operation;

`invoke`: the invocation ID of this request;

`linked`: the linked ID of this request (only present if the complete service discipline has been selected, use `NULLIP` otherwise);

`priority`: the priority of this request (use `ROS_NOPRIO` if the priority is undetermined); and,

`roi`: a pointer to a `RoSAPindication` structure, which is always updated on synchronous operations, and only updated if the call fails for asynchronous operations.

3.3.3 Replying to Requests

When a request to perform a remote operation has been received by the responder to an association, the responder either returns a result or an error (or in some cases, returns nothing at all).

The `RoResultRequest` routine is used to return a result, and corresponds to the RO-RESULT.REQUEST action. Under the basic service discipline, this action may only be taken by the responder to an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoResultRequest (sd, invoke, op, result, priority, roi)
int     sd;
int     invoke,
        op,
        priority;
PE      result;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`invoke`: the invocation ID of the request corresponding to this reply;

`op`: the operation code of the operation performed (meaningful only in the complete service discipline);

`result`: the results of the operation;

`priority`: the priority of this reply; and,

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

The `RoErrorRequest` routine is used to return an error, and corresponds to the RO-ERROR.REQUEST action. Under the basic service discipline, this action may only be taken by the responder to an association; under the advanced or complete service discipline, no such restriction is made.

```
int     RoErrorRequest (sd, invoke, error, params,
                priority, roi)
int     sd;
int     invoke,
        error,
        priority;
PE      params;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`invoke:` the invocation ID of the request corresponding to this reply;

`op:` the error code;

`params:` the parameters for the error;

`priority:` the priority of this request; and,

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

3.3.4 Reading Replies

The `RoWaitRequest` routine is used to await either a request or a reply from the other user.

```
int     RoWaitRequest (sd, secs, roi)
int     sd;
int     secs;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`secs:` the maximum number of seconds to wait for the data (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and,

`roi`: a pointer to a `RoSAPindication` structure, which is always updated.

Unlike the other routines in the *librosap*(3n) library, the `RoWaitRequest` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading a request or a reply), or `DONE` (when something else happens).

If the call to `RoWaitRequest` returns the manifest constant `NOTOK`, then the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

Otherwise if the call to `RoWaitRequest` returns the manifest constant `OK`, then a request or a reply has arrived. This event is encoded in the `roi` parameter, depending on the value of the `roi_type` element. Currently, when `RoWaitRequest` returns `OK`, the `roi_type` element is set to one of four values:

Value	Event
`ROI_INVOKE`	RO-INVOKE.INDICATION
`ROI_RESULT`	RO-RESULT.INDICATION
`ROI_ERROR`	RO-ERROR.INDICATION
`ROI_UREJECT`	RO-U-REJECT.INDICATION

Otherwise if the call to `RoWaitRequest` returns the manifest constant `DONE`, then some event other than a request or reply arriving has occurred. This event is encoded in the `roi` parameter, depending on the value of the `roi_type` element. Currently, when `RoWaitRequest` returns `DONE`, the `roi_type` element is set to one of two values:

Value	Event
`ROI_END`	RO-END.INDICATION (for old-style associations)
`ROI_FINISH`	A-RELEASE.INDICATION (or RT-CLOSE.INDICATION)

Invocation Indication

When an RO-INVOKE.INDICATION event occurs, the `roi_type` field of the `roi` parameter contains the value `ROI_INVOKE`, and a `RoSAPinvoke` structure is contained inside the `roi` parameter.

```
struct RoSAPinvoke {
    int     rox_id;

    int     rox_linkid;
    int     rox_nolinked;

    int     rox_op;
    PE      rox_args;
};
```

The elements of this structure are:

`rox_id`: the invocation ID of this request;

`rox_linkid`: if `rox_nolinked` is not set, then this contains the invocation ID of the linked request;

`rox_nolinked`: the linked ID indicator (if set, then this invocation is not linked to another operation);

`rox_op`: the operation code; and,

`rox_args`: the arguments for the operation.

Note that the `rox_data` element is allocated via *malloc*(3) and should be released using the `ROXFREE` macro when no longer referenced. The `ROXFREE` macro behaves as if it was defined as:

```
void    ROXFREE (rox)
struct RoSAPinvoke *rox;
```

The macro frees only the data allocated in the `RoSAPinvoke` structure and not the structure itself.

Under the basic service discipline, only the responder to an association will receive this event; it is expected that the user will (eventually) call either the `RoResultRequest`, the `RoErrorRequest`, or perhaps the `RoURejectRequest` routine in response.